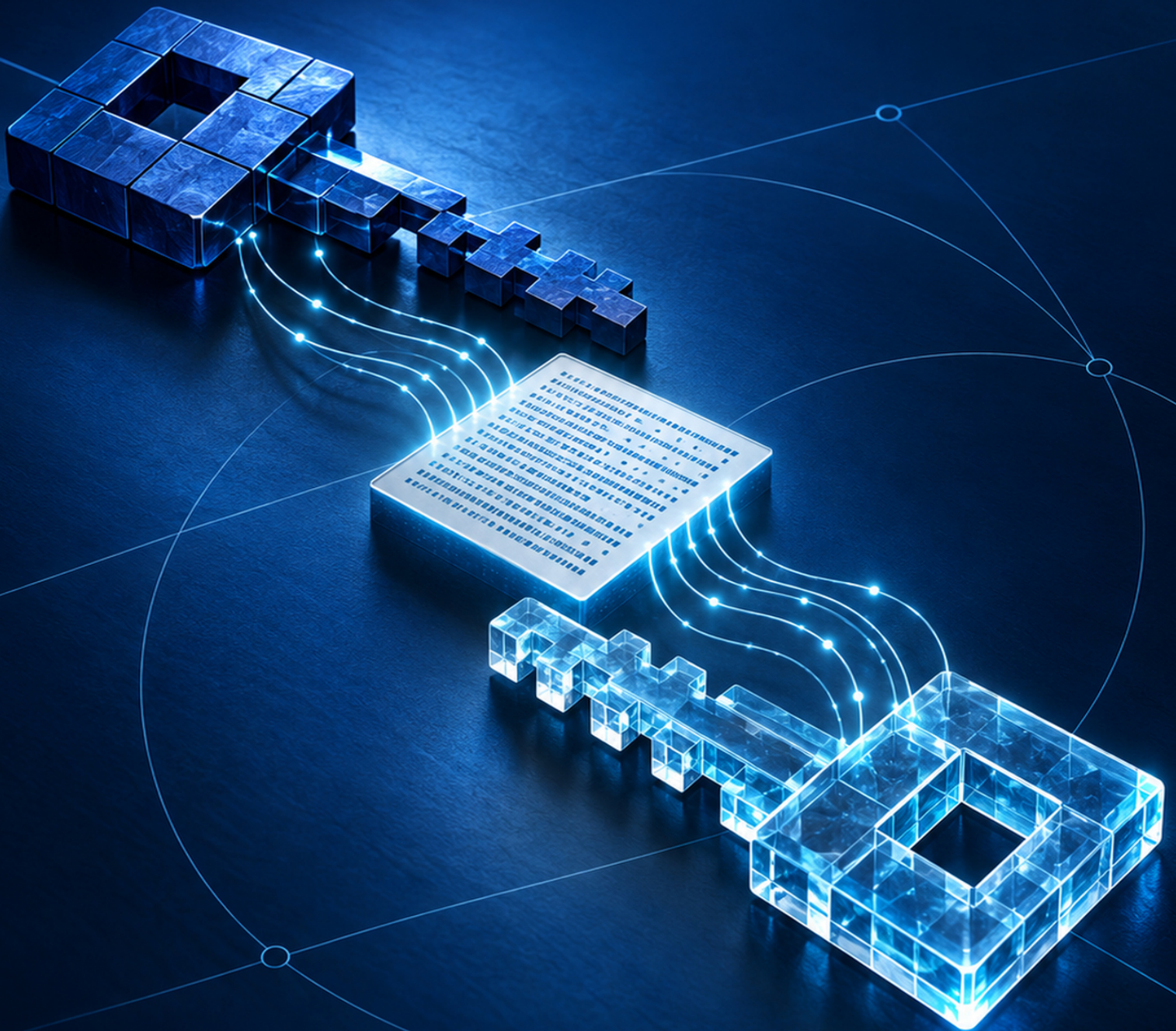


CRYPTOGRAPHY

How Mathematics Protects
Our Digital World



NASREDDINE HAQIQ

Cryptography

How Mathematics Protects Our Digital World

Nasreddine Haqiq

2026

Copyright © 2026 Nasreddine Haqiq. First edition, 2026. Third-party works retain their rights.

Cryptography

1. [Cryptography](#)
 1. [Publication details](#)
 2. [Understanding the evidence](#)
 3. [Introduction](#)
 4. [How to Read This Book](#)
 5. [A Guide to the Notation](#)
2. [I. Secrets and Trust](#)
 1. [1. From Secret Writing to Modern Cryptography](#)
 1. [1.1 Secret writing before computers](#)
 2. [1.2 Why substitution leaks patterns](#)
 3. [1.3 Kerckhoffs and the role of the key](#)
 4. [1.4 The one-time pad and perfect secrecy](#)
 5. [1.5 Enigma and changing substitutions](#)
 6. [1.6 Alan Turing and the wider codebreaking effort](#)
 7. [1.7 Shannon's contribution](#)
 8. [1.8 The public-key breakthrough](#)
 2. [2. What Cryptography Protects](#)
 1. [2.1 A message on an untrusted network](#)
 2. [2.2 Keeping a secret and detecting a change](#)
 3. [2.3 Knowing who holds a key](#)
 4. [2.4 What an attacker can see and do](#)
 5. [2.5 What cryptography cannot protect by itself](#)
3. [II. The Mathematics We Need](#)
 1. [3. Bits, Bytes and Remainders](#)
 1. [3.1 From text to bytes](#)
 2. [3.2 Binary and hexadecimal numbers](#)
 3. [3.3 XOR and the reversible mask](#)
 4. [3.4 Arithmetic with a fixed word size](#)
 5. [3.5 Remainders and modular arithmetic](#)
 6. [3.6 Reading formulas and small programs](#)
 2. [4. Primes, Groups and Hard Problems](#)

1. [4.1 Divisibility and prime numbers](#)
 2. [4.2 Greatest common divisors and modular inverses](#)
 3. [4.3 Fast powers by repeated squaring](#)
 4. [4.4 Groups, orders and generators](#)
 5. [4.5 Easy operations and hard reverse problems](#)
3. [5. Probability, Randomness and Security](#)
 1. [5.1 Events, probabilities and independence](#)
 2. [5.2 Guessing a key](#)
 3. [5.3 Why collisions appear surprisingly early](#)
 4. [5.4 Entropy and predictability](#)
 5. [5.5 Random sources and deterministic generators](#)
 6. [5.6 Security goals, assumptions and proofs](#)
4. [III. Working with Secret Keys](#)
 1. [6. AES and ChaCha20](#)
 1. [6.1 What a secret-key cipher does](#)
 2. [6.2 Blocks and streams](#)
 3. [6.3 The AES state and round structure](#)
 4. [6.4 The small finite field inside AES](#)
 5. [6.5 Following one AES transformation](#)
 6. [6.6 Words, rotations and the ChaCha20 quarter round](#)
 7. [6.7 Choosing a cipher within a complete construction](#)
 2. [7. Nonces and Authenticated Encryption](#)
 1. [7.1 Why a block cipher alone is not enough](#)
 2. [7.2 Nonces, IVs and counters](#)
 3. [7.3 What nonce reuse reveals](#)
 4. [7.4 Protecting ciphertext and associated data](#)
 5. [7.5 AES-GCM and ChaCha20-Poly1305](#)
 6. [7.6 Verification failure and message release](#)
 3. [8. Hashes and Message Authentication](#)
 1. [8.1 A digest of a message](#)
 2. [8.2 Preimages, second preimages and collisions](#)
 3. [8.3 SHA-2 and SHA-3 at the right level](#)
 4. [8.4 Why a bare hash does not authenticate a sender](#)
 5. [8.5 MACs and HMAC](#)
 6. [8.6 Length, encoding and context](#)
 4. [9. Passwords and Key Derivation](#)
 1. [9.1 A password is not a random key](#)

2. [9.2 Offline guessing and salts](#)
 3. [9.3 Memory and time as defensive costs](#)
 4. [9.4 Argon2 and password verification](#)
 5. [9.5 Extracting and expanding key material](#)
 6. [9.6 HKDF inputs and separate key purposes](#)
 7. [9.7 Parameters, upgrades and recovery](#)
5. [IV. Public Keys](#)
 1. [10. Agreeing on a Secret](#)
 1. [10.1 The key-distribution problem](#)
 2. [10.2 Diffie–Hellman step by step](#)
 3. [10.3 The active intermediary problem](#)
 4. [10.4 Key-encapsulation mechanisms](#)
 5. [10.5 Hybrid encryption and context binding](#)
 2. [11. RSA and ElGamal](#)
 1. [11.1 Public and private operations](#)
 2. [11.2 How RSA builds a key](#)
 3. [11.3 What factoring and RSA inversion mean](#)
 4. [11.4 Why textbook RSA is unsafe](#)
 5. [11.5 Padding and the RSA-OAEP guarantee](#)
 6. [11.6 ElGamal encryption and its fresh secret](#)
 7. [11.7 Changing a ciphertext without learning the message](#)
 3. [12. Elliptic Curves](#)
 1. [12.1 Points and the group operation](#)
 2. [12.2 From a real curve to a finite field](#)
 3. [12.3 Adding and doubling points](#)
 4. [12.4 Scalar multiplication and its reverse problem](#)
 5. [12.5 X25519 and modern key agreement](#)
 6. [12.6 Encodings, small-order inputs and validation](#)
 4. [13. Signatures, Certificates and Identity](#)
 1. [13.1 Signing and verification](#)
 2. [13.2 A small Schnorr example](#)
 3. [13.3 RSA signatures, ECDSA and EdDSA](#)
 4. [13.4 Nonces, deterministic signing and key reuse](#)
 5. [13.5 Certificates and trust anchors](#)
 6. [13.6 What a valid signature does not prove](#)
6. [V. Building Secure Systems](#)
 1. [14. Managing Keys](#)

1. [14.1 Generating keys with the right source](#)
 2. [14.2 Distribution and trust establishment](#)
 3. [14.3 Storage, access and hardware protection](#)
 4. [14.4 Usage periods, rotation and revocation](#)
 5. [14.5 Backup, recovery and destruction](#)
 6. [14.6 Responding to a key compromise](#)
2. [15. Securing a Connection with TLS 1.3](#)
 1. [15.1 The problem a secure channel solves](#)
 2. [15.2 The handshake in messages](#)
 3. [15.3 Authentication and the transcript](#)
 4. [15.4 Deriving separate traffic keys](#)
 5. [15.5 Records, nonces and key updates](#)
 6. [15.6 Resumption, 0-RTT and replay](#)
 7. [15.7 What the model and implementation assume](#)
3. [16. Secure Messages and Changing Keys](#)
 1. [16.1 Messages when the recipient is offline](#)
 2. [16.2 Initial key agreement and authentication](#)
 3. [16.3 The symmetric-key and DH ratchets](#)
 4. [16.4 Forward security and recovery after compromise](#)
 5. [16.5 PQXDH and post-quantum ratchets](#)
 6. [16.6 Endpoints, backups and metadata](#)
4. [17. How Good Cryptography Fails](#)
 1. [17.1 Secret reuse and poor randomness](#)
 2. [17.2 Padding oracles and observable errors](#)
 3. [17.3 Time, caches and secret-dependent work](#)
 4. [17.4 Encoding and protocol confusion](#)
 5. [17.5 Compromised endpoints and exposed state](#)
 6. [17.6 Lessons from corrected implementations](#)
5. [18. Testing and Verified Software](#)
 1. [18.1 Known-answer tests and independent expectations](#)
 2. [18.2 Negative tests with Wycheproof](#)
 3. [18.3 What functional and memory-safety proofs establish](#)
 4. [18.4 HACLxN, EverCrypt and the remaining trusted components](#)
 5. [18.5 A reproducible library example](#)
7. [VI. The Next Questions](#)
 1. [19. Quantum Computers and Post-Quantum Cryptography.](#)

1. [19.1 What changes in the attack model](#)
2. [19.2 Gates, interference and entanglement](#)
3. [19.3 Quantum hardware, noise and error handling](#)
4. [19.4 Shor and public-key problems](#)
5. [19.5 Grover and generic search](#)
6. [19.6 A first picture of lattices and LWE](#)
7. [19.7 ML-KEM, ML-DSA and hash-based signatures](#)
8. [19.8 Standards, experiments and crypto-agility](#)
9. [19.9 Migration and the limits of forecasts](#)
2. [20. Proving and Computing with Private Data](#)
 1. [20.1 Proving a claim without revealing the witness](#)
 2. [20.2 Interactive proofs and succinct arguments](#)
 3. [20.3 Secret sharing and threshold signatures](#)
 4. [20.4 Computing together with private inputs](#)
 5. [20.5 Computing on encrypted data](#)
 6. [20.6 Privacy, correctness and practical limits](#)
3. [Conclusion: Understanding Digital Trust](#)
4. [Mathematical Reference](#)
 1. [Bytes, powers and XOR](#)
 2. [Remainders and congruence](#)
 3. [Inverses and division](#)
 4. [Powers and group operations](#)
 5. [Probabilities and repeated trials](#)
 6. [Reading a cryptographic interface](#)
5. [Running the Optional Examples](#)
 1. [A recorded environment](#)
 2. [Prepare a separate Python environment](#)
 3. [Copy, run and inspect a listing](#)
 4. [Interpret the evidence](#)
6. [Answers to the Review Questions](#)
 1. [Chapter 1: From Secret Writing to Modern Cryptography](#)
 2. [Chapter 2: What Cryptography Protects](#)
 3. [Chapter 3: Bits, Bytes and Remainders](#)
 4. [Chapter 4: Primes, Groups and Hard Problems](#)
 5. [Chapter 5: Probability, Randomness and Security](#)
 6. [Chapter 6: AES and ChaCha20](#)
 7. [Chapter 7: Nonces and Authenticated Encryption](#)

8. [Chapter 8: Hashes and Message Authentication](#)
9. [Chapter 9: Passwords and Key Derivation](#)
10. [Chapter 10: Agreeing on a Secret](#)
11. [Chapter 11: RSA and ElGamal](#)
12. [Chapter 12: Elliptic Curves](#)
13. [Chapter 13: Signatures, Certificates and Identity](#)
14. [Chapter 14: Managing Keys](#)
15. [Chapter 15: Securing a Connection with TLS 1.3](#)
16. [Chapter 16: Secure Messages and Changing Keys](#)
17. [Chapter 17: How Good Cryptography Fails](#)
18. [Chapter 18: Testing and Verified Software](#)
19. [Chapter 19: Quantum Computers and Post-Quantum Cryptography](#)
20. [Chapter 20: Proving and Computing with Private Data](#)
7. [Key Terms](#)
8. [Notes and Bibliography](#)
 1. [Chapter 1: From Secret Writing to Modern Cryptography](#)
 2. [Chapter 3: Bits, Bytes and Remainders](#)
 3. [Chapter 4: Primes, Groups and Hard Problems](#)
 4. [Chapter 5: Probability, Randomness and Security](#)
 5. [Chapter 6: AES and ChaCha20](#)
 6. [Chapter 7: Nonces and Authenticated Encryption](#)
 7. [Chapter 8: Hashes and Message Authentication](#)
 8. [Chapter 9: Passwords and Key Derivation](#)
 9. [Chapter 10: Agreeing on a Secret](#)
 10. [Chapter 11: RSA and ElGamal](#)
 11. [Chapter 12: Elliptic Curves](#)
 12. [Chapter 13: Signatures, Certificates and Identity](#)
 13. [Chapter 14: Managing Keys](#)
 14. [Chapter 15: Securing a Connection with TLS 1.3](#)
 15. [Chapter 16: Secure Messages and Changing Keys](#)
 16. [Chapter 17: How Good Cryptography Fails](#)
 17. [Additional sources for Chapter 1: Enigma and Alan Turing](#)
 18. [Chapter 18: Testing and Verified Software](#)
 19. [Chapter 19: Quantum Computers and Post-Quantum Cryptography](#)

20. [Further sources for Chapter 19: quantum foundations and hardware](#)
21. [Chapter 20: Proving and Computing with Private Data](#)

Cryptography

Publication details

Copyright © 2026 Nasreddine Haqiq.

First edition, 2026.

Original explanations, teaching examples and instructional diagrams. Third-party publications, software and standards retain their respective rights. Sources are provided for the scientific claims discussed in the text.

This is an independent educational work. Mentioning a research paper, project or standard does not imply endorsement by its authors or maintainers.

Technical examples must be read with their stated assumptions. An educational calculation does not establish that a complete application is secure.

Understanding the evidence

Cryptography connects mathematical ideas to practical decisions. A calculation can show that an intended receiver recovers a message. A security argument asks what an attacker could learn or change under stated conditions. A specification describes the required behavior of a system, while source code and tests help us examine an implementation. These forms of evidence answer different questions. Understanding their scope is part of understanding the result.

The explanations in this book begin with a problem that a reader can recognize. Symbols are introduced with their meanings, and small examples make the steps visible. Examples with small numbers teach the method; they are not instructions for choosing real cryptographic keys. Programming is an optional way to check the reasoning, and the central explanation does not depend on writing code.

Sources identify the work behind historical and technical claims. Original teaching diagrams describe the same ideas in a form suited to this book. Read an unfamiliar formula together with the explanation around it, then ask what the result establishes and which assumptions it needs. That habit will be useful long after any particular software version has changed.

Introduction

Every day, people ask computers to keep secrets, check messages and recognize the right person or service. A medical report should reach the intended patient. A payment instruction should arrive without an unnoticed change. A software update should come from a source we have a reason to trust. These situations involve different questions, even when one application handles them together. Cryptography provides mathematical methods for answering some of those questions. Its value extends from a single stored document to communication among people who have never met. Each setting needs a clear account of what is protected and which decisions still depend on the surrounding system. To understand the methods, we first need to understand the problem, the evidence a receiver examines and the power we allow an attacker. This book is written for readers who want that understanding without already knowing the subject.

Our route begins with familiar messages and the research that changed secret writing into a mathematical science. We then build the tools needed to explain modern systems: bits and bytes, remainders, prime numbers, groups, probability and randomness. Each idea appears with a reason for learning it. Small calculations help us understand general formulas, and the surrounding words explain what their symbols mean. These foundations lead to secret-key encryption, message authentication, password protection, public keys and digital signatures. We ask what each construction achieves, where its assumptions enter the argument and what changes when those assumptions no longer hold. That gives a practical purpose to the mathematics instead of making it a separate obstacle. The aim is to understand how the pieces work together, including the conditions under which a valid calculation gives a useful security guarantee. An equation belongs in the discussion when it makes a real step of that reasoning clearer.

The later chapters connect these ideas to key management, secure internet connections, private messages and the behavior of actual software. We examine failures as well as successful constructions, because a good mathematical design still depends on correct use and implementation. Optional examples let readers reproduce selected results with established

libraries. The final part introduces quantum threats and methods for proving or computing with private data. Throughout the book, we distinguish an assumption, a proved result, an experiment and a proposed application. Sources allow readers to follow the evidence further. You should finish with the ability to ask precise questions about a cryptographic claim, follow its main explanation and recognize the limits of what has been established.

How to Read This Book

Read the opening chapters in order if cryptography is new to you. Chapter 1 follows secret writing, Enigma and the research that led to modern cryptography. Chapter 2 then separates the protection goals that we will use to examine today's systems. The mathematical chapters then provide a common language for the constructions that follow. You do not need to remember every symbol on the first reading. Return to its definition, work through the small example and ask what the operation does. A worked calculation shows intermediate steps so that an unfamiliar result does not appear without a reason. When a later chapter reuses an idea, the explanation will point back to the earlier foundation instead of introducing a second meaning for the same term.

The main route is written for readers without advanced mathematics or programming experience. Optional calculations and code give a deeper view of selected questions. Read their inputs, expected results and assumptions before following the steps. Small keys, fixed test values and simplified messages are learning tools, not settings for a real application. A successful demonstration establishes what happened in that example. A security argument must also describe the attacker, the goal and the assumptions. When an example uses software, its identity and relevant version matter; a result about one component does not automatically describe every application that uses it.

Use the short [Key Terms](#) section when you remember an idea but have forgotten its name. The main explanation defines terms where they are needed; the reference section helps you find them again. For the public-key calculations, the most useful foundations are remainders in [Section 3.5](#), modular inverses in [Section 4.2](#), powers in [Section 4.3](#) and groups in [Section 4.4](#). You can return directly to one of these sections instead of rereading all the mathematics. A displayed formula should be read with the words immediately before and after it: first identify the task, then name the inputs, follow the operation and interpret the result.

The [Mathematical Reference](#) gathers a few recurring operations for quick review. [Running the Optional Examples](#) records the checked software environment and explains how to run the listings, including the two linked TLS examples. These reference pages support a return visit; you do not need to read them before starting Chapter 1.

Use the diagrams with their captions and the surrounding paragraphs. Labels, shapes and line styles carry the meaning so that the figures remain useful in grayscale. Tables are used when comparison or a reference list is easier to follow in columns. Brief questions at the end of a chapter help you check the distinctions in its argument, and their explained answers are grouped in the reference material. Notes and sources are also kept apart from the main discussion. You can read a chapter continuously, then return to a note when you want to check the original work. The purpose of the book is to make the reasoning accessible while preserving the conditions that make it correct.

A Guide to the Notation

Mathematical notation is a compact way to describe a relationship. Read it as a sentence about named objects. A letter has no meaning merely because it appears in a formula; the surrounding explanation must define it. The same letter can have a different local meaning in another subject, so each calculation will state the meanings it needs.

Symbol	How to read it in this book
	A message, when the discussion defines it that way.
	A secret key in a secret-key example. Public-key notation is introduced later.
	A ciphertext: an encrypted representation of a message.
	Apply the operation named to the input . Parentheses show the input.
	The expressions on the two sides have the same value.
	Is an element of: says that belongs to the allowed set .
	An operation produces a value, which is assigned to the name on the left.
	The probability of the event inside the brackets; the probability chapter explains the conditions.

A formula can describe an ordinary procedure. If
 f
means “double the input,” then

$f(3)=6$
says to double 3 and obtain 6. Cryptographic operations have more detailed definitions, but the reading habit is the same: identify the input, follow the

operation and interpret the output. A statement that the intended operation works is different from a statement that an attacker cannot defeat it.

Some notation has a general reading rule. A statement such as

$$0 \leq m < n$$

means that

m

is at least zero and smaller than

n

. The expression

pq

means

p

multiplied by

q

; omitting a multiplication sign does not create a new variable. A superscript that is an exponent describes a power, while a subscript may label which value is being discussed. Letters are names that let us state one rule for many inputs. In a worked example, the text supplies particular numbers for those names. A Greek letter is also a name, not an extra operation.

Later chapters introduce powers, remainders, XOR, groups and probability only when their meanings have been explained. A symbol table is a reminder of that explanation, not a substitute for it. No formula should require guessing which quantity a letter represents or which conditions the result assumes.

I. Secrets and Trust

1. From Secret Writing to Modern Cryptography

Part I: Secrets and Trust

Cryptography begins with a human problem: communicating information without giving an unwanted reader the same access as the intended receiver. Long before computer networks, a person sending an instruction through a messenger or a telegraph line had to consider interception, changes and the reliability of the receiver. There was no single invention that supplied every part of the modern subject. Secret writing offered ways to change the visible message, and studying how those methods failed gradually led to more precise questions about keys, language, probability and the work required of an attacker.

This history is useful because it explains why changing the appearance of a message is not enough. A strange-looking message can still reveal repeated words. A large collection of keys can still have a shortcut that helps an attacker find the right one. We will meet two different ideas of secrecy. **Perfect secrecy** means that observing the ciphertext adds no information about the message in the stated model. **Computational security** instead relies on an attacker being unable to carry out the required attack with feasible resources. The first claim concerns information; the second concerns what can be computed. The examples below explain why that difference matters before we use more formal notation.

We will follow a selected route through that development. Classical substitution shows how structure survives a transformation. Auguste Kerckhoffs separates the general method from the secret key, and the one-time pad gives an example of secrecy that can be explained exactly. Enigma and the work of Alan Turing, the Polish mathematicians and their colleagues show how machine operations can become questions of logical consistency. Claude Shannon provides a mathematical language for information, and public-key research changes how people can establish protection without sharing every secret in advance. This is a history of connected ideas, not a complete catalogue of inventions or a claim that each discovery had only one contributor.

1.1 Secret writing before computers

A **cipher** is a rule-based method for encrypting and decrypting data. A classical cipher can be carried out with a written alphabet, a table or a simple mechanical device. The sender applies a rule to a message, and the receiver applies the corresponding recovery rule. The agreement must include enough detail for both people to perform compatible operations: which symbols are allowed, how the key is interpreted and how unusual characters are handled. Those details may appear small beside the central rule, but a receiver cannot recover an instruction reliably if the two people use different conventions. The practical problem includes operating the method as well as inventing it.

The **plaintext** is the data before encryption, and the **ciphertext** is the representation produced by encryption. A **key** is a value selecting the particular transformation or recovery operation within a defined method. Calling an output ciphertext describes its role; it does not prove that an outsider cannot recover the plaintext. **Cryptanalysis** studies how to defeat a stated protection, for example by recovering a message or learning a key without the intended permission. These names let us distinguish the operation that should work for the receiver from the task that should remain difficult for an attacker.

Two broad operations are useful for understanding many classical examples. In **substitution**, the sender replaces a symbol or group of symbols with another representation. In **transposition**, the sender rearranges positions. A simple transposition of the word MEET might produce ETME; the letters are still present, but their order has changed. A substitution changes the symbols while it may leave their order intact. More elaborate methods combine operations or change the substitution during the message. Kerckhoffs's nineteenth-century discussion examines such distinctions in the context of military communication, where reliability and ease of use were practical requirements.^[1]

For our first example, use the 26 uppercase letters of the English alphabet. Move each letter three positions forward, returning to A after Z. Thus A becomes D, B becomes E and Z becomes C. The number three selects the shift, while the instruction to use an alphabetic shift is the general method. To recover a message, the receiver moves each ciphertext

letter three positions backward. Spaces are left visible in this teaching example so that the operation is easy to follow:

The first M moves to P, each E moves to H and T moves to W. Applying the same rule to the remaining letters produces the second line. Recovery works because the forward movement has a defined inverse. That is a correctness observation. It does not yet tell us whether an outsider can find the instruction.

There are only 26 possible shifts if we include the shift that leaves the alphabet unchanged. Someone who knows the general method can try every possibility and inspect the results. Removing the visible spaces would remove one clue, but it would not create more shifts or change the consistency of the letter replacement. The example is valuable precisely because its weakness is easy to explain. It shows how a method can be reversible and usable while still offering little protection against an informed opponent. We are using it to study that distinction, not to protect a real message.

A codebook provides a related but different arrangement. It may assign an agreed group to an entire word or instruction, so that the receiver looks up the meaning rather than reversing a letter transformation. The book itself then becomes important secret material, and the parties need a way to distribute, protect and update it. The word “code” is often used loosely for secret writing in everyday speech. In this book, we will distinguish a codebook, a cipher and an ordinary data encoding when their roles matter. Encoding data into a public representation does not establish confidentiality merely because the result looks unfamiliar.

Another approach is to hide the existence of a message, for example by placing it within an apparently ordinary carrier. This concerns concealment, often called **steganography**, rather than only changing the meaning available from visible symbols. The distinction is important: a ciphertext may be openly visible while its contents remain protected, and a hidden message may become readable as soon as someone discovers it. Shannon explicitly distinguished concealment systems from the class of secrecy systems studied in his 1949 paper.^[2] Different approaches can be combined,

but finding one kind of protection should not lead us to assume the presence of the other.

These examples also reveal a problem of scale. A method used occasionally by two people may become difficult to operate across a large organization. People need compatible materials, instructions have to arrive in time and mistakes can expose a pattern or prevent recovery. A complicated manual rule can impose a burden that users avoid by taking shortcuts. Cryptography did not become scientific simply by making the transformation more elaborate. It also needed a clear account of what an attacker could exploit and which practical conditions the intended users could maintain.

1.2 Why substitution leaks patterns

The shift cipher belongs to a larger family in which one fixed alphabet replaces another. Instead of preserving the alphabet's order, choose an arbitrary rearrangement: A might become Q, B might become M and so on, with a different output for every input letter. This is a **simple substitution**, also called a monoalphabetic substitution because the same replacement alphabet is used throughout the relevant text. An attacker now faces many more possible alphabets than the 26 shifts. However, the rule still has a structure that can be studied without trying each possible key.

To count the replacement alphabets, there are 26 choices for the output assigned to the first input letter. Once that output is used, there are 25 choices for the next, then 24, and eventually one choice remains. Multiplying these counts gives

$$(1.1)$$

The exclamation mark is read “factorial.” Here it means the product of all the whole numbers from 1 to 26, and the dots mean that the descending sequence continues. The result has 27 decimal digits. This is a large collection of possible alphabets, but the count alone does not establish how difficult it is to solve a particular ciphertext. It only counts one possible search space.

The central weakness is that a fixed replacement preserves relationships between symbols. If E occurs repeatedly in the plaintext, its replacement occurs just as often in the ciphertext. If a word has the pattern of four letters with its middle two equal, the replacement has the same equality pattern. In MEET, the first and last letters differ, while the two middle letters match. The shifted form PHHW preserves that structure. An arbitrary one-to-one substitution would preserve it too, although the actual output letters could be different. Changing the names of symbols does not remove every fact about their arrangement.

Frequency analysis uses information about such repetitions. A reader can count the appearances of each ciphertext symbol and compare those counts with plausible properties of the language or message source. Some letters, combinations and word patterns are more common than others. These observations suggest candidate replacements, which can then be checked against the rest of the message. Kerckhoffs discusses the combination of frequency information and trial in solving classical ciphers; he also warns that very short messages need not exhibit the usual leading letter.^[1] A frequency estimate is evidence to interpret, not a rule that the most common ciphertext symbol must always represent E.

Consider an attacker who suspects that a message is a school equipment order. Repeated administrative phrases, the names of departments or the structure of a quantity field may provide additional clues. If one proposed replacement turns a likely word into nonsense elsewhere, it can be rejected. Other replacements can be combined into a consistent reading. The attack is using relationships across the message, not treating each key as an unrelated possibility. This is why the large count in Equation 1.1 can give a misleading impression of strength when presented without an analysis of the construction.

Some information may also be known directly. A standard greeting, a published form or a message that later becomes public can supply a piece of plaintext corresponding to a captured ciphertext. For a fixed letter substitution, each known pair reveals entries in the replacement alphabet. Knowing part of a message should therefore be considered an attacker capability, not dismissed as an unfair advantage. Later security definitions will deliberately allow certain kinds of known or chosen input. A sound

definition asks whether a construction protects what it is meant to protect even when the attacker has relevant information about other data.

Removing spaces or using unusual symbols can make a puzzle less familiar without eliminating the underlying statistical relationships. A transposition can hide word order while leaving the number of occurrences of each letter unchanged. A method that changes alphabets during a message can hide some simple patterns while introducing other relationships, particularly if its changes repeat. None of these observations is a universal attack on every possible combination. They show why a design has to be analyzed through its actual structure rather than through its visual complexity or the size of a key table.

The same caution applies to a failed attempt to solve a message. A short ciphertext may have several plausible interpretations. A person may lack a useful language model, enough intercepted material or a good attack method. Their failure does not establish that no better method exists, just as a successful recovery of one badly chosen message does not describe every use of a construction. The scientific task is to state the relevant circumstances and explain the source of the information or computational advantage. That task begins by making the method, key and opponent's knowledge explicit.

1.3 Kerckhoffs and the role of the key

In 1883, Auguste Kerckhoffs published *La Cryptographie militaire* in two parts in the *Journal des sciences militaires*. His discussion was concerned with a system that could support continuing military correspondence, rather than a private trick used for one short exchange. Among six requirements, he argued that the method should not depend on remaining secret and should withstand coming into an enemy's possession. This is the basis of what is now called **Kerckhoffs's principle**: security should survive knowledge of the general method, with the required secret concentrated in the key and other explicitly protected values. ^[1]

The distinction has a practical consequence. If an opponent obtains the description of a method, the parties should not have to redesign every device and teach everyone an entirely new procedure to restore the intended protection. They should have a controlled way to change the compromised

secret material while continuing to use a sound general design. This does not mean that replacing a key reverses a disclosure that has already happened. It concerns which part of a system must remain secret and how the system can be operated when exposure is possible. Its value is easier to see in an organization than in an isolated puzzle.

The principle gives the attacker a demanding but useful starting position. We do not assume that an outsider is confused forever by an unusual file format or a hidden implementation detail. The attacker may know the algorithm, inspect a device or obtain a copy of its program. A security argument must explain why that knowledge does not supply the particular secret or defeat the stated goal. An implementation can still have defects, and its behavior may reveal information beyond the abstract algorithm. Knowing the general method is one part of the model, not a promise that all other problems have been solved.

It is also important to distinguish a public design from public source code. Kerckhoffs's principle says that security should not rely on hiding the design. It does not, by itself, settle how a particular program is licensed or distributed. A published algorithm can have an implementation whose source is unavailable, while an open-source implementation can still use its algorithm incorrectly. Conversely, publishing a design does not prove that the design is secure. It makes a particular assumption about the attacker's knowledge unnecessary and allows the actual reasoning to be examined.

Kerckhoffs's other requirements show that the famous principle was part of an operational argument. He discussed communication of keys, the ability to change them, use with the telegraph, portability and the burden placed on the operator. These concerns connect the strength of the transformation to the conditions under which people must use it. His nineteenth-century requirements should not be copied as a modern password policy: for example, a preference for remembering a key without written notes is not a reason to choose a short, predictable key today. The continuing lesson is to make the required protection compatible with reliable operation.

The principle does not repair the substitution cipher by itself. We already gave its general rule to the attacker, and the structure remained weak. A design can satisfy the intention of being openly analyzable and still fail the

security goal. What changes is the standard of explanation. Instead of saying that an opponent probably will not discover a clever trick, we ask whether the method remains secure after the trick is known. The answer then depends on a construction and an argument, not on secrecy about the name of the procedure.

This separation also clarifies **cryptanalysis**, the study of ways to defeat a cryptographic protection goal. An analyst is not necessarily searching for a hidden algorithm. They may know the algorithm exactly and be looking for a better way to infer a key, recover protected information or produce an accepted forgery. The relevant question is which computational or statistical obstacle remains after the public parts are known. A much stronger form of secrecy is possible in a special setting, however: a ciphertext can leave the original uncertainty about the message unchanged. The one-time pad provides an example that we can understand with a few bits.

1.4 The one-time pad and perfect secrecy

For this example, a **bit** is a symbol that can take one of two values, 0 or 1. We will use a message consisting of four bits, without yet attaching those bits to a letter encoding. The sender and receiver share a secret sequence of four bits called a pad. They combine the message and pad one position at a time. The operation is **exclusive OR**, usually written XOR and represented by the symbol

$$\oplus$$

. Its rule is small: matching input bits produce 0, while different input bits produce 1. Thus

$$0 \oplus 0 = 0$$

,

$$1 \oplus 1 = 0$$

,

$$0 \oplus 1 = 1$$

and

$$1 \oplus 0 = 1$$

.

Using lowercase letters for particular values, let
m

be the message,

k

the pad and

c

the ciphertext. Encryption and decryption take the form

$$(1.2)$$

Read the first equality as “combine the message with the pad by XOR to obtain the ciphertext.” Read the second as “combine that ciphertext with the same pad by XOR to recover the message.” XOR is applied separately at each position, so the two inputs must have the corresponding lengths. No carrying from one position to another is involved. The fuller treatment of bits in Chapter 3 will reuse this rule; we need only its four possible single-bit results here.

Take the following particular values to see the operations:

In the first column of bits,

$$1 \oplus 0$$

produces 1. In the second,

$$0 \oplus 1$$

produces 1. The last two positions produce 0 and 1. Applying the pad again reverses the changes. A pad bit of 0 leaves the corresponding message bit unchanged; a pad bit of 1 flips it, and a second flip restores the original. This explains correctness. To explain secrecy, we need conditions on how the pad is obtained and used.

For a four-bit one-time pad, all 16 possible pad values must be equally likely. This is a **uniform** selection: no allowed value is favored over another. The pad must be independent of the message, kept secret from the attacker and used only for this one message. Independence here means that learning the message does not change how the pad was selected. In the general fixed-length version, each message bit has a corresponding

independent, uniformly random pad bit. A chosen or predictable sequence does not meet those conditions merely because it has the right length. The displayed value 0110 is one possible sample used to explain the operation, not a recommendation to reuse a fixed teaching key.

Suppose an observer sees the ciphertext 1101. Any four-bit message could have produced it: for each candidate message, exactly one four-bit pad would give that ciphertext. If the candidate message is 1011, the required pad is 0110. If the candidate is 0000, the required pad is 1101. Other candidate messages have their own matching pads. Since those pads were equally likely and chosen independently of the message, observing this ciphertext does not favor one candidate over another beyond the information the observer already had. That is the central reason for the secrecy result, not the visual appearance of the output.

The phrase “beyond the information already known” is essential. An observer might know that some messages are much more likely than others. Perfect secrecy does not make all messages equally likely in the observer’s judgment. It says that observing the ciphertext does not change their probabilities. Nor does it hide the message length in this fixed-length model. Our four-bit example concerns four-bit messages. Revealing that a transmission has four bits rather than a different length is outside the particular content-secrecy claim unless the system separately addresses length.

The use of binary combining operations has a history before Shannon’s theory. Gilbert S. Vernam’s US patent 1,310,719, issued on 22 July 1919, describes a secret telegraph signaling system that combines message signals with key signals and recovers them at the receiving side.^[3] A patented combining mechanism and a proof of perfect secrecy answer different questions. The proof needs the fresh, independent, uniformly random secret pad conditions just stated. We should not assume that every repeated tape or every historical device using a related operation satisfied them.

The word “one-time” points to a failure that we can calculate. Let

m_1

and

m_2

be two messages protected with the same pad

k

, producing ciphertexts

c_1

and

c_2

. A person who obtains both ciphertexts can combine them:

(1.3)

The subscripts 1 and 2 simply label the first and second message or ciphertext. The two occurrences of the same pad cancel because a bit XOR itself is 0, and XOR with 0 leaves the other bit unchanged. Consequently, combining the ciphertexts reveals a relationship between the messages that does not contain the pad. The observer has learned something that was absent from the single-use secrecy argument.

For the first message above, the ciphertext was 1101. If a second message is 0010 and the same pad 0110 is reused, its ciphertext is 0100. Combining the ciphertexts gives 1001; combining the two messages gives the same value. That relationship need not reveal both messages in full, but it can be useful when the observer knows something about either of them. In particular, knowledge of one complete message reveals the reused pad for those positions and permits recovery of the other. The failure comes from violating the single-use condition, while the XOR calculations still function correctly.

Perfect secrecy also does not provide message authentication. Someone who changes a ciphertext bit changes the corresponding recovered message bit. The receiver needs additional protection to detect an unauthorized instruction. This anticipates the distinction in Chapter 2: even a very strong confidentiality property does not automatically answer the integrity question. A one-time pad is not a complete secure messaging protocol just because its secrecy argument can be exact.

The practical cost is substantial. Before sending a long message, the parties need a secret pad of the corresponding length, a reliable way to keep their positions synchronized and a way to avoid reuse. Generating the pad

from a short secret with a **deterministic** program may be useful in another design, but it changes the security argument. Deterministic means that fixing all the inputs fixes the result: the same inputs produce the same output. The possible long outputs are then constrained by that shorter secret. Modern stream ciphers pursue computational protection under explicit assumptions rather than obtaining the one-time pad's information-theoretic guarantee merely by producing a long sequence.

1.5 Enigma and changing substitutions

Enigma was an electromechanical cipher machine: its encryption depended on electrical connections and moving mechanical parts. Commercial and military versions existed, with important differences. Its name now stands for a family of machines and operating arrangements rather than one unchanging wartime configuration. For our explanation, consider a military arrangement with a plugboard, three moving rotors, a reflector, a keyboard and lamps. The purpose was still recognizable: turn a readable message into letters that an authorized receiver could recover with compatible equipment and settings. [\[104\]](#)

A **rotor** is a wheel whose wiring connects input contacts to output contacts. Changing its position changes which external letters those connections join. A **plugboard** adds exchanges of letter pairs through cables. When a key is pressed, the electrical path passes through the plugboard and rotors, reaches the reflector, then returns through the rotors and plugboard to light a letter. The rotor positions change as the operator types, so the substitution changes during a message. Repeated input letters need not produce repeated output letters. This is more than the fixed alphabet replacement studied earlier, but its changes still follow a definite mechanism. [\[104\]](#)

The **reflector** pairs its contacts and sends the signal back through the machine. At one fixed configuration, this makes the letter mapping reciprocal: if A maps to Q, Q maps to A. This statement concerns the same rotor configuration, not two successive keystrokes after the wheels have moved. An intended receiver starts with the corresponding settings and follows the same stepping sequence to recover the message. The arrangement also has a restrictive property: a letter cannot encrypt to itself.

That is a property of this machine construction, not a condition that every secure cipher should satisfy.^[104]

Consider what reciprocity means before worrying about the full wiring. In a four-letter teaching alphabet, suppose a fixed configuration exchanges A with C and B with D. Entering A gives C; applying that same fixed mapping to C gives A again. Resetting the configuration matters. If we change the mapping between the two operations, the second need not undo the first. Enigma uses 26 letters and a much richer mechanism; the four-letter example only isolates the idea of applying an exchange twice. It is not a reconstruction of an Enigma rotor or a historical message.

The no-self-encryption property supplies a small but exact test. Suppose an analyst guesses that a ciphertext fragment `ABED` corresponds to plaintext `MEET`. At the third position, the guess requires E to encrypt to E. That alignment is impossible for the Enigma arrangement under discussion, so it can be rejected without recovering the rest of the key. Replacing the ciphertext fragment with `ZKPW` removes this particular contradiction, but does not prove that the guessed word is right. Many wrong alignments can pass one necessary test. The two fragments are original teaching strings, not intercepted wartime traffic.

This distinction between rejecting a possibility and proving an answer is central to cryptanalysis. A **crib** is a proposed piece of plaintext aligned with intercepted ciphertext. It may come from an expected report format, repeated wording or other contextual evidence. The guess becomes useful when the machine's rules imply consequences that can be checked elsewhere. A contradiction eliminates the hypothesis. A surviving hypothesis needs more checks. The basic reasoning does not require that an analyst already know every word of the message, and it does not mean that every likely phrase provides an equally effective attack.

Machine complexity and operational practice must therefore be examined together. A sender needs rules for selecting settings and communicating the information a receiver requires. Predictable wording or a weakness in those procedures can help an analyst organize the search. Changing the wheels' positions does not make the method equivalent to a fresh uniformly random one-time pad for each message. Enigma's substitutions arise from a structured device and a finite selection of settings.

The pad's information-theoretic guarantee relied on different conditions, so its proof cannot be transferred merely because both methods change ciphertext letters.

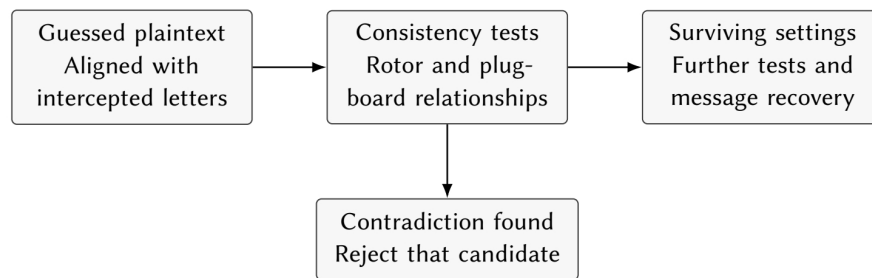
1.6 Alan Turing and the wider codebreaking effort

The history does not begin with Turing alone. At the Polish Cipher Bureau, Marian Rejewski, Jerzy Różycki and Henryk Zygalski developed important attacks before the Second World War. Rejewski's work used mathematical relations between substitutions, together with intelligence material obtained through French cooperation, to reconstruct military Enigma wiring in 1932. The Polish team developed further methods and equipment. In July 1939, Polish specialists shared their results with British and French counterparts. That transfer provided a foundation for the subsequent Allied effort; it was not merely a late confirmation of work already completed elsewhere. [\[105\]](#)

The mathematical idea is accessible even before learning the formal theory. A **permutation** rearranges a finite set without losing or duplicating any element. A rotor's fixed-position wiring can be described this way. Combining several wirings produces another such rearrangement, and observations made under related settings can impose relations between them. This changes the question from guessing an entire secret sentence to identifying a set of transformations consistent with the evidence. It does not make the historical problem easy: the useful relationships depend on the actual device, procedures and information available to the analyst.

At Bletchley Park, Alan Turing developed a crib-based approach embodied in the British **Bombe**, a machine for eliminating inconsistent setting candidates. A crib's linked letter relationships could be arranged into what analysts called a menu. The machine tested consequences of those relationships rather than treating every possible plaintext as an unrelated guess. Gordon Welchman's diagonal board used the reciprocal plugboard relationships to strengthen that elimination. Their work made a central mathematical principle practical: combine constraints so that one impossible consequence can reject a much larger hypothesis. [\[106\]](#)

The contrast with a simple exhaustive search matters. Imagine that a candidate setting requires one plugboard letter to connect to two different partners at the same time. Under a pairwise plugboard arrangement, those requirements cannot both hold. There is no reason to decrypt a complete page under that candidate before discarding it. On the other hand, the absence of this contradiction only makes a candidate worth further examination. This is the same distinction seen in our four-letter crib example, now applied to relationships across a more complicated machine. The explanation is conceptual; it is not the full electrical logic of a Bombe menu.



A likely word is a hypothesis.
Passing one test is not proof of the correct setting.

Figure 1.1. A conceptual crib-based search. A guessed alignment and the machine's rules constrain possible settings. A surviving candidate requires further checking. This original diagram explains the reasoning; it is not a wiring plan or a complete Bombe algorithm.

Theory alone did not produce a working service. Harold Keen and the British Tabulating Machine Company supplied essential engineering, while operators, including members of the Women's Royal Naval Service, configured machines and recorded candidate stops. Further checks and analysis were needed to turn a stop into a useful result. Intercepted traffic and the organization that moved it between specialists also mattered. Describing the outcome as one person building a device that instantly read every message would remove the very cooperation that made the method effective. [\[103\]](#)

Enigma traffic also differed across services and periods. New settings, procedures and machine variants changed the problem; results in one setting did not automatically unlock every other network. Historical accounts describe serious interruptions as well as successes. This helps

explain why a recovered message was an operational achievement, not a timeless proof that every use of the machine had become readable. Our purpose is to understand the interaction between mathematical structure, available evidence and practical conditions, rather than reduce an extended campaign to one decisive calculation. [\[103\]](#)

The Bombe should also be distinguished from Colossus. Colossus was electronic equipment used in the separate attack on Lorenz-encrypted teleprinter traffic, and Tommy Flowers and his colleagues played the central engineering role in its construction. It was not simply Turing's Enigma machine with a different name. The distinction matters because different cryptanalytic problems led to different machines and methods. A story about the birth of computing can connect these efforts without making their devices or contributors interchangeable. [\[107\]](#)

Turing's contribution belongs in this book because it makes a lasting habit of reasoning visible. State what the observer knows, identify the relations the system must obey, and use those relations to separate impossible explanations from candidates that remain. The historical attack was not a break of AES, a solution of every cryptographic problem or a claim that all predictable messages defeat modern encryption. Its lesson is more precise: complicated transformations must be examined through the information and structure an opponent can actually use. The next step in our history gives that question a mathematical language.

1.7 Shannon's contribution

Claude E. Shannon's *Communication Theory of Secrecy Systems* appeared in the *Bell System Technical Journal* in October 1949. It is distinct from his 1948 work on communication theory, although it applies related ideas about information and uncertainty. The opening footnote identifies earlier material in a classified report dated 1 September 1945. The public article therefore records a publication milestone rather than proving that all of its ideas originated in 1949. Its importance for our route is the mathematical treatment of what observing a ciphertext can tell an opponent. [\[2\]](#)

In Section 10, Shannon defines perfect secrecy through probabilities before and after observation. We can write the condition in modern notation as

$$(1.4)$$

Here M represents the random message and C the random ciphertext produced by the experiment. Lowercase m and c denote particular possible values. The expression

$\Pr(M = m | C = c)$ is the probability that the message has value m before the ciphertext is observed. The vertical bar in the left expression means “given that”: it is the probability of that same message value after learning that the ciphertext is c .

. The equality must hold for every relevant message value and every ciphertext value that can occur with positive probability. It formalizes the claim that this observation does not change the message probabilities.

Consider a source that produces a particular possible message three quarters of the time. If the encryption has perfect secrecy for that source, a ciphertext observation leaves that probability at three quarters. It does not reduce it to one half, and it does not prevent an observer from guessing the common message. What the observer lacks is an improvement supplied by the ciphertext. This example is a useful safeguard against reading “perfect” as “the attacker had no knowledge beforehand.” Security statements compare the information or ability available in a specified situation; they do not remove all facts from the outside world.

The four-bit pad gives a concrete way to see the condition. For any fixed four-bit message, each possible four-bit ciphertext occurs with probability

1/16

, because exactly one of the 16 equally likely pads produces it. That distribution is the same for every possible message. Observing which ciphertext occurred therefore supplies no additional preference among the message possibilities. This argument depends on the independence and uniformity of the pad, not on an assumption that the messages themselves were uniformly chosen. It is a small illustration of the relationship between a construction and a security definition.

Shannon also studied the amount of uncertainty associated with messages and keys. **Entropy** is a mathematical measure of average uncertainty for a specified probability distribution. We will define it more carefully in Chapter 5. For now, distinguish a long written representation from a large amount of uncertainty. Writing a predictable value many times does not give it the uncertainty of the same number of independently random bits. The possible choices and their probabilities matter. That distinction explains why a long output from a short secret cannot be credited with fresh independent secrecy merely from its length.

For finite message spaces with exact recovery, Shannon's analysis includes a counting restriction on perfect secrecy: the available key set cannot be smaller than the set of possible messages to be protected. In the four-bit example, there are 16 possible messages and 16 possible pad values. For a particular ciphertext, each possible message must remain a possibility through some suitable key. A key cannot recover two different original messages from that same ciphertext under exact decryption. This gives an intuitive reason why shrinking the key choices arbitrarily cannot preserve the same guarantee. The precise conditions matter; this is not a rule that every useful encryption system must have a key as long as every file it protects.

The paper also separates theoretical uncertainty from the work of solving a ciphertext. A message might be uniquely determined by the available information and still be difficult to find computationally. Conversely, several plausible messages may remain because the intercepted material is short or the source is uncertain. The amount of data needed to narrow the possibilities depends on the cipher and the message model. An estimate

developed for ordinary language cannot be transferred unchanged to compressed files or arbitrary binary records. Mathematical uniqueness and practical recoverability are different questions, and later constructions must state which one their argument addresses.

In its discussion of practical secrecy, Shannon introduces **diffusion** and **confusion**. Diffusion spreads statistical structure across combinations of output symbols, so that simple local counts provide less direct help. Confusion makes the relationship between observable ciphertext statistics and the key more complex.^[2] These ideas help explain why merely renaming letters is insufficient and why more carefully combined operations can be useful. They are design ideas to analyze, not a test that certifies any construction described with those words. We will return to concrete transformations when examining AES.

Shannon did not turn every cryptographic question into a solved problem. His paper established definitions, models and important relationships for a specified class of secrecy systems. Its value is partly that a reader can inspect the conditions of a result and see what it claims. The development from hidden tricks to explicit arguments was not a single event with one ending. Public research continued to investigate computational difficulty, authentication and communication between parties who did not already share the required secret material. The last of these problems became central to the public-key breakthrough.

1.8 The public-key breakthrough

A secret-key system can protect a fast public channel if the parties already share the appropriate secret. That arrangement can be effective, but it leaves an earlier question: how did the parties obtain that secret safely? A trusted courier, an existing protected connection or a managed distribution service may answer it in a particular organization. The problem becomes more demanding when many people want to communicate with parties they have not met. Increasing the speed of the encryption operation does not, by itself, provide a trustworthy first exchange of keys.

One simple model shows why this problem attracted attention. Suppose a group has

n

members and assigns a distinct shared long-term key to every unordered pair. Each member has

$$n - 1$$

possible partners. Multiplying those counts visits each pair twice, once from each end, so the number of pairwise keys is

$$(1.5)$$

The symbol

n

means the number of members, and

$$n - 1$$

means everyone except the member being counted. For four members, the calculation is

$$4 \times 3 / 2 = 6$$

. For 100 members, it is

$$100 \times 99 / 2 = 4,950$$

. This describes the stated pairwise arrangement, not a lower bound for every possible key-management architecture. A trusted distribution service can change the number of long-term keys and the trust relationships. The example shows why creating and maintaining pairwise arrangements can be a major part of the problem.

Whitfield Diffie and Martin E. Hellman's *New Directions in Cryptography*, published in the *IEEE Transactions on Information Theory* in November 1976, places key distribution and digital authentication at the center of the discussion. It distinguishes public-key encryption from establishing a shared secret over a public exchange, and acknowledges related work by Ralph Merkle. [\[4\]](#) The paper is an important public research milestone because it explains new ways to organize these problems. It should not be read as a claim that a public channel, without any further authentication, makes every resulting conversation secure against every active attacker.

In **public-key encryption**, the recipient has a public encryption key and a related private decryption key. A sender can use the public key without receiving the private one. This is different from giving every sender the same secret decryption capability. The construction must make the authorized operations feasible while making the relevant unauthorized task difficult under its assumptions. The key relation is mathematical, but knowing which public key belongs to the intended recipient remains a system question. An attacker who substitutes their own public key can change whom the sender is protecting the message for.

Key agreement takes a different route to a useful result. The parties exchange public values and use private information to calculate shared secret material. A passive observer receives the public transcript, but should not be able to derive the same secret efficiently under the construction's stated assumptions and parameter choices. The shared material can then be used through a defined key-derivation process to support symmetric protection. The important historical step is that the complete long secret need not be delivered in advance by the same confidential route as a one-time pad.

Authentication remains essential when the opponent can actively replace those public values. Without an appropriate check of the intended participants, an attacker may arrange separate exchanges with two correspondents, Nora and Sam, acting as one person to each of them. Both endpoints can finish a mathematically correct exchange while their peer assumptions are wrong. Chapter 10 will make that situation and the relevant arithmetic explicit. Here it explains why the public-key change does not remove the distinction between learning a secret and establishing it with the right party. The new methods change where trust can be established; they do not make trust irrelevant.

Diffie and Hellman also discuss digital signatures: evidence that can be checked using public information without giving each verifier the ability to create the same signing evidence. This addresses a different requirement from confidentiality. Ronald L. Rivest, Adi Shamir and Leonard Adleman's 1978 paper, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, supplies a concrete number-theoretic construction.^[5] Its publication in the *Communications of the ACM* is another milestone in turning the public-key idea into explicit algorithms that researchers could

study. The name RSA comes from Rivest, Shamir and Adleman. RSA is a public-key method; in its encryption use, a sender encrypts with a public key and the intended recipient recovers the message with a related private key. The RSA construction uses arithmetic with large whole numbers and a modulus formed from prime factors. Knowledge of those factors gives a route to generating the matching private operation. The historical contribution does not establish that every possible attack on every RSA-based system is equivalent to factoring, or that the unpadded textbook operations meet modern encryption and signature definitions. We will explain the arithmetic, encodings and actual security conditions in the RSA and signature chapters. For the historical discussion, the important connection is between a public mathematical operation, a private capability and a precise task that an outsider is assumed unable to perform efficiently.

The distinction from perfect secrecy is fundamental. Public-key protection generally relies on limits to feasible computation and on assumptions about the relevant mathematical problems. An attacker can know the public information without having an efficient way to perform the forbidden task. This is not the same statement as saying that the ciphertext leaves all message probabilities unchanged even for an opponent with unlimited computation. Computational protection can be extremely useful without being the one-time pad's information-theoretic guarantee. Its analysis must still address the actual construction, parameters and permitted attacks.

The public chronology also has a classified counterpart. GCHQ's historical account credits James Ellis, Clifford Cocks and Malcolm Williamson with related work that remained secret before its disclosure in 1997.^[6] That account is a reason to distinguish public publication from earlier internal discovery. It does not reduce the importance of the independent public research that made the ideas available for examination and wider development. We use the dates in Figure [1.2](#) to identify specific public documents in this chapter, rather than to assign a single first moment to every underlying idea.

1883	•	Kerckhoffs: military cryptography Separate the general method from the secret key.
1919	•	Vernam: secret signaling patent Combine and recover telegraph signals using key signals.
1949	•	Shannon: secrecy as an information question Define perfect secrecy and study its conditions.
1976	•	Diffie and Hellman: public-key research Examine key distribution and digital authentication.
1978	•	Rivest, Shamir and Adleman: RSA Present a concrete public-key construction.

Figure 1.2. Selected public documents in the chapter's argument. The dates identify publication or patent milestones, not a complete history or a claim that each idea originated in that year. The 1919 patent describes a signaling mechanism; it is not itself the proof of perfect secrecy.

Public-key methods and secret-key methods now provide complementary roles in many systems. A public-key operation can help establish or protect key material; symmetric constructions can then protect the data using appropriate derived keys and authentication. This arrangement does not follow automatically from placing two algorithms side by side. The protocol must bind the exchange to its context, use compatible inputs and interpret the verification results correctly. Later chapters will show the steps in detail. The historical point is that a new way to establish a private capability can enable communication at a scale that a prearranged pad alone would make difficult.

We can now describe what changed in the development of the subject. A cipher is examined through its structure and the attacker's knowledge, rather than its unfamiliar appearance. Key secrecy is separated from knowledge of the method. A definition can express exactly what a ciphertext observation reveals, while a computational argument asks how difficult a forbidden task is. Public-key techniques introduce new ways to distribute capabilities and verify evidence. The next chapter turns these historical lessons into explicit protection goals: what must remain secret, which changes should be detected and why a key belongs to an intended person. Chapter [3](#) then begins our mathematical foundations with bits,

bytes, XOR and remainders, using the small examples from this chapter as a familiar starting point.

Questions to check your understanding

1. A substitution cipher has many possible replacement alphabets. Why does counting those alphabets not establish that an attacker must test them all?
2. What should remain true if an attacker learns the general method in a system designed according to Kerckhoffs's principle? Does publishing a method prove that it is secure?
3. Recover the message from ciphertext `1101` and pad `0110`. Which conditions on the pad are needed for the one-time pad's secrecy argument?
4. An observer believes a message has probability $\frac{3}{4}$ before seeing its ciphertext. Under perfect secrecy, what is the probability afterward? Why is it not automatically $\frac{1}{2}$?
5. What relationship is exposed when two ciphertexts use the same XOR pad? Does that always reveal both complete messages immediately?
6. Why can a correct public key-agreement calculation still leave Nora and Sam communicating with an active intermediary?
7. Why does the alignment of `MEET` with `ABED` fail the Enigma no-self-encryption test? Would passing that test prove that a crib is correct?
8. Distinguish the contributions of the Polish mathematicians, Turing, Welchman, engineers and operators. Why is a Bombe stop not the same as a recovered plaintext?
9. Why should an account of Enigma distinguish the Bombe from Colossus?

2. What Cryptography Protects

Part I: Secrets and Trust

Imagine that Nora is organizing a science activity at a school. She sends Sam a message asking him to order 20 sensors for the laboratory. Sam must read the instruction, decide whether it came from Nora and act on the quantity she intended. The message is short, but several things can go wrong. Someone could read information that was meant to remain private, change the quantity to 200, send a false instruction under Nora's name or repeat an old order. Each event raises a different protection question. Treating them all as a single problem called "security" makes it harder to understand which method can help.

Cryptography studies methods that help keep information secret, detect unauthorized changes and check evidence about a message or a participant. A **cryptographic algorithm** is a set of exact steps for a mathematical task. A **key** is data used by those steps to select a particular operation; it is not necessarily a password that a person can remember. For example, two devices can hold the same secret data value and use it to protect their messages. The method describes what both devices do, while the key supplies the particular secret they share. Our first question is what protection this arrangement should provide. We can then ask which algorithms provide it and what happens if the key is copied, the message is changed or the receiver is deceived.^[7] This chapter builds a vocabulary for asking those questions. We will follow Nora's instruction through observation, modification and checks of identity. We will also examine what happens before a message is protected and after it is recovered. No previous knowledge of encryption or number theory is needed. By the end, you should be able to separate a protection goal from a mechanism, describe a simple attacker and explain why a valid cryptographic check does not settle every question about a message.

Several names will help us keep these levels separate. A **primitive** is a basic cryptographic tool with a narrow job, such as transforming one block of data. A **scheme** or **construction** combines defined steps to provide a

specified service, such as encrypting and checking a whole message. A **protocol** gives the rules that communicating parties follow: what they send, what they check and what they do next. An encryption tool can be correct while a protocol uses it badly. In the laboratory example, protecting the message is one task; deciding whether Sam should place the order is another.

2.1 A message on an untrusted network

When Nora presses Send, the application represents her instruction as digital data. Those data may pass through a wireless access point, several networks and one or more services before Sam's application displays them. The exact route depends on the application and connection. Nora and Sam cannot usually inspect every machine on that route, and they do not personally know every operator involved. An **untrusted network** is a useful starting assumption: we design the protection without requiring all the systems carrying the data to behave honestly. The word "untrusted" describes what our security argument is allowed to depend on. It does not mean that every network operator is an attacker.

An intermediate system may need to see enough information to forward a packet, which is a small unit of transmitted data. It does not follow that the system needs to understand the instruction inside the application. Keeping those roles separate is one reason cryptography is useful. A network can carry protected data while the endpoints, the systems where a particular protection begins and ends, perform the cryptographic operations. We must still identify those endpoints. If protection ends at a service that reads the message and sends it onward, that service is inside the boundary of the confidentiality claim for that step.

Suppose Nora's application sends the instruction as readable text with no protection. A person able to observe the relevant traffic may learn the quantity and purpose of the order. If that person can also replace the data on their way to Sam, the same message might arrive with a different quantity. Hiding the text would address the first concern, but we should not assume that it also addresses the second. Some encryption methods allow an attacker to change protected data in ways that affect the recovered message. Later we will examine concrete examples. At this stage, the lesson is that

reading and changing are different capabilities, so they need different security questions.

We also need to decide which information is important. A laboratory order might be public, but its quantity must still be reliable. A medical message may require both privacy and evidence that it has not been changed. A software publisher might want everyone to read a new program while preventing an attacker from presenting a modified version as an official release. The same mathematical tools can appear in these settings, but the required services are different. The starting point is therefore the information and the decision that someone will make from it, rather than the name of a fashionable algorithm.

Protection can also concern stored data. Nora may save the order on a laptop, keep a backup or send it to a shared archive. In each case we ask who can obtain the stored representation and who can use the keys needed to recover it. Encrypting a file can reduce what an unauthorized reader learns from a copied storage device. It does not establish that the computer displaying the file is safe, or that the file is the latest version. A system can need protection during storage and during transmission, with different keys and boundaries for each. NIST's key-management guidance treats both communication and stored information as important uses of cryptography.^[8]

Before choosing a method for Nora and Sam, we can describe the intended result in ordinary language. Sam should recover the intended instruction, and a person outside the authorized group should not learn its protected contents from the transmitted data. An attacker should not be able to make Sam accept a changed instruction as an unchanged one. Sam also needs a basis for connecting the instruction to the right source. Finally, the application must decide whether it has already processed this order. These requirements are related, but none should disappear inside an unexplained claim that the connection is encrypted.

2.2 Keeping a secret and detecting a change

Confidentiality concerns disclosure: information should not become available to an unauthorized party. Encryption is one mechanism for pursuing that goal. The readable input to an encryption operation is called the **plaintext**; its protected output is the **ciphertext**. Decryption uses the

appropriate key and other required inputs to recover the plaintext. The contents may be text, an image, a program or a sequence of measurements. “Plaintext” does not mean that the input must look like a sentence. It means that these are the data being protected at that layer.^[8]

In a secret-key example, the sender and receiver use the same secret key for the corresponding encryption and decryption operations. Their software knows the procedure, and they must already have a suitable way to obtain and protect that key. The procedure itself does not need to be hidden from the attacker. The historical opening introduced this distinction through Kerckhoffs’s principle. For now, distinguish the public description of a method from the secret values on which its use depends. A system whose security requires an outsider never to discover its procedure is making a different and often much harder assumption than one that requires a key to remain secret.

We can express a basic requirement with a short formula. For a correctly used secret-key encryption scheme with exact decryption correctness, valid encryption followed by the matching decryption recovers the original message:

(2.1)

Here
 m

is the message and

k

is the secret key. The name

Enc

means “apply the encryption operation,” and

Dec

means “apply the decryption operation.” Read the inner operation first:
 encrypt

m

using

k

. Then apply decryption, with the matching key, to the resulting ciphertext. The equality says that the recovered value is

m

. This notation leaves fresh random values and other scheme-specific inputs implicit; the later construction chapters will make those inputs explicit. It describes the intended relationship between the operations, not a complete software interface.

Equation [2.1](#) is a **correctness condition**. It does not by itself establish confidentiality. A procedure that returns its input unchanged could satisfy an equivalent recovery condition while revealing the whole message. Even a reversible transformation that changes every visible character might be easy to undo without a secret. A security argument must go further: it describes what the attacker can observe and do, and why this does not give a useful way to recover protected information or distinguish the protected alternatives covered by the definition. This distinction between functioning as intended and resisting an opponent will return throughout the book.

Integrity concerns unauthorized change. For Nora’s order, Sam needs a way to detect that an attacker has changed the quantity or another protected field. The word “detect” matters. Cryptographic protection does not make it physically impossible to overwrite a packet or damage a file. It can make altered data fail a check, so that the receiver rejects them instead of treating them as an accepted message. The application’s behavior after that check is part of the system. Showing an error while still using the unverified instruction would not achieve the intended result. [\[8\]](#)

A **checksum** is a value calculated from data to help detect accidental damage. A **cryptographic hash** also calculates a value from data using a public rule, with additional security goals that we will examine in Chapter [8](#). Its output is called a **digest**. If an attacker can replace both a file and an unprotected checksum or digest, they can generally calculate a matching value for their replacement. Sam therefore needs a trustworthy reason to expect one particular digest. A mismatch can warn him that two copies differ; a matching value received from an unknown source does not identify that source. Later we will distinguish this replacement problem from a hash collision, in which two different inputs have the same digest.

One way to produce evidence of integrity is a **message authentication code**, or MAC. In a suitable construction, a secret key and the protected data determine a tag that a receiver can check. Someone without the required key should have no practical method, within the stated limits of the scheme, for making a new accepted message and tag. The tag does not need to hide the message. It addresses a different goal. Conversely, encryption that hides the message should not be assumed to supply an integrity tag. These distinctions prevent us from treating every cryptographic output as if it supplied the same protection.

Authenticated encryption combines confidentiality with an integrity property in one defined construction. Its rules specify the required inputs and the conditions for accepting or rejecting a ciphertext. In a correctly used scheme, an unauthorized modification should fail verification except with the small probability allowed by the security bounds. That is more precise than saying that a changed message can never pass a check. Real guarantees have assumptions and limits, including rules about keys and other inputs. The authenticated-encryption chapter will explain them without requiring readers to invent their own combination of encryption and authentication. [\[7\]](#)

These mechanisms can also protect some information without hiding it. For example, an application might leave an order number visible for routing while requiring the receiver to check that it has not been changed. Authenticated-encryption schemes can support such **associated data**: inputs covered by authentication but not encrypted by that operation. The application must supply the intended values during verification. Leaving a field outside the ciphertext does not automatically put it inside the integrity guarantee. We must ask which exact bytes the construction protects and how their meaning is determined.

Table 2.1 keeps the main questions separate. It is a guide to choosing what evidence is needed, rather than a claim that every application must expose a separate operation for every row. A well-designed protocol can combine several protections. We still need to identify them individually when explaining why the complete system works.

Table 2.1. Different questions about the same instruction.

Goal	Question for Sam	What needs separate attention
Confidentiality	Can an unauthorized observer learn the protected contents?	Visible metadata, endpoints and key access.
Integrity	Can an unauthorized change pass as an accepted message?	Which fields are protected and how rejection is handled.
Source authentication	What connects the accepted message to its source?	The identity and authorized use of the relevant key.
Freshness	Is this instruction valid for this occasion?	Replay, sequence, time or request state.

Keeping these questions separate also avoids a common misunderstanding about the word “authenticated.” An authenticated-encryption check gives evidence relative to a key and a defined set of inputs. It does not identify the human holding the device through mathematics alone. To connect that key to Nora, Sam needs an additional basis for trusting the key assignment and the way it was used. That is the next part of our example.

2.3 Knowing who holds a key

Suppose Nora and Sam alone share a secret authentication key. When Sam receives an instruction whose tag verifies, this gives evidence that the protected data were produced by someone with access to that key, under the scheme’s assumptions. In the intended arrangement, that group includes Nora and Sam. Sam may have a good reason to attribute an incoming instruction to Nora because he knows that he did not create it and trusts how the key was protected. However, a third person cannot look at the same shared-key tag and distinguish Nora from Sam as its creator. Both have the

ability to produce a valid tag. The cryptographic relation and the surrounding facts must be read together.^[8]

The situation becomes weaker if the same key is shared by a whole department. A valid tag may then show membership of the key-holding group without distinguishing its individual members. Even a key assigned to one person can be used by software acting for that person, by an authorized service or by someone who has stolen it. For this reason, statements about identity should name the evidence they rely on. “This check succeeded with the key assigned to Nora’s device” is narrower than “Nora personally read and approved every word.” The second statement needs evidence about the device, its user and the approval process.

Public-key cryptography introduces a different arrangement. A participant has a public key that others may know and a corresponding private key that must be protected. The operations depend on the construction. Public-key encryption and digital signatures solve different problems, so a reader should not infer their behavior from the words “public” and “private” alone. Public-key encryption can let someone protect a message for the holder of a private decryption key. A digital signature lets the holder of a private signing key produce evidence that others check using the corresponding public verification key. Later chapters will explain the mathematics and the precise security goals behind both arrangements.

A digital signature does not need to conceal the signed data. A software publisher can sign a program that everyone is free to download and inspect. Verification can connect the exact signed data to the relevant signing key and detect unauthorized changes, under the signature scheme’s assumptions. It does not establish that the program has no defects. Nor should a signature be described as encryption with a private key. Such a description confuses the direction of a particular arithmetic operation with the purpose, encoding and security definition of a complete signature scheme. We will use the separate names for the separate tasks.

The public key itself raises a new question. If an attacker gives Sam a key and labels it “Nora,” successful verification with that key does not correct the false label. Sam needs a basis for connecting the verification key to the source he intended to trust. He might already possess an independently confirmed key, use an established organizational directory or

rely on an appropriate certificate system. Each method moves part of the trust question somewhere specific. It does not remove that question. A chain of mathematically valid statements can still begin from the wrong assumption about whose key it is.

A **certificate** is one common way to associate a public key with an identified subject, such as a service name, within a defined trust system. In a certificate-based internet connection, checking a valid chain is part of the work. The application must also check that the certificate identifies the service it intended to reach. A valid certificate for one name is not evidence that it belongs to a different name. The rules for service-identity checking are therefore distinct from certification-path validation. We will develop this distinction in the signature and TLS chapters rather than treating a certificate as a universal badge of trust.^[10]

This matters in a familiar situation: a deceptive website may use an encrypted connection to its own correctly certified domain. The connection can protect the bytes between the visitor and that site while the site's message remains dishonest. There is no contradiction. The cryptographic check is answering a question about a named endpoint and the channel, not deciding whether its sales claims, advice or requests are truthful. The same principle applies to a signed statement. A legitimate key holder can sign an inaccurate measurement, and a correctly verified signature does not make that measurement true.

Authentication and **authorization** must also be distinguished. Authentication supplies evidence about an entity, source or key relationship. Authorization concerns whether the requested action is allowed. Sam may receive an authentic instruction from a member of staff who has no authority to order equipment. An application needs a policy for such a decision, even if cryptographic credentials help enforce it. Similarly, permission to read a record need not include permission to change it or share it. The security design must connect a verified identity or role to the correct permissions instead of assuming that being recognized permits every action.^[8]

For Nora's order, the complete identity claim is now more concrete. Sam must know which key or credential is relevant, why it is associated with the intended source and what that source is permitted to request. He must also

decide how to respond if the key may have been exposed or replaced. These questions will lead to key distribution, certificates and lifecycle management later in the book. For now, they explain why protecting a secret key is necessary and why a successful verification result has to be interpreted inside a system.

2.4 What an attacker can see and do

A security statement needs an attacker model: a description of the opponent's access, actions and resources. Without that description, "the message is secure" is incomplete. An attacker able only to observe traffic is different from one able to replace it, request operations from a server or read the memory of Nora's laptop. A method may meet its goal under one set of conditions and fail under another. Naming those conditions does not weaken a scientific explanation. It makes the claim specific enough to examine. A **passive attacker** observes without changing the relevant communication. The attacker may collect many protected messages and may know information about their likely contents. For example, an order form may have a public structure, or the attacker may know that a particular field contains a small set of possible values. A useful confidentiality argument cannot simply assume that every message is a completely unpredictable secret. Later definitions will make precise what information the attacker receives and which differences between messages are meant to remain hidden. The point here is to avoid giving an encryption scheme credit for protection that actually came from keeping the test situation unrealistically obscure.

An **active attacker** can interfere with the communication in the ways allowed by the model. They might alter a quantity, insert a new message, delay delivery, reorder packets or pretend to be one participant. Different protocols provide different checks against these actions. Figure [2.1](#) follows the same instruction through three simple cases. The picture does not claim that an attacker must occupy a literal physical position between Nora and Sam. An equivalent capability might come from a compromised service, a wireless access point or another part of the route.

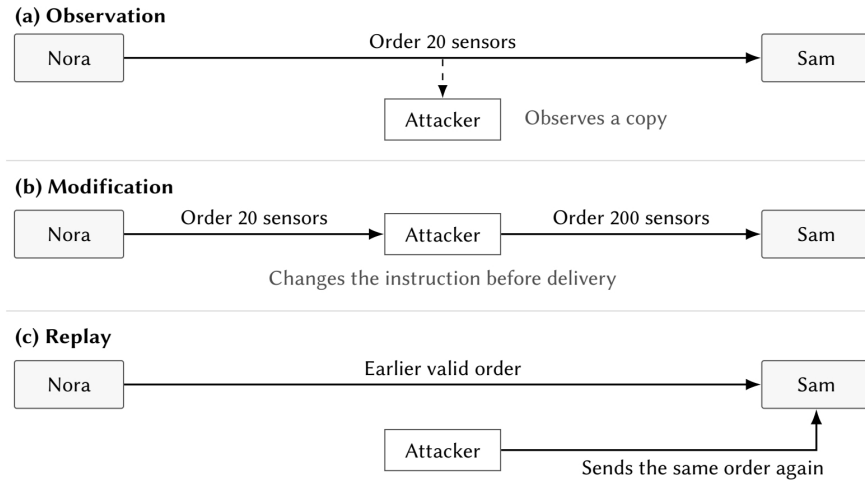


Figure 2.1. One instruction, three attacker capabilities. In (a), the attacker observes a copy. In (b), the attacker changes the quantity before delivery. In (c), an earlier instruction is sent again. The arrows describe capabilities to consider; they do not imply that the application has no protection.

The third case is a **replay**: an attacker sends previously valid data again. Imagine that Nora's first order was authentic, its tag was valid and Sam processed it correctly. Sending those same protected bytes a second time may still produce a valid cryptographic check, because the attacker did not change them. Yet processing the order twice would be a mistake. The system needs a separate reason to accept an instruction only in the appropriate context. An order identifier recorded as already processed is one possible application-level control. Other protocols use sequence information, challenges or carefully managed time information. Their details and failure conditions matter, so a timestamp or an arbitrary extra number should not be assumed to solve replay on its own.

Freshness is the requirement that the accepted data or interaction are suitable for the current occasion. It is related to integrity but is not the same question. An unchanged instruction can be old, a valid credential can be used in the wrong interaction and a correctly signed file can be an earlier release. Where context values are used, they must be protected and checked in a way that ties them to the intended action. For Nora's application, a unique order number is useful only if Sam's system checks it, records the relevant state and handles retries consistently. Merely displaying the number beside the message does not establish that the order is new.

The model must also say which secrets the attacker can obtain. A network attacker who cannot read an endpoint's memory may be stopped by a sound channel protocol. The same protocol does not promise that malware controlling the receiver cannot read the plaintext after decryption. Some systems are designed to limit the effect of a later key exposure, and some can recover specified protections after a temporary compromise. Such properties require additional constructions and conditions. They must not be assumed simply because the application changes keys from time to time. We will return to those questions when discussing key management and secure messaging.

An attacker may learn from more than the final message. A server's error, the time taken to respond or the pattern of accepted requests can become part of the observation. In later chapters we will study cases where these details expose information that the main encryption operation was meant to hide. For the present chapter, it is enough to notice that a system's observable behavior is wider than its advertised input and output. A threat model that includes only the happy path may leave out the behavior on which a practical attack depends.

Finally, a claim should describe what counts as success. Recovering a whole secret key is only one possible objective. An attacker might succeed by learning one sensitive fact, causing one false message to be accepted or impersonating a peer in one interaction. The relevant resources and acceptable probabilities depend on the security definition and use case. A useful explanation therefore names the protected information, the forbidden result and the conditions under which the attacker is considered. It does not demand that every attack look like the complete recovery of a secret in a film.

2.5 What cryptography cannot protect by itself

Cryptography can protect a representation of data while the data pass through a defined boundary. It cannot, by itself, establish that the information was true before protection began. Suppose Nora accidentally writes 200 instead of 20 and her application authenticates that instruction correctly. Sam may receive strong evidence that the protected bytes were unchanged, but those bytes still contain the wrong quantity. Detecting the

mistake needs another process, such as reviewing the order, enforcing a sensible limit or comparing it with a separate record. Integrity protection preserves the relationship to the authenticated data; it does not supply an independent measurement of the outside world.

The endpoints remain important for the same reason. Nora's application must process the instruction before encrypting it, and Sam's application must process the recovered instruction afterward. A compromised endpoint may expose the information at either stage or cause a different instruction to be protected. Protecting the network path still has value, but its guarantee must be stated for that path. **End-to-end encryption** identifies an arrangement in which the relevant message protection extends between the designated communicating endpoints rather than ending at an intermediate service. It does not establish that those endpoint devices are free from compromise or that their users will keep the recovered message private.

Keys also have a lifecycle. They are generated, distributed or established, stored, used and eventually replaced or destroyed. If a key is copied by an attacker, a strong primitive may still perform its calculation perfectly while the attacker uses the copied value. If the only decryption key is lost, an honest owner may lose access to protected information. Availability of authorized key material and prevention of unauthorized access can pull in different directions, which is why backup and recovery require design rather than an informal promise. NIST's guidance emphasizes that poor key management can compromise otherwise strong algorithms. [\[8\]](#)

Information about a message is often called **metadata**. Examples include its length, the time it was sent and the addresses used to deliver it. These facts are different from the message's contents. An observer who cannot read an order might still notice when the laboratory sends orders or how much data it sends. A claim about content secrecy must therefore be read separately from a claim about hiding communication patterns.

Confidentiality of content is also narrower than complete privacy. A protected connection can still expose information through timing, communication volume or data length. Which addresses and names are visible depends on the protocol stack and deployment. Even when a layer hides one kind of metadata, other observations may remain. The specification for Transport Layer Security (TLS), a protocol used to protect

many internet connections, for example, distinguishes its content-protection goals from its limitations around transmitted lengths and traffic analysis. An explanation of an encrypted connection should therefore state what it protects instead of claiming that an observer learns nothing.^[9]

Cryptography does not guarantee that a message will be delivered. An attacker who can block the communication may prevent Sam from receiving the order, even if changing an accepted order is infeasible.

Availability concerns whether a service or information can be obtained when needed. Systems address it through measures that can include redundancy, resource controls and recovery procedures. Cryptographic authentication may help a service reject some unwanted requests, but it cannot force a failed network link to carry traffic or create unlimited processing capacity. Rejection of a forged message is a useful security result even when the receiver still faces a denial of service.

The protected boundary must also match the application's purpose. A file may be encrypted on a storage device and then copied as readable data into an export. A message may be secure between two services while one of them writes its contents to an accessible log. A signed object may be checked correctly and then interpreted under the wrong units or rules. These are reasons to trace how the data are used, not reasons to abandon the underlying protections. The useful question is where a guarantee begins, where it ends and which additional decisions must preserve its meaning.

Testing helps us examine some of those decisions, but successful tests are not a complete security argument. A round-trip test may confirm that one encryption and decryption example recovers the input. Negative tests can check that specified changes are rejected. Neither procedure, by itself, explores every attacker strategy or proves the assumptions of the construction. Mathematical analysis and implementation checks complement each other when their scopes are stated. Later chapters will show how public test vectors, adversarial cases and verified components contribute different kinds of evidence. None should be described as a universal certificate that an application has no vulnerabilities.^[7]

Return now to the laboratory order. A good explanation should tell Sam which contents are confidential, how a change is detected, why a key is connected to Nora and how an old order is recognized. It should also

identify the devices and services that can read the instruction and the policy that permits an order to be placed. These questions form a small specification of the intended protection. Only after stating them can we sensibly choose and analyze the mechanisms. The next chapter begins the mathematical tools we need to follow those mechanisms, starting with bits, bytes, XOR and remainders. We will connect each operation to an explicit task instead of assuming that an unfamiliar symbol explains itself.

Questions to check your understanding

1. An order is public, but changing its quantity must be detected. Which protection goal is essential? Is confidentiality also necessary for that stated requirement?
2. A stranger gives Sam a public key labeled “Nora.” A signature verifies with that key. What has Sam still not established?
3. A correctly authenticated order is recorded and processed. The same protected order arrives again. Why can another successful authentication check be insufficient?
4. A receiver recovers every test message after decryption. Does this establish that an observer cannot learn the messages? Explain the missing question.
5. A signed measurement contains a wrong value because its sensor was faulty. Which part of the claim can the signature support, and which part needs other evidence?

II. The Mathematics We Need

3. Bits, Bytes and Remainders

Part II: The Mathematics We Need

A cryptographic operation does not receive a word in quite the same way that a reader sees it on a page. It receives a representation of data: a sequence of bits or bytes, interpreted according to specified rules. Before an application can protect Nora's instruction, it must decide how the text becomes bytes, where one field ends and another begins, and how numbers are written. These decisions do not replace the cryptographic construction, but they determine its actual inputs. Two people who believe they are protecting the same message can obtain different results if they apply different representation rules.

The mathematics needed to follow these steps is accessible. We begin with counting and place value, then examine a few operations on small binary patterns. The XOR operation from Chapter [1](#) will become a precise rule that can be applied to whole bytes. We will distinguish it from ordinary addition and from putting two sequences next to each other. We will also explain how calculations can wrap around a fixed range and what a remainder means when it appears in a cryptographic formula.

The chapter ends with a short optional program that repeats operations already explained on paper. Programming is another way to inspect the calculation, rather than a prerequisite for the argument. Our examples specify their encodings, widths and expected results. None of the teaching masks or small values is presented as a complete encryption system. The aim is to make later algorithm descriptions readable without leaving the meaning of their inputs or symbols to guesswork.

3.1 From text to bytes

A **bit** is a binary digit, with possible values 0 and 1. A **byte**, in the conventions used throughout this book, is a sequence of eight bits. An **octet** is another name for an eight-bit unit, often used in internet specifications. A byte can be part of a text encoding, a number, an image, a key or a

ciphertext. Its role comes from the surrounding rules. The same bit pattern does not carry a built-in label saying that it is a letter or a secret number.

Text needs an **encoding**: a defined mapping from a character representation to units that software can store and transmit. In US-ASCII, the capital letter A has the numeric value 65. UTF-8 preserves that representation for the ASCII range, so the UTF-8 encoding of A is a single byte with value 65. We will shortly write that same byte as binary `01000001` and hexadecimal `41`. These are different notations for the same eight-bit pattern, not three different messages and not three encryption methods.^[11] The phrase “one character” requires care. Unicode is a common system for identifying text characters and related elements. It assigns numbers, called **code points**, to characters and related elements used in representing text. UTF-8 encodes an allowed code point using one to four bytes. A visible letter with an accent may also have more than one valid underlying character sequence. Therefore, counting visible symbols, counting code points and counting encoded bytes are different tasks. For cryptographic input, the byte sequence and its length are often the quantities that the operation actually receives.

For example, write the word *Café* using a single precomposed code point for its final accented letter. In UTF-8, its bytes, written in hexadecimal, are

The C, a and f each use one byte, while the precomposed é uses two, giving five bytes in total. The spaces in this printed display separate byte values for readability; they are not extra space characters in the encoded word. We do not need the full UTF-8 construction to use this example. We need to recognize that four visible letters have not become four bytes.

The same word can be represented with an ordinary e followed by a combining acute accent. That sequence has the UTF-8 bytes

It contains six bytes, despite normally displaying the same word. Unicode defines canonical equivalence and normalization forms to handle specified relationships between such sequences.^[12] An operation that

compares or authenticates exact bytes does not automatically apply those text rules. If a protocol requires a normalized representation, the participants must agree on that requirement and its place in the processing steps. Arbitrarily changing text while checking an already authenticated object can instead change the very data being checked.

This is a representation issue before it is a cryptographic issue. A correct signature or message-authentication operation cannot make two different byte sequences identical merely because they look alike on a screen. Likewise, successful decoding does not establish that a message is authentic, and not every arbitrary byte sequence is valid UTF-8. A ciphertext may contain values that have no useful interpretation as text. Trying to display it as ordinary characters can lose information or cause a decoding error. Software therefore needs a way to carry arbitrary bytes without pretending that every byte sequence is readable language.

An **encoding** and **encryption** answer different questions. Anyone who knows a public encoding rule can normally reverse the representation when the input is valid. Seeing text written as a series of hexadecimal digits does not mean that its contents are secret. Encryption adds a construction with keys and a defined security goal. Encodings remain useful around it: an application might encode a ciphertext for transport and decode that representation before applying the cryptographic operation. The order and meaning of those steps need to be explicit.

The same need for agreement applies to structured messages. If an order contains a name, a quantity and an identifier, the receiver needs rules for separating those fields and interpreting their values. Joining the text fields AB and C gives the same undivided text as joining A and BC . The application cannot recover the intended boundary from that joined representation alone. Length fields, fixed-size fields or an agreed structured format can supply the missing information. We will return to protocol context later; the foundation is that cryptography operates on the specified representation, not on an unstated interpretation in the sender's mind.

3.2 Binary and hexadecimal numbers

An **integer** is a whole number, which may be negative, zero or positive: for example,

−2

,

0

or

65

. It has no fractional part. **Unsigned** means that the representation we are discussing contains no negative values. In an **unsigned eight-bit integer**, the eight positions describe one of the numbers 0 through 255. The same byte can instead be part of a text encoding. We must say which interpretation we are using before doing arithmetic with it.

In ordinary decimal notation, position gives a digit its weight. The number 253 means two hundreds, five tens and three ones. Moving one position to the left multiplies the position's weight by ten. Binary uses the same idea with a base of two: its positions have weights 1, 2, 4, 8, 16 and so on. Only the digits 0 and 1 are needed because each position states whether its weight contributes to the total. A written sequence of bits becomes an unsigned number when this interpretation has been specified.

For an eight-bit value written with the highest-weight position on the left, the weights are 128, 64, 32, 16, 8, 4, 2 and 1. Consider 01000001. The positions with a 1 have weights 64 and 1. Its unsigned value is therefore

(3.1)

The subscript 2 identifies the base of the numeral inside the parentheses; the unmarked number on the right is decimal. The leading zero preserves the eight-bit width in this display. It does not add to the unsigned value, but it helps us see that the data occupy a complete byte. Figure [3.1](#) separates the bit positions, their weights and the two four-bit groups used for hexadecimal notation.

A in UTF-8: one byte

Position	7	6	5	4	3	2	1	0
Bit	0	1	0	0	0	0	0	1
Weight	128	64	32	16	8	4	2	1

0100 gives hex 4
0001 gives hex 1

Hexadecimal: 41
 Unsigned decimal: 64 + 1 = 65

Figure 3.1. The UTF-8 encoding of A is one byte. The same byte can be read as eight bits, the hexadecimal pair 41, or the unsigned decimal value 65. The weights explain the numeric reading; they do not encrypt the character.

For a second example, the 1 bits of 10100111 occupy positions with weights 128, 32, 4, 2 and 1. Adding those weights gives

This is a conversion of notation, not a transformation of the data's meaning. The byte could be part of a larger number or a cryptographic input. Its numeric value alone does not establish that it is a valid standalone text character. To read it as text, we would need an encoding and a valid sequence in that encoding.

An eight-bit sequence has two choices at each of eight positions. Multiplying those choices gives

$$2^8 = 256$$

possible patterns. The superscript 8 means that eight factors of 2 are multiplied together. If the patterns represent unsigned integers starting at zero, their values range from 0 to 255. More generally,

$$n$$

independent bit positions have

$$2^n$$

possible patterns, where

$$n$$

is the number of positions. This counts possibilities; it does not assert that a particular method selects them uniformly or that an

$$n$$

-bit object automatically supplies

$$n$$

bits of security.

Hexadecimal, or hex, is base sixteen. Its digits 0 through 9 have their familiar values, while A, B, C, D, E and F represent 10, 11, 12, 13, 14 and 15. A four-bit group has exactly sixteen possible patterns, so one hex digit can represent one such group. Two hex digits represent one complete byte. In `10100111`, split the bits as `1010` and `0111`. Their values are 10 and 7, giving the hex pair `A7`. FIPS 197 uses this correspondence when describing data for the Advanced Encryption Standard (AES), the encryption algorithm studied in Chapter [6](#).^[13]

The same result can be read directly using base-sixteen place value:

(3.3)

The left digit has weight sixteen, and the right digit has weight one. A contributes the value ten, not the text-encoding value of the letter A. This distinction is easy to miss: A used as a hexadecimal digit represents 10, while the text character A has UTF-8 byte value 65. The role of a symbol depends on the notation and operation being used.

Programs often write a prefix such as `0x` to mark a hexadecimal integer. Thus `0xA7` and the decimal integer 167 can denote the same integer value. A prefix such as `0b` marks binary notation in the Python examples later in the chapter. The prefix is part of the written notation; it does not add two bytes to the represented integer. Uppercase and lowercase hex letters are commonly accepted as equivalent by a hex decoder, but the literal text strings `A7` and `a7` are different sequences of text bytes. An application must distinguish decoding a representation from authenticating that representation as text.

For a byte sequence, writing two hex digits for every byte preserves its displayed boundaries. The bytes `00 01` form a two-byte sequence. If they are interpreted as a big-endian unsigned integer, their numeric value is 1, but that does not make them the same sequence as the single byte `01`. The leading zero is part of the data in the sequence even though it does not change that numeric interpretation. RFC 4648 describes base-sixteen encoding as representing each input octet with two output characters.^[14]

The text `41` provides another useful check of our vocabulary. Interpreted as a hexadecimal byte representation, it denotes the one byte with decimal value 65. Encoded literally as ASCII or UTF-8 text, the two characters `4` and `1` produce two bytes: `34 31` in hexadecimal. Those inputs have different lengths and different values. When a test vector gives a key or message in hex, the reader must determine whether the program should decode that display into bytes. Giving the cryptographic operation the printed hex characters instead is a different experiment.

3.3 XOR and the reversible mask

Chapter [1](#) used XOR to explain a one-time pad. We now examine the operation as a general way to combine equal-width bit patterns. At one position, XOR outputs 1 exactly when the two input bits differ. It outputs 0 when they match. For whole sequences, it applies that rule to each corresponding pair of bits without carrying between positions. We will use

$$\oplus$$

for this bitwise operation. The width and position alignment of the inputs are part of the statement, not details to be chosen afterward.

Take a byte with hex value `5A` and combine it with a byte mask `0F`. Their binary forms make the calculation visible:

The four leading mask bits are zero, so those input bits remain unchanged. The last four mask bits are one, so the corresponding input bits flip. The result is hex `55`, or decimal 85. A **mask** here is simply a bit pattern used in a defined operation. Calling it a mask does not assert that it is random, secret or suitable as an encryption key.

For equal-length sequences

a

and

b

, the basic identities include

(3.4)

In this equation, 0 means the all-zero sequence of the same length as the other input. The first identity says that matching each bit with zero leaves it unchanged. The second says that a bit compared with itself never differs, so every output bit is zero. In the third, the positions where

b

has a 1 are flipped twice and therefore restored. These are statements about the operation for all such sequences, rather than results that depend on one lucky example.

The order of the two inputs does not change XOR's result:

$$a \oplus b = b \oplus a$$

. The grouping also does not change a sequence of XOR operations, provided corresponding positions have the same interpretation. These facts explain why the two copies of a reused pad could cancel in Chapter 1. They do not establish confidentiality on their own. A public or predictable mask can still be removed by anyone who knows it. A useful cryptographic construction has to explain how the combined values are selected and what information is available to the attacker.

XOR differs from ordinary integer addition. Combining the four-bit patterns 0011 and 0001 with XOR gives 0010, whose value is 2. Adding their unsigned integer values gives

$$3 + 1 = 4$$

, represented as 0100. Ordinary addition carries from one bit position to the next when necessary; XOR does not. Although XOR is addition modulo two at each individual bit position, it is not generally the same as adding two multi-bit integers modulo a power of two. Keeping that distinction clear will be important when AES and ChaCha20 use different kinds of operations.

Concatenation is different again: it places one sequence after another. We will write

$$x \parallel y$$

when we need an explicit symbol for joining sequences. If

x

is the one byte 5A and

y

is the one byte 0F, their concatenation is the two-byte sequence 5A 0F. No bits have been added or XORed. The operation changes the length by joining the inputs. Table [3.1](#) compares the three operations on the same displayed values.

Table 3.1. Three different operations on the values displayed as 5A and 0F.

Operation	Result in hex	Meaning
XOR	55	Compare corresponding bits; no carries.
Integer addition	69	Add decimal 90 and 15 to obtain 105. The sum fits in one byte.
Concatenation	5A 0F	Join two one-byte sequences to produce two bytes.

A formula or program must tell us which of these operations is intended. The same written plus sign can even mean different things for different data types in a programming language. In the Python examples, integer addition combines numeric values, while adding byte sequences concatenates them. The caret character ^ applies bitwise XOR to integers; it is not Python's exponent operator. Reading a calculation therefore requires both the operator and the type of each input. We will make both explicit in the optional program.

3.4 Arithmetic with a fixed word size

A byte is one useful unit, but algorithms often work on larger groups of bits called **words**. A word's size must be stated in the relevant specification. For example, the ChaCha20 description in RFC 8439 works with 32-bit unsigned words, and its additions are performed modulo 2^{32}

.[\[15\]](#) The word size in that algorithm does not change merely because the computer running it has a different processor word size. An implementation must reproduce the algorithm's defined operations. For an unsigned

w

-bit word, the possible values run from 0 to

$$2^w - 1$$

. Here

w

is the number of bit positions, and

$$2^w$$

counts all their possible patterns. For eight bits, the range is 0 through 255. The value 256 requires a ninth position if it is written as an ordinary binary integer. If an operation is defined to retain an eight-bit unsigned result by wrapping, that extra contribution is removed according to a precise rule. It must not simply be left to whatever behavior a particular language happens to choose.

Addition with this wrapping convention can be written

$$(3.5)$$

The name

$$\text{add}_w$$

means the defined

w

-bit addition, while

a

and

b

are the unsigned input values. First calculate their ordinary sum. The expression “mod

$$2^w$$

” means take its remainder after division by

$$2^w$$

, leaving a result in the permitted range. Section [3.5](#) develops that remainder rule more fully. For now, the equation states exactly what is retained instead of using the vague instruction to ignore overflow.

In the eight-bit case,

$255 + 2 = 257$

. Since

$257 - 1 \times 256 = 1$

, the wrapped result is 1, written as the byte 01 in hex. Similarly, subtracting 5 from 3 gives the ordinary integer

−2

, whose representative in the eight-bit wrapping range is 254, or hex FE. A language with unbounded-size integers will not necessarily wrap its values automatically. A language using a fixed representation may have other rules for signed and unsigned arithmetic. The algorithm’s requirement must be implemented deliberately, including the input ranges.

The word “unsigned” prevents another ambiguity. The same eight bits can be interpreted under a signed-number convention instead of as a value from 0 through 255. It is the interpretation and the operation’s rules that determine the numeric meaning. In this chapter’s examples, byte arithmetic uses unsigned values unless stated otherwise. We do not infer a signed interpretation from the leftmost bit. A protocol or algorithm can define signed fields, but it must specify how those fields are represented and handled.

Shifts and **rotations** also require an agreed width. In a logical left shift, bits move toward the higher-weight positions and zeros enter at the right. If the result is restricted to a fixed width, the bits that leave the left edge are discarded. A left rotation instead brings those leaving bits around to the right. For an eight-bit example, compare the following results:

The three leading bits of the original are 101. The shift drops them and appends three zeros. The rotation appends those same three bits after the remaining five, so no bit is discarded. The shift result is hex 88; the rotation result is hex 8D. These are different operations despite sharing the same direction and movement count.

Representing a larger word in bytes introduces **byte order**, often called endianness. In a big-endian unsigned representation, the first byte has the greatest positional weight. In a little-endian representation, it has the

smallest. Consider the two-byte sequence 01 02. As a big-endian integer it has value

$$1 \times 256 + 2 = 257$$

. As a little-endian integer it has value

$$1 \times 1 + 256 = 257$$

. The sequence of bytes has not changed. We have changed the rule used to interpret it as one number. Byte order is not a command to reverse the individual bits in each byte. The byte 01 still has the same eight-bit pattern; the question is which power of 256 supplies its weight in the multi-byte number. This also differs from the convention used to label bit positions in a written diagram. FIPS 197 specifies its own indexing of sequences, bytes and words, while RFC 8439 specifies how the bytes of its state are interpreted. The relevant rule must be read at the right level. A label such as “32-bit” states a width, not a complete serialization format.

These operations are useful because their behavior is precise, not because any one of them is a complete security mechanism. A known rotation can be reversed by rotating the other way. A word addition can be reversed with the corresponding subtraction. Later constructions combine defined operations with keys and other inputs, and their security needs its own analysis. Our purpose here is to be able to follow those operations faithfully, including the boundaries at which carries, bits or bytes are interpreted.

The next operation asks a familiar question: what is left after taking away complete groups? From 17 objects, taking away one group of 12 leaves 5. That leftover is a **remainder**. Modular arithmetic follows these remainders instead of keeping every complete group. The **modulus** is the positive number that defines the group size, here 12. This idea lets a calculation return to a fixed range; it is the reason that a large number can have a small result in the examples below.

3.5 Remainders and modular arithmetic

A remainder records what is left after removing a whole number of groups of a specified size. Dividing 17 into groups of twelve gives one complete group and five left over. We write

$$17 = 1 \times 12 + 5$$

. If we care only about the position in a repeating cycle of twelve, then 17 and 5 lead to the same position. This is the starting idea of **modular arithmetic**. The number twelve is the **modulus**, the size of the repeating cycle in this example.

For any integer

a

and positive integer modulus

n

, we can express division with a remainder as

$$(3.6)$$

Here

q

is an integer counting complete multiples of

n

, and

r

is the remainder. The inequalities say that

r

is at least zero and smaller than

n

. The notation

$$a \bmod n$$

denotes this

r

. For the example,

$$a = 17$$

,

$$n = 12$$

,

$$q = 1$$

and

$$r = 5$$

. The remainder is unique when the range is specified this way. The quotient

q

can be negative if the starting integer is negative.

A clock gives a familiar picture of a cycle, but its labels can conceal the zero position. In arithmetic modulo twelve, the representatives we use are 0 through 11. A clock face that labels one position 12 can display that same cycle with a different label at zero. We will state the numeric range in calculations rather than letting the drawing decide it. The purpose of the clock analogy is to explain repeated positions, not to introduce an exception in which 12 becomes a second representative inside the range 0 through 11.

Two integers are **congruent modulo**

n

when their difference is a multiple of

n

. We express the example as

$$(3.7)$$

The symbol

$\equiv$

is read “is congruent to,” and the expression at the end states the modulus. Since

$$17 - 5 = 12$$

, the difference is one multiple of twelve. This does not say that 17 and 5 are equal as ordinary integers. It says that the chosen modular calculation treats them as representatives of the same position. Confusing the two statements can make an otherwise correct derivation appear to change a number without justification.

The remainder convention also works for negative inputs. For

-3

with modulus twelve, write

(3.8)

The quotient is

-1

and the remainder is 9, which lies in the specified range. On a cycle, moving three positions backward from zero reaches the same position as moving nine forward. Programming languages can use different remainder conventions in some settings, so a part of a mathematical description must check how its operators behave. The Python examples here use a positive modulus and the nonnegative representative just defined.

We can reduce intermediate results when calculating sums and products. For example,

$$29 + 31 = 60$$

, and the remainder on division by twelve is 0. Reducing each input first gives

$$29 \bmod 12 = 5$$

and

$$31 \bmod 12 = 7$$

. Their sum is 12, which again reduces to 0. The same principle gives

(3.9)

The first replacement uses the fact that 17 and 5 differ by one modulus. The ordinary product

$$17 \times 9$$

is 153, and

$$153 - 12 \times 12 = 9$$

, confirming the same remainder. The intermediate reduction is useful because larger cryptographic calculations often need to keep values within a defined range.

Division cannot be carried over without extra reasoning. Modulo twelve,

$$2 \times 1$$

and

$$2 \times 7$$

are congruent: their ordinary values are 2 and 14, which differ by twelve. Cancelling the factor 2 as if this were an equality of ordinary integers would incorrectly conclude that 1 and 7 are congruent modulo twelve. Their difference is not a multiple of twelve. Modular division requires an appropriate **inverse**, a value that reverses multiplication under the specified modulus, and such a value need not exist. Chapter 4 will explain when it exists and how to calculate it.

It is also important to identify which objects and operations a modulus applies to. Integer addition modulo 256 is not the same as adding two byte values in the finite field used by AES. The latter adds corresponding binary coefficients with XOR; it does not carry between positions as an integer sum does.^[13] We will develop the field interpretation when AES needs it. A shared count of 256 possible byte patterns does not make different arithmetic rules interchangeable.

Modular arithmetic therefore gives us more than a way to keep a number small. It defines the relationships preserved when values are reduced, the representatives used in a calculation and the operations that are valid. With those conventions stated, a remainder in a formula becomes a step we can check. Without them, a program may silently use the wrong range, width or division rule while still producing an output that looks plausible.

3.6 Reading formulas and small programs

Read a formula by identifying the objects before calculating. Are the inputs individual bits, byte sequences, unsigned words or arbitrary integers? What does the operator mean for those objects, and where is the width or modulus specified? Then follow the order of the operations and interpret the output under the same rules. Parentheses group the intended inputs. A subscript may identify a position, a base or a word size, depending on its definition; a superscript used as an exponent means repeated multiplication. The surrounding explanation must identify that role rather than expecting the shape alone to settle it.

Programs add another layer of notation. In Python, `=` assigns a value to a name, while `==` compares values. The expression `a ^ b` computes bitwise XOR on integers, `a ** b` computes a power and `a % n` produces a remainder for these integer examples. A byte sequence is a different type

from an integer or a text string. Python's `bytes` represents a sequence of byte values; iterating through it produces integers from 0 through 255. Its documented operations include converting to hex, decoding text and interpreting a sequence as an integer with a specified byte order.^[16]

Before the longer example, compare two small constructions. The expression `b"A"` is a one-byte literal containing the ASCII byte for A. The expression `bytes.fromhex("41")` decodes a hexadecimal display and gives the same one-byte value. However, `b"41"` contains the two literal text bytes for the characters 4 and 1. Calling `.hex()` on that last value produces `3431`, making the difference visible. The quoted text inside a function call can describe an encoding to parse; it is not necessarily the byte sequence that the function returns.

Listing 3.1 applies the XOR rule to the two-byte ASCII message `OK`. Both the message and mask are fixed and public teaching inputs. The function first requires equal lengths, then pairs corresponding byte values, XORs each pair and collects the results into a new byte sequence. It does not generate a secret, provide authentication or implement a complete encryption construction. Its purpose is to connect the earlier paper calculation to the types and operators used in a real programming language.

Listing 3.1. An optional byte-mask calculation with known inputs.

```
def xor_bytes(left, right):
    if len(left) != len(right):
        raise ValueError("Inputs must have the same length")
    result = []
    for a, b in zip(left, right):
        result.append(a ^ b)
    return bytes(result)

message = b"OK"
mask = bytes.fromhex("13 37")
ciphertext = xor_bytes(message, mask)
recovered = xor_bytes(ciphertext, mask)

print(message.hex())
print(ciphertext.hex())
print(recovered.decode("ascii"))
```

The words `def`, `if`, `for` and `return` describe a function, a condition, repeated work and the returned result. Indentation identifies which statements belong inside those steps. The function `zip` pairs the inputs at corresponding positions. Its use here follows an explicit length check,

because pairing by itself would stop at the shorter input and could hide a mistake. The result list collects integers, and `bytes(result)` turns those byte values into a byte sequence. A reader can understand the operation without learning every feature of Python.

The expected output is the following three lines:

```
4f4b
5c7c
OK
```

The first line is the hex representation of the original two bytes. The first message byte has value `4F`, and XOR with the first mask byte `13` gives `5C`. The second message byte `4B` XOR `37` gives `7C`. Applying the same mask again restores the message bytes, which are then decoded as ASCII to print `OK`. The variable name `ciphertext` labels the transformed value in this demonstration; the public fixed mask does not give it a confidentiality guarantee.

There are other small observations a reader can make without modifying the function. In Python, `int.from_bytes(bytes.fromhex("01 02"), "big")` gives 258, while selecting `"little"` gives 513. The order argument identifies the numeric interpretation. Similarly, `(255 + 2) % 256` gives 1, whereas the ordinary integer expression `255 + 2` gives 257. Python does not automatically infer eight-bit wrapping from the fact that we were discussing bytes. These results repeat the rules already stated rather than letting the programming language silently supply a new mathematical model.

A useful check includes an expectation derived independently of the program. If the program encrypts and decrypts with two matching mistakes, a successful round trip may still hide the error. Here the byte values, XOR outputs and byte-order calculations were worked out before running the code. A second check passes unequal-length inputs and expects rejection. That verifies a stated condition of this small function; it does not prove that an entire cryptographic library or application is secure. Keep the scope of the observation attached to the result.

We now have a language for the representations and elementary operations used throughout the rest of the book. A byte display identifies

data only when its interpretation is clear. XOR, concatenation and integer addition have different meanings. Width and byte order belong to the operation being implemented, and modular arithmetic supplies precise rules for reducing integer calculations. Chapter 4 builds on these foundations with primes, greatest common divisors, inverses and groups. Those tools will explain why some operations are easy to perform and why particular reverse problems are used as the basis for cryptographic assumptions.

Questions to check your understanding

1. Why can the visible word *Café* occupy more than four bytes in UTF-8? Does identical appearance establish identical bytes?
2. Convert the byte `10100111` to hexadecimal and unsigned decimal. State which notation the prefix `0x` identifies.
3. Explain the difference between decoding the hex text `41` into bytes and passing the literal text bytes `b"41"`.
4. For the one-byte inputs `5A` and `0F`, give the results of XOR, integer addition and concatenation. Which result has two bytes?
5. Calculate

$(28 + 7) \bmod 30$

 and

$-3 \bmod 12$

 . Explain why a fixed-width left shift and a rotation can give different results.
6. Why can the byte sequence `01 02` mean either 258 or 513? Does changing that numeric interpretation reverse each byte's individual bits?
7. Why does the XOR program reject unequal-length inputs before pairing them? What has a successful run with these fixed inputs established about security?

4. Primes, Groups and Hard Problems

Part II: The Mathematics We Need

Chapter 3 explained how a remainder keeps a calculation inside a specified range. That gives us a way to calculate, but several questions remain. Can we always undo multiplication? How can a computer calculate a power with a very large exponent? Why do cryptographers choose particular sets of numbers instead of using any convenient modulus? Answering these questions will connect the arithmetic to the security arguments that appear later in the book. We will begin with ordinary whole numbers and build the new ideas one step at a time.

The important distinction is between an operation that has a reverse and a reverse that is difficult to find. Multiplication by 7 modulo 26 can be undone efficiently once we understand inverses. Raising a public group element to a large exponent can also be efficient, while recovering an unknown exponent may be difficult in carefully chosen groups. Those are different reverse questions. Neither the presence of a prime nor the size of a number settles all of them. We need to identify the operation, the information supplied and the result an attacker is asked to produce.

Our examples use small numbers so that every step can be checked. They establish arithmetic relationships, not useful protection for a secret. The same definitions will later support larger constructions, where parameter choices, randomness, authentication and implementation behavior also matter. This chapter provides the mathematical language for those constructions. The probability chapter will then explain how researchers state and assess an attacker's chances of success.

4.1 Divisibility and prime numbers

An integer

d

divides an integer

n

if there is an integer

t

such that

$$n = dt$$

. For example, 7 divides 91 because

$$91 = 7 \times 13$$

. We write this as

$$7 \mid 91$$

, reading the vertical bar as “divides.” By contrast, 7 does not divide 93: division leaves a remainder of 2. A divisor is also called a factor. Unless another range is stated, the divisors considered in this section are positive. The quotient must be an integer; merely being able to enter a division on a calculator is not the same condition.

A **prime number** is an integer greater than 1 whose only positive divisors are 1 and itself. Thus 2, 3, 5, 7 and 11 are prime. The number 1 is not prime, and 2 is the only even prime: any larger even integer has 2 as an additional divisor. An integer greater than 1 that is not prime is **composite**. The number 91 is composite because its factorization

$$91 = 7 \times 13$$

uses two integers greater than 1. These definitions classify the ordinary integers, before any modular interpretation is introduced.

Every integer greater than 1 can be written as a product of primes. Apart from changing the order of the factors, the prime factors and their repetitions are unique. For example,

$$(4.1)$$

The exponent 2 above the first factor says that 2 appears twice. It does not turn 2 into a different kind of prime. Repeated factors matter when we describe a modulus or the size of a group. Two numbers may have some prime factors in common without being equal, and a number with several

prime factors can still have useful invertible residues. We will make that last statement precise in the next section.

One direct primality test tries possible divisors. We do not need to test every integer smaller than the candidate. If a positive integer

n

has a factorization

$$n = uv$$

with

$$u > 1$$

and

$$v > 1$$

, at least one of these factors is at most

$$\sqrt{n}$$

. Otherwise both would be larger than

$$\sqrt{n}$$

and their product would exceed

n

. The symbol

$$\sqrt{n}$$

means the nonnegative square root of

n

. This observation gives a stopping point for trial division: once all possible prime divisors up to that point have been ruled out, a candidate greater than 1 is prime.

Consider 97. Since

$$9^2 = 81$$

and

$$10^2 = 100$$

, its square root lies between 9 and 10. The possible prime divisors to test are therefore 2, 3, 5 and 7. The number is odd, its decimal digits sum to 16, and it ends in neither 0 nor 5, ruling out the first three candidates. For 7, we have

$$97 = 7 \times 13 + 6$$

, so this candidate also fails to divide it. That completes a small trial-division proof of primality. Applying the same method to 91 instead finds

the factor 7 before reaching the stopping point.

This procedure is clear, but trying divisors up to a square root is unsuitable for the very large candidates used in many cryptographic constructions. The number of candidates grows rapidly with the number of bits needed to write

n

. Practical prime generation uses more effective primality tests and a specified way to select candidates. We will discuss the relevant requirements when primes become part of a key-generation method. A hand example should not be mistaken for an instruction to generate a real key by testing a short list of small divisors.

Primality testing and **factorization** also ask different questions. A primality test decides whether a candidate is prime. A factorization procedure decomposes a composite integer into factors, or ultimately into primes. Learning that a number is composite does not necessarily supply its prime factors. Conversely, a proposed factor can be checked by division. If someone claims that

$$323 = 17 \times 19$$

, multiplication verifies that claim immediately. The cost of finding the factors without being given them is a different question, whose answer depends on the size and structure of the input.

A useful test condition can be necessary without being sufficient. Fermat's little theorem states that, for a prime

p

and an integer

a

not divisible by

p

,

(4.2)

Here

$$p - 1$$

is the exponent, and the congruence means that the power has remainder 1 on division by

$$p$$

. For instance,

$$2^6 = 64 = 8 \times 7 + 1$$

agrees with the theorem for

$$p = 7$$

. The theorem gives a consequence of primality under its stated condition. It does not say that passing this check for one selected base proves a candidate prime.^[7]

The composite integer

$$341 = 11 \times 31$$

illustrates the difference. We have

$$2^{340} = 1081 \times 10^{10} + 1$$

, so

$$2^2 = 1 \pmod{341}$$

. Raising that relationship to the power 34 gives

$$2^{34} = 1 \pmod{341}$$

. It passes this particular base-2 condition despite being composite. The repeated-squaring method later in the chapter makes such powers easy to check. For now, the example shows why a cryptographic requirement must name the actual test and its guarantees rather than relying on the informal phrase “it behaves like a prime.”

Primes are important because they give modular arithmetic a particularly useful structure. When the modulus is prime, every nonzero residue has a multiplicative inverse. With a composite modulus, some residues have inverses and others do not. That difference controls which equations can be solved by division and which sets form multiplicative groups. A prime is therefore a mathematical building block with specific consequences, not a general certificate of cryptographic security.

An **inverse operation** undoes another operation under the rules being used. Adding 3 can be undone by subtracting 3. Multiplication needs more care when we keep only a remainder. We want to know whether multiplying by one allowed number can undo multiplication by another. This question

leads to the modular inverse. Before finding one, we need a way to recognize which numbers share a factor; that is the purpose of the greatest common divisor.

4.2 Greatest common divisors and modular inverses

The **greatest common divisor** of two positive integers is the largest positive integer that divides both. We abbreviate it as

$$\gcd(a, n)$$

. The positive divisors of 18 are 1, 2, 3, 6, 9 and 18; those of 30 are 1, 2, 3, 5, 6, 10, 15 and 30. Their largest common divisor is 6, so

$$\gcd(18, 30) = 6$$

. When the greatest common divisor is 1, the integers are **coprime**, also called relatively prime. Neither integer has to be prime: 8 and 15 are coprime although both are composite.

Listing all divisors is unnecessary. The **Euclidean algorithm** repeatedly replaces a division problem by a smaller one involving its remainder. Its central relationship is

$$(4.3)$$

In this equation

a

and

n

are positive integers and

$$a \bmod n$$

is the remainder defined in Chapter 3. If the remainder is zero, the process ends with the nonzero divisor, using

$$\gcd(n, 0) = n$$

. To see why the relationship holds, write

$$a = qn + r$$

. Any integer dividing both

a

and

n

also divides

$$r = a - qn$$

. Conversely, an integer dividing both

n

and

r

divides

$$a = qn + r$$

. The two pairs therefore have exactly the same common divisors.

For 26 and 7, the successive divisions are

$$(4.4)$$

The remainders decrease from 5 to 2 to 1 to 0. The last nonzero remainder is 1, establishing

$$\gcd(26, 7) = 1$$

. Each step preserves the common divisors while reducing the values still to be handled. There is no need to factor 26 or 7 first. The same method applies to large integers and is a basic part of efficient number-theoretic computation.

To undo multiplication modulo

n

, we need a **multiplicative inverse**. For an integer

a

and modulus

$$n \geq 2$$

, an inverse is an integer

u

satisfying

(4.5)

The value

u

is often written

a^{-1}

when the modulus is understood. The superscript

-1

here names a modular inverse; it is not a request to calculate the ordinary fraction

$1/a$

and round it. Multiplying by

a

and then by

u

restores the starting residue. The target 1 is the multiplicative identity, the value that leaves a product unchanged.

Such an inverse exists exactly when

$\gcd(a, n) = 1$

. An explanation also provides a way to find it. The extended Euclidean algorithm expresses the greatest common divisor as an integer combination

(4.6)

The coefficients

s

and

t

are integers and may be negative. If the right side is 1, reducing the equation modulo

n

removes the term

tn

and leaves

$ax \equiv 1 \pmod{n}$
. Thus

s
, reduced to the desired range, is an inverse of
 a

. In the other direction, if

$ax = 1 + vn$
for some integer

v
, any common divisor of
 a

and
 n

must divide 1, so their greatest common divisor must be 1. This proves both the existence condition and its necessity.^[7]

We can obtain the coefficients for 7 and 26 by working backward through the earlier divisions. Each replacement below uses one of the recorded remainders:

$$(4.7)$$

Here
 $a = 7$

,
 $n = 26$

,
 $s = -11$

and
 $t = 3$

. Reducing the coefficient
 -11

modulo 26 gives 15, because

$$-11 + 26 = 15$$

. Therefore the inverse of 7 modulo 26 is 15. A direct check gives

$$7 \cdot 15 = 105 = 4 \cdot 26 + 1$$

. The remainder is exactly the 1 required by Equation [4.5](#).

This inverse allows us to solve an equation without guessing every possible input. Suppose a deliberately simple letter-number exercise maps an input

x

to

y

by

$$(4.8)$$

Both

x

and

y

are represented in the range 0 through 25. To recover

x

, subtract 3 and multiply by the inverse 15:

$$x = 15(y - 3) \pmod{26}$$

. For

$$x = 4$$

, the forward calculation gives

$$7 = 4 + 3 = 21$$

, whose remainder is

$$y = 5$$

. The reverse gives

$$15(5 - 3) = 30$$

, whose remainder is 4. This small affine mapping illustrates reversible arithmetic. Its tiny set and simple structure make it unsuitable as modern encryption.

Now consider 6 modulo 26. Its greatest common divisor with 26 is 2. Every integer combination of 6 and 26 is even, so it cannot equal 1. There is no multiplicative inverse of 6 in this setting. In particular,

$$6x \equiv 1 \pmod{26}$$

has no solution. This does not mean that every equation involving multiplication by 6 is impossible. The equation

$$6x \equiv 4 \pmod{26}$$

has the two distinct solutions 5 and 18 in the range 0 through 25: the products are 30 and 108, both with remainder 4. Multiplication by 6 loses the distinction between these two inputs.

When an inverse exists, it is unique as a residue modulo

$$n$$

. There can be many ordinary integers representing it: 15 and 41 both invert 7 modulo 26 because they differ by 26. They describe the same residue. If multiplication has an inverse, applying that inverse also justifies cancellation. This explains the failed cancellation in Chapter 3: 2 has no inverse modulo 12. Cancelling it was not a permitted reversal of the operation. With an appropriate inverse, the reasoning would have been valid. For a prime modulus

$$p$$

, every integer in the range 1 through

$$p - 1$$

is coprime to

$$p$$

, so each has an inverse. Zero does not. We can therefore solve an equation

$$ax \equiv b \pmod{p}$$

uniquely for

$$x$$

when

$$a$$

is nonzero modulo

$$p$$

. With a composite modulus, the gcd condition must be checked. The distinction between the modulus, its allowed input residues and the

operation being reversed will continue to matter even when all three values become much larger.

4.3 Fast powers by repeated squaring

A power such as
 a^e

means multiplying

e

copies of

a

when

e

is a positive integer. In the expression

$a^e \bmod n$

, we want the remainder of that product after division by

n

. It would be wasteful to construct an enormous ordinary integer and only then reduce it. Chapter 3 established that reducing intermediate products preserves the final remainder. We can therefore keep each intermediate result inside the modular range. The next improvement is to avoid doing one multiplication for almost every copy of

a

.

Repeated squaring uses the fact that squaring a power doubles its exponent. Starting from

a

, one square gives

a^2

, another gives

a^4

, then

a^8

,
 a^{16}

and so on. The exponents grow quickly while the number of squaring steps grows slowly. If we write the desired exponent in binary, its nonzero positions tell us which of these powers to multiply. This connects the binary place values from Chapter 3 to an efficient calculation with much larger exponents. [\[17\]](#)

We will calculate

$$7^{13} \bmod 23$$

. The base is 7, the exponent is 13 and the modulus is 23. Since

$$13 = 8 + 4 + 1$$

, we need the powers with exponents 8, 4 and 1. Starting with the base, compute and reduce each square:

$$(4.9)$$

The second line uses the already reduced representative 3 instead of storing 49. The third uses 9 instead of expanding

$$7^4$$

into a larger integer. None of those substitutions changes the required remainder. Three squaring operations have supplied all the separate powers we need.

Now combine those selected powers:

$$(4.10)$$

The answer is 20. After the three squares, we used two more multiplications to combine the selected factors. Straight multiplication of thirteen copies would use twelve multiplications. Different implementations can organize the steps differently, but this five-operation chain shows the main saving without hiding any arithmetic.

The binary representation

$$13 = (1101)_2$$

explains the selection directly. Reading the positions from right to left gives weights 1, 2, 4 and 8. The bit at weight 2 is zero, so the power

$$7^2$$

is not included in the final product. It was still calculated as an intermediate step on the way to

$$7^4$$

and

$$7^8$$

. A zero in the exponent's binary representation means "do not select this power," not "replace the base by zero." This distinction makes the usual square-and-multiply procedure easier to follow.

One version processes the bits from right to left. It begins with a running result of 1 and a current power equal to

$$a \bmod n$$

. For each bit of the exponent, it multiplies the running result by the current power if that bit is 1, then advances to the next squared power. Each multiplication and square is reduced modulo

$$n$$

. For a nonnegative exponent

$$e = 0$$

, no powers are selected, so the result is 1 for the moduli

$$n \geq 2$$

considered here. The general procedure works for composite as well as prime moduli; calculating a nonnegative power does not require the base to have an inverse.

The efficiency is measured against the length of the exponent's representation. A positive integer with

$$L$$

binary digits has

$$L$$

positions to process, regardless of the very large value those positions encode. A straightforward binary method needs at most about

$$2L$$

modular multiplications, counting squares, rather than about

e

multiplications. This is a count of modular operations, not a claim that their individual cost is constant. Multiplication of large integers costs more than multiplication of small ones, and the modulus's length also affects the work. For perspective, an exponent occupying 256 bits gives only 256 bit positions to visit in such a method. Its ordinary numeric value can be far larger than 256. That is why efficient exponentiation is compatible with exponents that cannot plausibly be handled by repeating one multiplication for every unit in their value. This example describes an algorithmic scaling relationship. It does not say that every construction with a 256-bit exponent has a particular security level, nor does it choose parameters for a real protocol.

Efficient forward computation matters for security because intended participants must be able to perform it. However, a clear mathematical algorithm is not automatically a safe implementation for secret inputs. In a simple square-and-multiply program, the choice to perform an extra multiplication depends on an exponent bit. Execution time or memory behavior can then reveal information about a secret exponent. Making the final answer correct does not remove that possibility. The detailed implementation chapter will examine such side channels; here it is enough to distinguish the arithmetic method from the requirements for keeping its inputs secret.

The collected OpenSSL source illustrates that distinction. Its modular-exponentiation documentation describes a variant that uses fixed windows and a particular layout of precomputed values to reduce dependence on secret data. The associated implementation contains dedicated handling for this purpose.^[20] That is a concrete engineering concern beyond the equations above. The observation concerns the documented routine and selected source passages; it is not a claim that every surrounding application is secure. Our optional calculations use public small inputs, and we will leave real secret-key operations to appropriate cryptographic interfaces.

Fermat's theorem also supplies a second way to express an inverse when the modulus is prime. If

p

is prime and

a

is not zero modulo

p

, then

(4.11)

The exponent

$p - 2$

is two less than the prime modulus. Repeated squaring makes that power practical to calculate; the extended Euclidean method gives another route. The equation's prime-modulus and nonzero-input conditions are essential. Substituting an arbitrary composite modulus into

a^{p-2}

does not turn it into a general inverse formula. Choosing an implementation for sensitive values also requires considering its behavior beyond the returned result.

4.4 Groups, orders and generators

A **set** is a collection of allowed objects, and an **element** is one object in that collection. The set

$\{1, 2, 3\}$

, for example, has three elements. **Finite** means that the collection has a fixed, limited number of elements. A group adds a rule for combining elements to a chosen set. We study these rules because a cryptographic calculation must stay within its allowed values and must permit the reversals that the legitimate receiver needs. The word group is a mathematical description, rather than a claim that all the values are secret.

We have seen several sets in which similar-looking operations behave differently. Multiplication modulo a prime can undo every nonzero input, while multiplication on all residues of a composite modulus cannot. A **group** is a way to describe a set and one operation whose behavior obeys four rules. The operation need not be ordinary multiplication, although we

will initially write it that way. Naming the set without naming its operation is incomplete: the same residues can behave differently under addition and multiplication.

First, the operation is **closed**: combining any two elements of the set gives another element of that set. Second, it is **associative**: grouping three inputs in different ways does not change the result. In multiplicative notation this means

$$(ab)c = a(bc)$$

. Associativity does not allow us to change the order of the inputs. The additional rule

$$ab = ba$$

is called commutativity; a group satisfying it is **abelian**. The groups used in our immediate examples are abelian, but commutativity is not required by the general definition. Third, a group has an **identity**, an element that leaves any input unchanged when combined on either side. In multiplicative notation we call it 1, so

$$1a = a1 = a$$

. Fourth, each element has an **inverse** in the set that combines with it to give the identity. These rules make repeated use and reversal of the operation consistent. A set containing an element that has no inverse for the chosen operation cannot be a group under that operation. [\[17\]](#)

Take the nonzero residues modulo 11: the set

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

, with multiplication followed by reduction modulo 11. Products remain nonzero because 11 is prime. Multiplication is associative, 1 acts as the identity and every element has an inverse. We therefore have a group. The **order of a finite group** is its number of elements, written

$$|G|$$

for a group named

$$G$$

. The two vertical bars in this expression mean “number of elements,” rather than divisibility. This group has order 10 even though its arithmetic uses modulus 11.

With a composite modulus, the invertible residues can still form a group. Modulo 12 they are

$$\{1, 5, 7, 11\}$$

, precisely the residues coprime to 12. Each is its own inverse:

$$5^2 = 25$$

,

$$7^2 = 49$$

and

$$11^2 = 121$$

all have remainder 1, as does

$$1^2$$

. Products of invertible residues remain invertible, so the operation stays inside the set. The full set of residues 0 through 11 instead includes elements such as 2 with no inverse and is not a group under multiplication. Zero is excluded from every nontrivial multiplicative example here because it cannot multiply with anything to give 1.

Now select an element

$$g$$

of a finite multiplicative group and calculate its powers. The power

$$g^0$$

means the identity, and

$$g^{k+1}$$

means combine

$$g^k$$

with

$$g$$

once more. The resulting sequence eventually returns to the identity. The smallest positive exponent

$$r$$

for which

$$g^r = 1$$

is the **order of the element**

$$g$$

. The group order counts the whole set; an element's order counts the length of the cycle generated by that particular element. These quantities can

differ.

For the full nonzero group modulo 11, powers of 2 give

For example, the step after 8 gives

$$8 \times 2 = 16$$

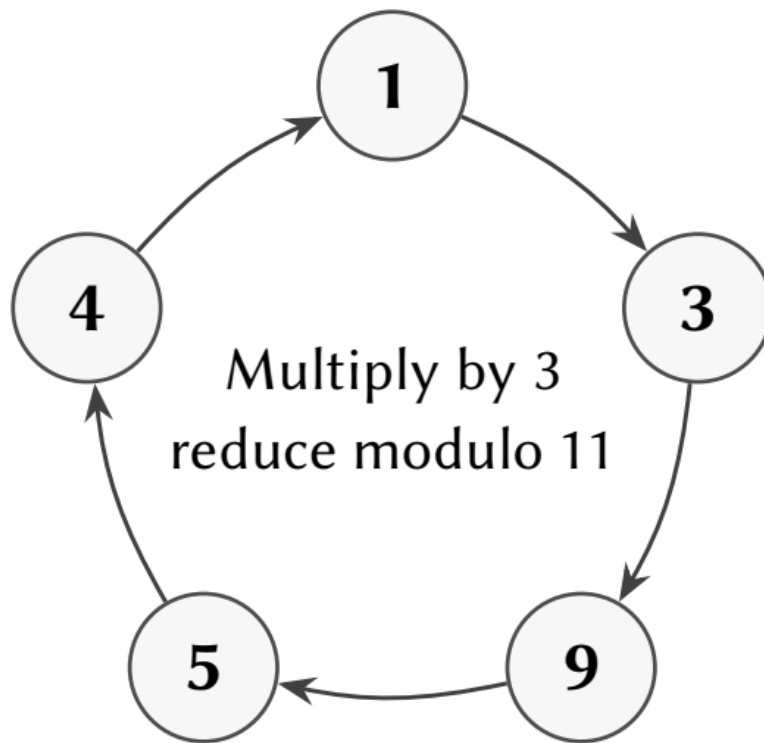
, reduced to 5. After the entry 6, multiplying once more gives 12, reduced to 1. No earlier positive exponent returned to 1. The order of 2 is therefore 10, and its powers visit every element of the group. An element with that property is a **generator**, and a group with a generator is **cyclic**.

The element 3 behaves differently in the same full group. Its powers give

$$1, 3, 9, 5, 4, 1$$

, returning to the identity at exponent 5. Its order is 5, so it does not generate all ten nonzero residues. However, its five distinct powers form a group of their own under the same multiplication rule. A group contained in a larger group with the same operation is a **subgroup**. Figure [4.1](#) shows this smaller cycle, which we will also use to distinguish reverse problems in the final section.

Start at the identity



Five distinct elements; $3^5 \equiv 1 \pmod{11}$

Figure 4.1. The powers of 3 modulo 11 form a five-element subgroup. Each arrow multiplies the previous residue by 3 and reduces modulo 11. Starting from the identity, the cycle visits

before returning to 1. The modulus is 11; the subgroup order is 5.

Closure can be checked without listing every product. If two elements are powers

$$g^a$$

and

$$g^b$$

, their product is

$$g^{a+b}$$

, another power of

$$g$$

. For a cycle of order

r

, the inverse of

g^a

is represented by

g^{-a}

, with the exponent reduced modulo

r

. In the five-element example, 3 and 4 are inverses because

. Likewise, 9 and 5 are inverses because

. The identity in the smaller group is still 1, and associativity is inherited from the original operation.

In any finite group, an element's order divides the group's order. This is a consequence of Lagrange's theorem. In our example, 5 divides 10. For a short explanation, the powers of a chosen element make a subgroup

H

. Multiplying all elements of

H

by a fixed group element produces a translated set of the same size. Any two such translated sets are either identical or disjoint, so these sets partition the full group into equally sized pieces. The number of elements in

H

must therefore divide the total. This counting argument explains why the cycle lengths fit the group's structure rather than being arbitrary.^[17]

A cyclic group's exponents can be reduced modulo its order. If

g

has order

q

, then

$g^q = 1$

, and adding

q

to an exponent gives the same group element. In particular,

$$(4.12)$$

Here

g

is the chosen element,

q

is its order and

a, b

are integer exponents. Products on the right use the group operation. In the subgroup generated by 3 modulo 11,

3^7

and

3^2

agree because

$$7 - 2 = 5$$

. Their common residue is 9. The exponent is reduced modulo 5, while the represented group elements are multiplied modulo 11. Confusing these two moduli would change the calculation.

For a prime group order

q

, every element except the identity generates that group. Its order must divide

q

, so the only possibilities are 1 and

q

; only the identity has order 1. This property helps explain the frequent use of large prime-order subgroups. It does not follow that the nonzero

multiplicative group modulo a prime has prime order. For modulus 11 the full group order is 10, which is composite. Both the prime defining the arithmetic and the order of the selected group must be stated.

Not every group is cyclic. The invertible residues modulo 12 give a concrete example: every nonidentity element has order 2, so none generates the whole group of order 4. Also, a cyclic group does not have to be drawn as a geometric circle. The cycle describes repeated application of an operation. For addition modulo

n

, the identity is 0 and the element 1 generates all residues by repeated addition. In this additive notation, the analogue of

g^a

is

aP

, meaning repeated addition of a group element

P

. It does not mean multiplying the coordinates of a point by an ordinary number.

Elliptic-curve cryptography uses groups whose operation is conventionally written as addition. Koblitz's original paper describes point addition and computing a multiple of a point by repeated doubling and addition, the analogue of repeated squaring.^[18] The detailed point formulas belong in Chapter 12. What transfers now is the language: identity, inverse, element order, subgroup and generator. A selected cyclic subgroup can be used even when the full group of curve points is not cyclic. Reusing the abstract rules does not make every group's reverse problems equally difficult.

Optional: checking the arithmetic

The short Python example in Listing [4.1](#) checks several results already established on paper. The function `gcd` calculates a greatest common divisor. Python's three-argument `pow(a, e, n)` calculates a modular power without first constructing the full ordinary power. For an invertible integer base, using exponent

−1

requests the modular inverse; an input without an inverse raises an error. These documented operations make a compact independent check of the examples. [\[21\]](#)

Listing 4.1. Checking public teaching values with Python.

```
from math import gcd

print(gcd(7, 26))
print(pow(7, -1, 26))
print(pow(7, 13, 23))
try:
    pow(6, -1, 26)
except ValueError:
    print("6 has no inverse modulo 26")

g, modulus, order = 3, 11, 5
print([pow(g, k, modulus) for k in range(order)])
a, b = 2, 3
A = pow(g, a, modulus)
B = pow(g, b, modulus)
print(A, B, pow(B, a, modulus), pow(A, b, modulus))
```

The expected output is

```
1
15
20
6 has no inverse modulo 26
[1, 3, 9, 5, 4]
9 5 3 3
```

The final line shows two public powers followed by the equal results obtained by applying the other exponent. The expression inside square brackets collects the five powers with exponents 0 through 4. This is an arithmetic check with public inputs, not a key-exchange implementation. It provides no authentication or secrecy. The next section explains precisely what an observer would be asked to recover and why this tiny example gives the observer no meaningful obstacle.

4.5 Easy operations and hard reverse problems

A mathematical operation does not become a useful security foundation merely because its result looks complicated. In the group of addition modulo

n

, computing

aP

is easy, but choosing

$P = 1$

also makes the reverse trivial: the result is simply

$a \bmod n$

. The group is cyclic, yet the exponent-like quantity is visible directly. We therefore need a specific group, a specified representation and an exact computational question. The word “group” supplies consistent algebraic rules; it does not supply hardness.

A **hard problem** in this discussion is a task for which no efficient solution is known under the stated conditions and assumptions. It does not mean that nobody can solve any instance. We can solve the tiny examples below by listing possibilities; the intended security question concerns appropriately generated, much larger problem cases, called instances. We must also name the task precisely. Finding an unknown exponent, calculating a shared group value and deciding whether a proposed value is correct are three different requests.

The **discrete logarithm problem**, abbreviated DLP or sometimes DL, asks for an exponent. Suppose

G

is a finite cyclic group of known order

q

, with public generator

g

. An input

A

is a group element satisfying

(4.13)

Given

G

,

q

,

g

and

A

, the problem is to recover an exponent

a

in the range 0 through

$q - 1$

. Group equality and exponentiation use the chosen group rules. Because

g

is a generator, every element has exactly one exponent in this range. Other integer exponents differing by multiples of

q

describe the same element. This is a logarithm in a finite algebraic setting, not a request to apply a calculator's real-number logarithm to the displayed residues.

In our subgroup modulo 11,

$g = 3$

and

$q = 5$

. If

$A = 9$

, the answer is

$a = 2$

, since

$3^2 = 9 \pmod{11}$

. The earlier cycle exposes the answer immediately. A general exhaustive method could also test

$g^0, g^1, \dots$

until it finds

A

. This is why the example is useful for explaining the question but useless for protecting an exponent. In a much larger, appropriately chosen group, enumerating the whole cycle is impractical. Whether there is a better attack depends on the group's structure and the available algorithms.

There are faster methods than full enumeration. Baby-step/giant-step solves discrete logarithms using roughly a square root of the group order in group operations, with substantial storage. Other methods trade resources differently. Some represented groups admit additional attacks that are faster than generic group methods. A **generic** method here treats elements through the group operations instead of exploiting special features of their representation. Consequently, two groups with the same number of elements need not offer the same resistance to attacks. A large prime modulus alone is not enough information to assign a security level. [\[7\]](#)

The **computational Diffie–Hellman problem**, abbreviated CDH, asks for a combined group element rather than either hidden exponent. Under the same public group parameters, it supplies

(4.14)

Here

a

and

b

are hidden exponents and

S

is the requested group element. Someone who knows

a

can calculate

$$B^a = (g^a)^a = g^{a^2}$$

. Someone who knows

b

can calculate

$$A^b$$

and obtain the same result. An outside observer receives

A

and

B

but is not given either exponent. The task is not to multiply

A

and

B

, because their product is

$$g^{a+b}$$

, which generally differs from

$$g^{ab}$$

.

For the tiny example, choose

$$a = 2$$

and

$$b = 3$$

. Then

$$A = 9$$

and

$$B = 5$$

modulo 11. The two participant calculations give

$$(4.15)$$

The combined exponent is

$$ab = 6$$

, which reduces to 1 modulo the subgroup order 5, so

$$g^{ab} = 3$$

is consistent with both calculations. By contrast, multiplying the public values gives

$$9 \cdot 9 = 81 \text{ and } 10$$

. This difference identifies the observer's actual problem. In our example the observer still solves it easily by recovering an exponent from the five-element cycle.

The **decisional Diffie–Hellman problem**, abbreviated DDH, asks whether a supplied candidate has the expected relationship. In its usual distribution-based formulation, one compares the two kinds of triples

$$(4.16)$$

The exponents

a

,

b

and

c

are selected independently and uniformly from 0 through

$$q - 1$$

in the comparison experiment. In the left triple, the final element has the Diffie–Hellman relationship with the first two. In the right triple, the final element is chosen independently of them. The problem is to distinguish the two experiments with a useful advantage, given the public group parameters. The probability chapter will make “uniformly” and “advantage” precise. [\[19\]](#)

An independent candidate is not guaranteed to be different from the correct element. For any fixed

$$a, b$$

, exactly one of the

q

possible values of

C

gives

$$g^c = g^{ab}$$

. Thus a random candidate matches by chance with probability

$$1/q$$

. In the five-element example this probability is

$$1/5$$

. With public values

$$A = 9$$

and

$$B = 5$$

, candidate

$$T = 3$$

satisfies the relation, while candidate

$$T = 4$$

does not. This small relationship check illustrates what the relationship means, but the DDH assumption concerns the ability to distinguish sampled experiments efficiently as the chosen parameters grow.

Table 4.1 keeps the requested outputs separate. The same powers appear in all three problems, but the information an algorithm must recover is different. Recovering an exponent, calculating a combined element and recognizing a distribution should not be used as interchangeable descriptions of security.

Table 4.1. Three computational questions under public cyclic-group parameters

Nam	Give	Requested result
e	n	Recover the exponent
DLP		modulo
Nam	Give	Requested result
e	n	Calculate the group element
CDH		
Nam	Give	Requested result
e	n	Distinguish a related
DDH		from an independently sampled

There are useful directions of implication. An efficient discrete-logarithm solver can recover a from A , then calculate B^a to solve CDH. A reliable CDH solver can calculate the related element and compare it with T , giving a DDH distinguisher. These statements describe ways to build one attack using another. They do not establish the reverse implications. In

particular, claiming that discrete logarithms are hard is not by itself a proof that DDH is hard in the same group. Certain group structures permit a relationship test even when computing a discrete logarithm is still believed difficult. The exact assumption must match the construction. [\[19\]](#)

The direction for hardness assumptions therefore goes the other way: if DDH is hard under the stated group and sampling model, then the attacks just described rule out efficient CDH and discrete-logarithm solvers there. If an efficient discrete-logarithm solver existed, it would undermine those Diffie–Hellman assumptions. Keeping track of the direction prevents a common reasoning error: showing that one powerful attack would break a system does not show that every successful attack has to use it. Later security arguments will identify the reduction they actually establish.

Factoring gives another important example of an easy operation with a potentially difficult reverse. Multiplying selected primes

p

and

q

to form

$$n = pq$$

is efficient. Recovering suitable large prime factors from

n

alone is the factoring problem used in later discussions. This differs from the discrete logarithm problem: the unknowns are factors of an ordinary integer, not an exponent in a selected group. It also differs from the RSA inversion problem, which asks for an appropriate root modulo an RSA modulus. Knowing the factorization helps solve RSA inversion under the usual setup, but we will not assume that every possible RSA inversion attack must first factor the modulus.

What does “hard” mean in these statements? It is a claim about the work needed to succeed, against a specified class of algorithms, on specified inputs. Cryptographic assumptions normally concern families of increasingly large instances with an agreed generation procedure. There are easy special inputs, weak parameter choices and situations in which partial information leaks. A claim that the general problem is believed difficult

does not mean that every individual instance is equally difficult or that no information about a secret can ever be learned. Both the distribution of inputs and the success criterion belong in the claim.

The group order also matters through its factors. When an order decomposes into small prime-power factors, discrete-logarithm methods can exploit that decomposition instead of treating the whole order as one indivisible obstacle. This is one reason for choosing an appropriate large prime-order subgroup rather than judging the modulus by its printed length. The representation matters too: attacks specialized to finite-field groups can differ from attacks on suitable elliptic-curve groups. We will return to the relevant choices in the key-agreement and elliptic-curve chapters, rather than turning the present examples into a parameter-selection guide.^[7]

Finally, the computational model can change the conclusion. The familiar hardness assumptions for factoring and discrete logarithms concern classical computation. Shor's efficient algorithms in a quantum-computing model change that landscape, which motivates the later post-quantum chapter.^[22] Separately, an attacker may bypass the intended mathematical problem by exploiting a leaked key, an unauthenticated exchange or an implementation error. An easy forward operation and a carefully stated reverse assumption are components of a security argument. They must be connected to the actual protocol and its environment before the argument protects a real application.

The foundations are now connected. Divisibility explains primes and common factors. The gcd decides when modular multiplication can be reversed, and the extended Euclidean calculation finds that reverse. Repeated squaring makes large powers efficient. Groups state which operations and reversals fit together, while their orders identify the cycles in the exponents. The hard-problem definitions specify what remains unknown to an observer. Chapter 5 adds the probability language needed to discuss how secrets are selected, how attacks are measured and what a security claim promises under its assumptions.

Questions to check your understanding

1. Why is trial division up to the square root enough to prove 97 prime? Does

$$2^{30} \equiv 1 \pmod{341}$$

prove 341 prime?

2. Use the Euclidean divisions to explain why 7 has an inverse modulo 26. Give the inverse and verify it by multiplication.

3. Why does

$$6x \equiv 1 \pmod{26}$$

have no solution, while

$$6x \equiv 4 \pmod{26}$$

has more than one? Give two solutions of the second equation.

4. Reproduce the calculation of

$$7^{13} \pmod{23}$$

using the powers with exponents 1, 4 and 8. Why can each intermediate product be reduced?

5. The full multiplicative group of nonzero residues modulo 11 has order 10. What is the order of its element 3, and what subgroup does 3 generate? Which modulus applies to its exponents?

6. Why do the invertible residues modulo 12 form a group that is not cyclic? Is zero one of its elements?

7. With

$$g = 3$$

modulo 11,

$$a = 2$$

and

$$b = 3$$

, give

A

,

B

and

$$g^{ab}$$

. Why is multiplying

A

by

B

a different calculation?

8. What distinguishes DLP, CDH and DDH? Does assuming DLP is hard automatically establish DDH hardness or authenticate a communicating peer?

5. Probability, Randomness and Security

Part II: The Mathematics We Need

A large collection of possible keys is useful only if we understand how a key is selected and what an attacker knows about it. A program may store a value in 128 bits while using a selection rule that produces only a few predictable possibilities. A generator may produce a long sequence that looks irregular while repeating the same sequence whenever it starts from the same state. To assess either case, we need more than a count of bit positions. We need a probability model, a description of the available information and a clear question about the attacker's task.

Probability does not mean that cryptographers leave correctness to chance. A decryption algorithm can recover its intended message exactly, while a security experiment measures an attacker's chance of guessing a secret or recognizing a hidden choice. Randomness can enter through key generation, a protocol's fresh inputs or the attacker's own decisions. A formula becomes useful when we can say which of those choices it describes and why the assumed distribution is appropriate. The same expression can be misleading when applied to a different experiment.

This chapter develops those distinctions through small examples. We will compare guesses with and without repetition, count collisions before using an approximation, and examine two different measures of uncertainty. We will then connect a real source of uncertainty to a deterministic generator and explain what a security proof establishes. These foundations will let us read the later cipher, authentication and public-key chapters without treating words such as “random,” “negligible” or “secure” as guarantees with no stated conditions.

5.1 Events, probabilities and independence

An **experiment** is a process whose possible outcomes and selection rule we describe. A fair coin gives heads and tails equal chances. Two coin tosses are **independent** when learning the result of one does not change the

chances for the other. Tossing two independent fair coins gives the four ordered outcomes HH, HT, TH and TT. H means heads and T means tails; the first position refers to the first coin. Fairness and independence make each ordered outcome have probability

$$1/4$$

. The **sample space**, written

Ω

, is the set of possible outcomes. An **event** is a subset of that space, such as the outcomes with at least one head.

When a finite sample space has equally likely outcomes, an event's probability is

$$(5.1)$$

The symbol

$\Pr$

means probability, and

E

is the event being considered. For at least one head, the favorable outcomes are HH, HT and TH, so the probability is

$$3/4$$

. Probabilities lie between 0 and 1. Multiplying a probability by 100 expresses it as a percentage:

$$3/4 = 0.75$$

means 75 percent. An event with probability 1 is certain under this model, not under every possible account of the physical process.

The counting formula requires equal likelihood. If a coin favors heads, the four two-coin outcomes need not have equal probabilities. A more general model assigns a nonnegative probability to each outcome, with total 1. An event's probability is the sum of the probabilities of its outcomes. Merely listing four possibilities does not establish a probability of

$$1/4$$

for each. The same issue appears in cryptography when an implementation can represent many keys but selects some much more often than others.

A **random variable** names a value determined by an outcome. Let X

be the number of heads in the two-coin experiment. The event

$$X = 1$$

contains HT and TH, so

$$P(X = 1) = 1/2$$

. The capital letter

X

names the variable before its value is known; a lowercase value such as

$$x = 1$$

is one possible result. A random variable can also be a bitstring or a group element. Its **distribution** describes the probabilities assigned to its possible values, rather than the visual appearance of one observed sample.

For an event

E

, its **complement**, written

$\overline{E}$

, contains the outcomes where

E

does not happen. Because an outcome either belongs to

E

or to its complement,

$$(5.2)$$

Here the bar means “not this event.” For at least one head, the complement is TT, with probability

$$1/4$$

, giving

$$1 - 1/4 = 3/4$$

. Counting the complement is often easier than listing every successful outcome. We will use this method to calculate the probability of at least one collision.

Learning information changes the experiment available to an observer. Suppose the two coins are fair and independent, but we are told that the first coin is H. Only HH and HT remain possible under that condition. The probability of HH is now

$$1/2$$

, rather than its original

$$1/4$$

. A **conditional probability** states this restricted question explicitly:

$$(5.3)$$

Read the vertical bar as “given that.” The event

$$E \cap F$$

means that both

$$E$$

and

$$F$$

happen. Dividing by

$$\Pr[F]$$

rescales the surviving outcomes so that their total probability is 1. For

$$E = \{HH\}$$

and

$$F =$$

first coin H, the calculation is

$$(1/4)/(1/2) = 1/2$$

. The condition must have positive probability for this formula to be defined.

Two events are **independent** if knowing that one occurred does not change the probability of the other, when the relevant conditional probability is defined. An equivalent definition is

$$(5.4)$$

For the independent coins, let

E

mean first coin H and

F

mean second coin H. Their individual probabilities are

$1/2$

, and both happen with probability

$1/4$

. That agrees with the product. Independence concerns the joint distribution. It cannot be established simply because two values were generated at different times or stored in different variables.

For comparison, generate one fair bit and copy it into two positions. Each position individually has an equal chance of being 0 or 1, but the two-position output can only be 00 or 11. The event that both positions are 1 has probability

$1/2$

, not

$1/4$

. The positions are correlated. This distinction matters for keys: knowing that every individual bit is balanced does not establish that all complete bitstrings are equally likely. For many variables, claims of mutual independence must cover their joint choices, rather than just isolated marginal probabilities.

Another useful rule is the **union bound**. If

$E_1, \dots, E_t$

are events, then

(5.5)

The union symbol

$\cup$

means “at least one of these events,” and the summation adds the

t

individual probabilities. The inequality is valid even if the events are dependent. An outcome lying in several events is counted several times on the right but only once on the left. Consequently, adding the probabilities gives an upper bound and does not always give an exact answer. If the sum exceeds 1, the trivial upper bound 1 is better. Later proofs use this rule to control the chance that any one of several unfavorable events happens.

Before applying a probability formula, therefore, identify the outcome, the distribution and the information being conditioned on. Independence and uniformity are assumptions that require support from the model or construction. They do not follow from the notation. Stating these assumptions is what turns a calculation about a toy experiment into a claim that can be checked against an actual cryptographic process.

5.2 Guessing a key

A **probability distribution** tells us how likely each possible value is. If a selection is **uniform**, all the allowed values have the same probability. For a uniform choice among 16 keys, each key has probability

$$1/16$$

. The size of the set tells us the number of candidates; the distribution tells us which candidates are likely. Keeping those questions separate will also help us understand why a long password can be less predictable than another long password only if their selection processes differ in the relevant way.

Suppose a secret key

K

is chosen uniformly from a nonempty set of

N

possible keys. An observer has no information favoring one key over another and is allowed one guess. Each candidate has probability

$$1/N$$

, so

The capital

K

names the selected secret, while

N

counts the allowed candidates. If every

k

-bit string is an equally likely key, then

$$N = 2^k$$

and this probability is

$$2^{-k}$$

. The negative exponent is a compact way to write

$$1 / 2^k$$

. The conclusion depends on the selection rule and the observer's information, not just on the storage length.

Now allow

q

distinct guesses against that one fixed secret, where

$$0 \leq q \leq N$$

. Assume that each attempt can be checked for exact key equality and that no other information or shortcut changes the search. The guesses cover

q

different candidates, so

$$(5.7)$$

For a four-bit uniform key there are

$$N = 16$$

possibilities. Four distinct guesses cover four of those possibilities, giving probability

$$4/16 = 1/4$$

. After a failed guess, that candidate is excluded; the remaining candidates stay equally likely in this model. The trials are therefore not independent

attempts with an unchanged conditional success probability, although the total success probability has the simple expression above.

A different experiment allows guesses chosen independently and uniformly *with replacement*, meaning that a previously tried candidate may be chosen again. Each guess misses a fixed key with probability

$$1 - 1/N$$

. The chance of missing on all

q

independent guesses is the product of those probabilities. Taking the complement gives

$$(5.8)$$

For

$$N = 16$$

and

$$q = 4$$

, this is

$$1 - (15 / 16)^4 = 14911 / 65536$$

,

about 22.75 percent. It is less than 25 percent because repeated guesses can waste attempts. Repeating one particular guess four times is different again: against the same unchanged key, it still covers only one candidate and succeeds with probability

.

The expected search position also needs a definition. If all

N

candidates are tested in a fixed order without repetition, a uniform secret is equally likely to appear at any position from 1 to

N

. The average position is

$$(5.9)$$

The left side sums the possible positions and divides by their number. For sixteen candidates the average is 8.5 tests. This is an average over independently selected secrets, not a promise that each individual search ends halfway through. It is also a count of candidate checks, rather than a measurement of seconds. A check's cost depends on the construction and the information available for verification.

This distinction is particularly important in attacks on real keys and passwords. A remote service may limit the number of attempts, whereas a stolen verification record can permit offline checks without contacting that service. One record may allow a candidate to be rejected cheaply, while another requires a costly derivation step. A guessed decryption key may produce plausible-looking data without establishing that it is the exact original key. Our equality-check model isolates the counting question; an actual attack analysis must specify its verification method and cost.

Uniformity must also survive the information given to the observer. Imagine a 128-bit stored key whose first 100 bits are fixed and publicly known, while the remaining 28 bits are chosen independently and uniformly. The observer needs to search only

2^{28}

candidates, not

2^{128}

. The other positions increase the string's length without increasing its uncertainty in this model. Encoding the key as more hexadecimal characters or bytes does not change the number of unknown choices. Chapter 3's distinction between data and their representation therefore matters directly to guessing probability.

Several targets can change an attacker's chances too. If there are M independently selected uniform keys from the same set, one fixed candidate matches at least one of them with probability

$$1 - (1 - 1/N)^M$$

. Here

$$M$$

counts targets, not attempts against one target. The union bound gives the upper bound

$$M/N$$

, capped by 1. Without independence, that upper bound can still apply when each target individually has matching probability

$$1/N$$

, but the product formula need not. A statement about one user's secret should not silently be substituted for a claim about success against any user in a large population.

Finally, exhaustive guessing is only one possible attack. A construction may have a mathematical weakness that recovers a key without testing all candidates, or an implementation may expose it directly. The earlier discrete-logarithm discussion already showed why printed bit length is not a universal measure of work. Counting candidates establishes the cost and success rate of a specified guessing strategy under specified information. It does not prove that the best possible attack has to follow that strategy.

5.3 Why collisions appear surprisingly early

A **collision** occurs when two selected values are equal. The task differs from guessing one fixed target: any matching pair can qualify. To see this before using a formula, draw three values independently and uniformly from the set

$$\{0, 1, 2, 3\}$$

, allowing repeated values. There are

$$4^3 = 64$$

equally likely ordered triples. A triple such as

$$(0, 0, 2)$$

has a collision, while

$$(0, 1, 2)$$

does not. The order of the draws is recorded, even though any pair of positions can supply the match.

Count the triples without a collision. The first position has four choices, the second has three choices different from the first, and the third has two choices different from both earlier values. There are

$$4 \times 3 \times 2 = 24$$

such triples. Therefore 40 of the 64 triples have at least one repeat, and

$$(5.10)$$

The probability exceeds one half even though there are only three draws. We did not assume that three separate pair-matching events were independent. We counted complete outcomes, which avoids that unjustified assumption.

The same reasoning works with

q

draws from a set of

N

values, where

$$N \geq 1$$

. Assume that the draws are mutually independent, uniform and made with replacement. Conditional on the first

i

draws all being different, the next draw avoids their

i

values with probability

$$(N - 1) / N$$

. Multiplying these conditional probabilities gives

(5.11)

The product symbol

$\prod$

means multiply one factor for each integer

i

from 0 through

$q - 1$

. The first factor is 1 because the first draw cannot repeat an earlier one. An empty product, for zero draws, is also 1, giving collision probability zero.

With more than

N

draws, a collision is certain because there are too few distinct values to assign a different one to every position.

For five draws from sixteen values, the no-repeat probability is

Subtracting from 1 gives collision probability

$4097 / 8192$

, about 50.01 percent. The denominators stay 16 because each new draw uses the original sixteen-value set. The decreasing numerators describe the condition that it avoids the values already seen. They do not describe a generator that removes values from its actual selection pool.

Why does a repeat arise so early? With

q

positions there are

$q(q - 1) / 2$

distinct pairs of positions. The numerator counts ordered choices of two different positions, and division by 2 avoids counting each pair twice. Each pair matches with probability

$1 / N$

. Applying the union bound gives

(5.12)

The function

$\min$

selects the smaller of the two displayed bounds. For five draws and sixteen values, there are ten pairs, giving bound

$10/16 = 0.625$

. The exact probability is smaller because some outcomes contain several matching pairs. The expected number of matching pairs is

$4(6 - 1)/(25)$

, but that expectation is not generally the probability that at least one pair matches.

For a large uniform set and a number of draws small relative to

N

, a useful approximation is

(5.13)

The symbol

$\approx$

means approximately equal. The function

$\exp(t)$

means

e^t

, where

e

is about 2.71828; its negative input here produces a value between 0 and 1. The approximation replaces the exact product by an exponential describing its combined decrease. It is especially useful around the square-root scale for large

N

. It is not a replacement for exact counting in a small example or for checking the assumptions of a real process. [\[7\]](#)

We can also explain the approximate halfway point. Set the right side of Equation [5.13](#) to

$$1/2$$

. The remaining exponential must then equal

$$1/2$$

, giving

$$e^{q(1/2) - k/2}$$

. Here

$$\ln$$

is the natural logarithm, the inverse of the exponential function, and

$$\ln 2$$

is about 0.6931. For a large draw count, replacing

$$q(q-1)$$

by

$$q^2$$

gives

$$(5.14)$$

The square root is the important growth scale; the factor 1.177 comes from

$$\sqrt{2 \ln 2}$$

. These are approximations for large sets, not exact thresholds for a finite experiment. Figure [5.1](#) uses exact calculations for

$$N = 256$$

. Nineteen draws give about 49.57 percent, and twenty give about 53.32 percent. At sixteen draws, which is exactly

$$\sqrt{256}$$

, the probability is about 38.03 percent. A square root is a scale, not an assertion that the probability is exactly one half there.

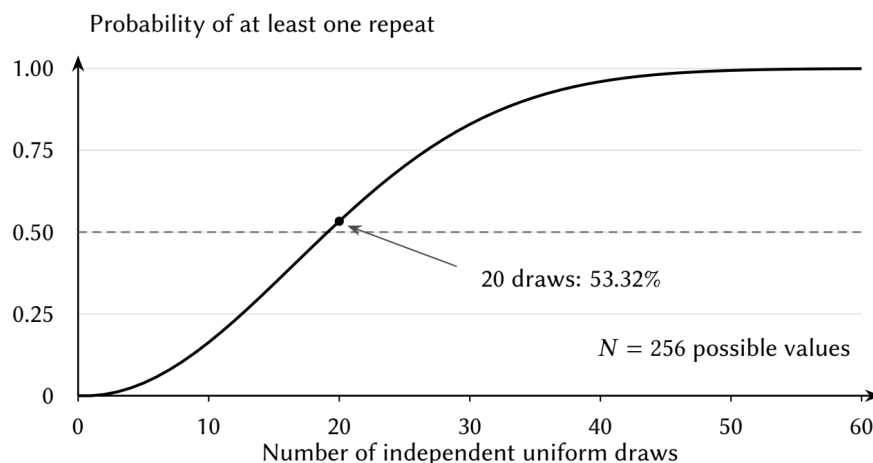


Figure 5.1. Exact probability of at least one repeated value in independent uniform draws from a set of 256 values. The horizontal guide marks probability one half. The curve concerns any matching pair, not recovery of a fixed secret. Its coordinates are calculated from Equation [5.11](#), without a simulation or fitted model.

This result explains the **birthday effect** in cryptography. For an idealized hash with

n

-bit outputs, independent uniform output behavior on distinct queried inputs leads to collisions on a scale of about

$2^{n/2}$

evaluations. Finding an input matching one fixed target output instead has the generic guessing scale

2^n

. These are different tasks. Repeating an identical input trivially repeats its hash, but does not give a collision between *distinct* inputs. The hash chapter will define the relevant security goals and examine where the idealized model is appropriate. The model also cannot be applied unchanged to outputs deliberately kept unique. A counter does not repeat within its allowed range before wrapping, and sampling without replacement prevents collisions within the selected set. Nonuniformity and dependence can change the probability as well. A collision calculation therefore begins with the selection rule and the event being counted. The formula's familiarity is not evidence that its assumptions fit every identifier, nonce, key or hash output in a system.

5.4 Entropy and predictability

In this chapter, **uncertainty** concerns what an observer can predict from the information they have. A key can already be fixed in a device while remaining unknown to that observer. Entropy measures describe aspects of a probability model for that uncertainty; they are not measurements of how confusing a printed string looks. We will compare a model that favors one value with a model that treats all values equally, then ask how well the best guess can perform.

We now need a way to describe uncertainty when possible values are not equally likely. Suppose a variable

X

can take four values with probabilities

$1/2$

,

$1/4$

,

$1/8$

and

$1/8$

. The best single guess is the first value and succeeds with probability

$1/2$

. Counting four possible values and assigning probability

$1/4$

would miss the bias. An **entropy measure** summarizes a specified feature of the distribution. Different measures answer different questions, so the name of the measure belongs beside its numeric value.

Shannon entropy describes average information uncertainty. For a finite random variable with

$p_i = \text{Pr}(X = x_i)$

, it is

(5.15)

The sum runs over values

x

with positive probability. The notation

$$\log_2(y)$$

asks which power of 2 equals

y

: for example,

$$\log_2(8) = 3$$

because

$$2^3 = 8$$

. Each term weights the value's information quantity

$$\log_2(1/p_i)$$

by its probability. Using base-two logarithms measures entropy in bits. The equivalent expression

$$-\sum_i p_i \log_2 p_i$$

is also common, with zero-probability terms defined to contribute zero. [\[2\]](#)

For four equally likely values, every probability is

$$1/4$$

. Each information quantity is

$$\log_2(4) = 2$$

, so the weighted sum is 2 bits. For the biased four-value example, the calculation is

$$(5.16)$$

The terms 1, 2 and 3 come from the reciprocals 2, 4 and 8. This is a weighted average, not the number of guesses required on every run. In particular, it does not change the best single-guess probability of

$$1/2$$

into

$$2^{-1.75}$$

. The distribution's most likely value remains its most likely value.

Min-entropy instead focuses directly on that strongest single guess. Let

$$p_{\max}$$

be the largest probability among the possible values. Then

(5.17)

The subscript ∞ is the conventional name for this entropy measure; it does not mean that the uncertainty is infinite. In the biased example, giving min-entropy 1 bit. For a uniform set of N values, both Shannon entropy and min-entropy are $\log_2 N$. NIST SP 800-90B uses min-entropy to assess uncertainty relevant to predicting a source’s output.^[24]

Table 5.1 compares distributions whose output lengths alone would give an incomplete picture. The last row might store either of its two possible outcomes as eight-bit strings, such as 00000000 and 11111111. Each position is individually balanced in that example, but the entire outcome is determined by one fair choice. Its uncertainty is 1 bit despite its eight-bit representation.

Table 5.1. Different distributions and the questions answered by two entropy measures. Entropies are in bits.

Distribution	Best guess	Shannon	Min-entropy
Four values, each with probability		2	2
Distribution	Best guess	Shannon	Min-entropy
Four values with probabilities		1.75	1
Distribution	Best guess	Shannon	Min-entropy
All 256 eight-bit strings, uniformly selected		8	8
Distribution	Best guess	Shannon	Min-entropy
Two specified eight-bit strings, each with probability		1	1

Uncertainty also depends on available information. A uniformly selected two-bit secret initially has four possibilities and min-entropy 2 bits. If an observer learns that its first bit is 0, the remaining possibilities are 00 and 01, equally likely under that condition. The best guess now succeeds with probability

$$1/2$$

. This is a statement about a particular conditional distribution. More general conditional entropy definitions must specify how they combine different observations. An unconditional entropy value should not be used as though an attacker had learned nothing else.

This limitation matters when estimating uncertainty from measurements. A long recorded sample can show patterns or support a statistical model, but it does not establish all the information an attacker may possess. Repeated samples might be dependent, a physical source may change behavior or an observer may influence part of its environment. An entropy assessment needs a source model, stated assumptions and appropriate validation. Counting zeroes and ones in one output file does not certify a key-generation system. SP 800-90B's source-design and testing requirements concern this wider problem, not simply a formula applied to a convenient sample.

Nor can deterministic processing create new information uncertainty from nothing. If an algorithm's only changing input is an

S

-bit seed, then at most

$$2^s$$

seed values can be supplied. For a fixed output length, a deterministic function of that seed can therefore produce at most

$$2^s$$

different outputs, even if every output is far longer than

S

bits. Two seeds might even lead to the same output, reducing the number further. Expanding the representation does not add new independent choices to the process. That observation does not make deterministic cryptographic generators pointless. Information-theoretic uncertainty and computational predictability are different questions. A computationally bounded observer

may be unable to recognize the restricted output set or recover the unknown seed without impractical work. A secure generator seeks that computational protection under its assumptions, while an entropy source provides the uncertainty on which the generator depends. The next section joins these roles without treating a long pseudorandom output as proof of equally many fresh bits of entropy.

Optional: reproducing the small calculations

Listing [5.1](#) checks the counting experiment and several later results. It enumerates all 64 ordered triples from four values, rather than relying on random sampling. Its second calculation uses rational arithmetic, which preserves fractions exactly. The entropy calculation uses logarithms and reports rounded decimal values. None of these operations generates a key or assesses an actual entropy source. [\[28\]](#)

Listing 5.1. Exact counting and public probability calculations.

```
from fractions import Fraction
from itertools import product
from math import log2

triples = list(product(range(4), repeat=3))
repeats = sum(len(set(t)) < len(t) for t in triples)
print(Fraction(repeats, len(triples)))

no_repeat = Fraction(1)
for used in range(5):
    no_repeat *= Fraction(16 - used, 16)
print(1 - no_repeat)
print(1 - Fraction(15, 16) ** 4)

probabilities = [Fraction(1, 2), Fraction(1, 4),
                 Fraction(1, 8), Fraction(1, 8)]
shannon = sum(p * log2(1 / p) for p in probabilities)
minimum = -log2(max(probabilities))
print(f"{shannon:.2f} {minimum:.2f}")
```

The expected output is

```
5/8
4097/8192
14911/65536
1.75 1.00
```

The first two lines concern collisions, the third concerns four with-replacement guesses against one fixed sixteen-value secret, and the fourth compares the two entropy measures. The different answers describe different experiments. The program checks that the stated counting and arithmetic agree with their expected results; it cannot establish that a real source follows the assumed distribution.

5.5 Random sources and deterministic generators

A cryptographic random-generation system must connect uncertainty in the world to the values consumed by software. A **noise source** supplies an entropy-providing process that is not fully determined by the observer's information. Sampling and digitization turn its observations into data. The observations may be biased or dependent, so the design must assess the behavior of the actual source instead of declaring every recorded bit an independent fair coin. Dedicated hardware and suitable system processes can play different roles; neither a physical label nor a timestamp alone establishes useful unpredictability.

An **entropy source** includes the noise process and the mechanisms needed to use it appropriately. These can include conditioning and health tests. A **conditioning function** processes raw samples to reduce bias or concentrate the available entropy into a suitable output. It does not create uncertainty absent from its inputs. **Health tests** watch for specified failures or unexpected behavior. They can detect important problems without being a complete proof that every future output is unpredictable. These component roles are described in SP 800-90B. [\[24\]](#)

A **deterministic random bit generator**, or DRBG, is different. It maintains an internal state and uses a defined algorithm to generate pseudorandom output and update that state. **Instantiation** initializes a particular generator instance from seed material and other specified inputs. A **seed** is the material used to establish its starting state; the full state may include additional working values and counters. If the algorithm, complete state and subsequent inputs are known, its future behavior is determined. Its security depends on protecting the relevant state and supplying the required uncertainty. [\[23\]](#)

Pseudorandomness is a computational goal. A generator can map a shorter secret seed to a longer output while aiming to make that output indistinguishable from an appropriate ideal random sequence to an efficient observer. An observer with unlimited computation could enumerate the possible seeds and compare their outputs. The practical claim instead bounds the observer's resources and identifies the generator's assumptions and limits. It is compatible with the earlier count of at most

2^s

possible fixed-length outputs from an

s

-bit seed. The count describes information; indistinguishability describes what a bounded observer can recognize.

There is also a difference between a mathematical pseudorandom generator used in a proof and a complete stateful generation service. A proof may define one deterministic mapping from a uniformly sampled seed to a fixed-length output. A deployed service has repeated requests, initialization, state updates, failures and possibly reseeding. Each layer needs its own conditions. Reusing a fixed public seed to make a simulation reproducible can be appropriate for that experiment, but it does not establish secrecy for generated credentials. A reproducible sequence and an unpredictable secret solve different problems.

Reseeding introduces new entropy input according to the construction and updates the generator's state. Simply resetting a counter or restarting from the same seed does not supply new uncertainty. Adding a publicly predictable timestamp also does not automatically repair exposure of the state. Recovery after compromise depends on new input that the attacker does not know, an appropriate update mechanism and the relevant model of access. Even a successful recovery of future generation does not make already exposed keys secret again.

Two related properties require care. **Backtracking resistance** concerns protecting earlier output after a later state exposure. **Prediction resistance** concerns protection of subsequent output in the face of relevant state compromise, with fresh entropy supplied as required by the construction. A state update that helps erase a link to the past is not automatically a source of future uncertainty. SP 800-90A describes the mechanism concepts and

requirements; whether a complete system supplies a property also depends on its entropy access and construction.^[23]

The NIST SP 800-90 documents address different layers. SP 800-90A Revision 1 specifies deterministic mechanisms. SP 800-90B addresses entropy sources and their assessment. SP 800-90C, published in its final form in September 2025, specifies constructions combining sources and deterministic mechanisms. Its constructions have different capabilities, including differences in access to entropy after initialization. The documents are related, but none of the three names is a synonym for the entire engineering task.^[25]

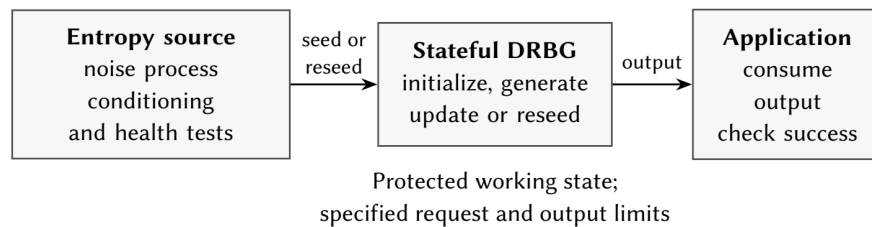


Figure 5.2. A conceptual route from an entropy source through a stateful generator to an application. Seed and reseed input carries assessed uncertainty; the generator maintains secret working state. Health tests belong to the source’s operation, while applications must also handle generation errors. This is an explanatory arrangement, not a complete diagram of every SP 800-90C construction.

Figure 5.2 separates those roles in one common arrangement. It does not imply that every construction has continuous access to fresh entropy or that each output bit contains an independent new physical observation. Some services generate many outputs between reseeds. The supported security strength, output limits and compromise properties depend on the chosen design. This chapter introduces the questions needed to understand such a design; it does not provide an entropy-source validation procedure or instructions for assembling an unreviewed generator.

A software interface also has a failure contract. OpenSSL’s `RAND_bytes` documentation describes a cryptographically secure generator and, for supported default configurations, automatic seeding from the operating system. It also requires callers to check whether generation succeeded; a failure is not a usable random result. The collected source and manual make this concrete: an application cannot obtain a security guarantee merely by calling a function whose name mentions randomness. It must use the

appropriate interface, honor its result and operate under the required configuration.^[27]

Statistical testing remains useful within that wider picture. A test can detect an obvious bias, repetition or a source failure covered by its design. A deterministic sequence selected by an attacker could nevertheless pass a limited battery of tests. Conversely, genuinely independent fair samples can sometimes look repetitive. Observing ten identical bits is not logically impossible under the fair-bit model; it has a specified probability. A test's false-alarm behavior, observation length and failure model therefore matter. Passing tests is evidence about the properties tested, not a replacement for the generator's cryptographic analysis.

The result is a chain of responsibilities. A source supplies assessed uncertainty, the construction combines and manages it, the deterministic mechanism protects and evolves its state, and the application consumes output correctly. Losing any one of these conditions can undermine the use made of the output. Calling a byte string random is the beginning of the explanation, not its conclusion. The final section gives a way to state precisely what the resulting system promises against a defined observer.

5.6 Security goals, assumptions and proofs

A **security goal** describes what an attacker should be unable to achieve. Recovering a key is one goal, but it is not the only relevant question. A system might hide the full message while exposing a sensitive yes-or-no fact about it. Another might keep data confidential while allowing an attacker to replace them. The goal therefore specifies the protected property, while the **attack model** specifies what the attacker can observe, request, modify or learn. Earlier chapters introduced these distinctions in words; probability lets us measure success in a defined experiment.

Goldwasser and Micali's work on probabilistic encryption made a central point precise: difficulty of reversing a mathematical function does not by itself establish that the ciphertext hides all useful partial information about the message. Their treatment connects a computational security definition to a stated hardness assumption and a proof.^[26] We will use that lesson throughout this book. A construction's output can be difficult to invert in

one sense yet still fail the particular confidentiality goal an application needs.

Consider a simplified message-choice experiment. The attacker chooses two messages

m_0

and

m_1

of the same length. A challenger generates a secret key according to the scheme, chooses an independent uniform bit

b

, and returns a valid encryption of

m_b

. The attacker examines the ciphertext and outputs a guess

b'

. The attacker wins if

$b' = b$

. The equal-length requirement prevents the experiment from being decided by length alone when length is intentionally visible. Any other public inputs, such as a nonce, must also be handled according to the scheme's specified rules. Without useful information about the challenger's bit, a guess succeeds with probability

$1/2$

. The experiment asks whether the ciphertext permits a meaningful improvement. Random choices can include key generation, encryption and the attacker's own behavior. Full security definitions specify whether more encryption queries, decryption queries or adaptive interactions are allowed. The small experiment here introduces the comparison; it does not replace the complete definition required for a particular mode or protocol. [\[7\]](#)

A related way to express advantage compares two experiments directly. Let an algorithm

D

output either 0 or 1 after receiving data from a real or ideal experiment. We define its **distinguishing advantage** by

(5.18)

The vertical bars take the absolute difference, so the direction of the output label does not matter. Each probability uses the random choices specified by its experiment. The advantage is between 0 and 1. If the two output probabilities are 0.60 and 0.50, the advantage is 0.10. This measures the gap between the experiments, not the probability that an entire application is compromised.

When the two experiments are chosen with equal probability and the decision labels are oriented in the better direction, that gap corresponds to classification success

$$1/2 + Adv/2$$

. With gap 0.10, the success probability is 0.55. Other texts define an advantage as the excess success probability above

$$1/2$$

, which would be 0.05 in this example. Both conventions can express the same comparison, but their numbers differ by a factor of two. A proof or claimed bound must identify which normalization it uses before values are compared.

We can now distinguish three kinds of protection. **Perfect secrecy** makes the relevant message and ciphertext distributions independent under its stated conditions, as Chapter 1 explained for a one-time pad. **Statistical security** allows a specified small difference detectable even by an observer without computational limits. **Computational security** limits the advantage of a specified resource-bounded observer under the stated assumptions. These are different claims. A computationally secure generator can have a small output support relative to the ideal space while still resisting efficient distinguishers.

A **security assumption** states a problem believed difficult under an explicit generation and resource model. Chapter 4 separated DLP, CDH and DDH because the requested outputs differ. Here probability adds the distribution of inputs and success criterion. Saying that a construction uses primes or has a long key does not identify such an assumption. Likewise, a

failure to find attacks in experiments is valuable evidence but not a mathematical proof that every relevant algorithm fails.

A **reduction** proves a relationship by converting a successful attack on one object into an attack on another. Suppose an attacker

$\mathcal{A}$

breaks a precisely defined scheme. The proof describes an algorithm

$\mathcal{B}$

that uses

$\mathcal{A}$

as a component to attack a specified underlying problem. One possible form of a derived bound is

(5.19)

The two advantages must use the definitions specified by the theorem. The nonnegative term

ϵ

accounts for a stated additional loss or bad-event probability. This displayed form is an illustration of how to read a bound, not a theorem about an unnamed encryption scheme. A real proof must identify the scheme, the problem, the conversion algorithm, its resource cost and every term in the bound.

For example, a proof may compare games that behave identically unless a sampled value repeats. The difference in the attacker's behavior can then be bounded by the probability of that bad event, using a justified collision calculation. Another proof may lose a factor because its constructed algorithm must guess which of several attacker actions will matter. Such losses affect the concrete result. If the reduction requires far more work or succeeds far less often than the original attacker, the assumption's numeric strength cannot simply be copied into the scheme's claimed strength. [7]

Security analyses often ask how an attack changes when we increase a size setting. Such a setting is called a **security parameter**; it describes a family of larger instances rather than one particular key. The term **asymptotic** means that we study this growth as the parameter becomes

large. This is different from estimating an attacker’s chance for one fixed deployment. The definition below makes one particular growth condition precise, and the example following it explains why merely tending toward zero is not enough.

The phrase **negligible probability** often has a specific asymptotic meaning. Let

λ

be a positive integer security parameter that controls a family of instances. A nonnegative function

$\varepsilon(\lambda)$

is negligible if it eventually becomes smaller than every inverse fixed positive power of

λ

:

(5.20)

The starting point for “sufficiently large” may depend on

c

. A function such as

$2^{-\lambda}$

is negligible, while

$1/\lambda^2$

is not: for

$c = 3$

, it remains larger than

$1/\lambda^3$

. Both tend toward zero, so tending to zero alone is insufficient. Asymptotic language describes growth across a family; deployment decisions also need concrete bounds at the actual chosen parameters.

Proofs remain conditional on their models. If a theorem assumes fresh independent randomness, using a repeated seed may place the implementation outside the theorem. If it models a black-box mathematical

operation, secret-dependent timing may reveal information that the model did not supply to the attacker. A confidentiality proof does not automatically establish authentication, availability or the truth of a signed statement. Mathematical analysis, implementation review and tests provide complementary evidence about different parts of the system.

The habit to carry forward is specific: identify the experiment, its random choices, the attacker's access and the measured outcome. Then examine the assumption, the proof and the gap between that model and the software using it. Probability makes the conditions visible instead of removing them. With these foundations in place, Chapter 6 begins secret-key encryption, using AES and ChaCha20 to show how carefully defined operations become practical constructions.

Questions to check your understanding

1. Two independent fair coins are tossed. What is the probability of HH before any information is given, and after learning that the first coin is H? Would copying one fair bit into two positions produce independent bits?
2. For one uniform sixteen-value secret, compare four distinct guesses, four independent uniform guesses with replacement, and four repetitions of one fixed guess.
3. How many of the 64 ordered triples drawn from four values contain no repeat? How does this give a collision probability different from guessing a fixed target?
4. Calculate Shannon entropy, min-entropy and the best single-guess probability for probabilities
0.125/0.125/0.125/0.125/0.125/0.125/0.125/0.125.
. Does an eight-bit representation by itself establish eight bits of entropy?
5. A deterministic function expands an eight-bit seed to a 64-bit output. What is the maximum number of possible outputs? Does that fact alone rule out computational pseudorandomness?
6. What different roles do an entropy source, a DRBG and an RBG construction play? Does passing health tests by itself prove full cryptographic security?
7. Why does resetting a generator or adding a predictable timestamp not automatically repair state exposure? Can reseeding restore the secrecy of keys already exposed?
8. A distinguisher outputs 1 with probabilities 0.60 and 0.50 in two experiments. What is its advantage under this chapter's convention, and its best-oriented classification success when the experiments are equally likely?
9. In the illustrative reduction bound, what must be known about $\mathcal{B}$ and

ε

before interpreting the result? Why is

$1/\lambda^2$

not a negligible function?

III. Working with Secret Keys

6. AES and ChaCha20

Part III: Working with Secret Keys

A protected message begins as ordinary data: a sequence of bytes representing text, a picture, a record or part of a network connection. The encryption software does not need to understand what those bytes mean. It needs a precisely defined transformation and the right secret key. The intended receiver must be able to recover the bytes, while an observer should learn no useful information about their contents beyond what the chosen construction permits. Turning this goal into a working system requires several layers. In this chapter, we examine two of its central building blocks rather than treating the name of a cipher as a complete security guarantee.

The two methods in this chapter are **AES**, short for Advanced Encryption Standard, and **ChaCha20**. Both are secret-key ciphers: the intended parties need suitable shared secret material. AES changes one fixed block of sixteen bytes into another block of sixteen bytes. ChaCha20 generates a sequence of bytes that is combined with the message using XOR. These are different ways to build an encryption process, not two names for the same calculation. We will first identify what enters and leaves each method, then look inside its steps. The representations from Chapter [3](#) will let us follow the bytes without needing to know a programming language.

Our aim is to understand what actually happens inside these ciphers without making a reader implement an encryption library. A library is a collection of ready-made software routines; here it provides the cipher operations that an application can call. We will build the small field used by AES, follow part of a published AES example and trace a complete ChaCha20 quarter round. Every displayed test key is public teaching data. The final section connects the internal operations to the larger choice of a construction, implementation and protocol. That boundary matters: learning how bytes are transformed is necessary for understanding encryption, but it

does not by itself explain how a receiver detects a changed or forged message.

6.1 What a secret-key cipher does

Suppose Nora and Sam already share an appropriate secret key. Nora wants to protect a message before it travels through a network that an observer can read. A **secret-key**, or **symmetric**, design lets the participants use shared secret material for the relevant encryption and decryption operations. The word symmetric describes the relationship between their secrets. It does not mean that every design runs exactly the same internal procedure in both directions. It also leaves open how the participants obtained the key and how they protect it while it is stored or used.

A **block cipher** gives one especially precise kind of transformation. For each allowed key

P

, it defines an invertible mapping of all blocks of a specified length to blocks of that same length. We can write one application as

(6.1)

Here

P

is one input block,

C

is its output block,

E

names the forward transformation and

D

names its inverse. The subscript

K

means that both use the chosen key. The equations concern the block-cipher primitive; an actual message-encryption construction may need additional

inputs and procedures. In particular, this notation does not assert that directly applying

E_K

to each part of a document is an adequate way to protect the document.

For a fixed key, the forward mapping is a **permutation**: it rearranges the possible block values without losing or duplicating any. Two distinct input blocks cannot produce the same output block under that fixed key, because a unique inverse would then be impossible. Equal input blocks, however, do produce equal output blocks. The primitive is deterministic once its key and input block are fixed. These facts explain both how exact recovery is possible and why a block cipher needs a carefully designed surrounding construction when messages contain repeated or related data. [\[13\]](#)

AES always uses a block of 128 bits, or sixteen bytes. Its three standardized key lengths are 128, 192 and 256 bits. The number in AES-256 names the key length; it does not turn the block into 256 bits. There are

2^{128}

possible AES input blocks, regardless of which of those key lengths is chosen. A 256-bit key has more possible bitstrings than a 128-bit key, but the probability discussion in Chapter [5](#) still applies: the way the key is selected and the attacks allowed in the model matter as well as its length.

It is useful to separate *correctness* from the intended security property. Invertibility says that the participant with the right key can recover an input block. A security model for a block cipher asks whether a feasible observer, given specified access to the transformation, can distinguish it from the appropriate ideal behavior. The ideal comparison is normally a randomly chosen permutation on the same block space, with its own consistent behavior on repeated inputs. It is not a new unrelated random answer on every call. This distinction prevents us from demanding a property that no deterministic permutation could satisfy. [\[7\]](#)

A **round** is one stage in a cipher's sequence of transformations. A **round key** is key material used at a particular stage, and the **key schedule** is the specified procedure for deriving those values from the original key. The schedule specifies which values are produced and where they are used. These terms let us describe a long calculation as a sequence of smaller

steps. We will inspect the actual AES steps below instead of expecting the round names alone to explain them.

The inverse algorithm and the key schedule are public too. An attacker is not supposed to be unable to read the instructions. The difficulty is intended to come from not knowing an appropriately selected key and from the absence of an effective way to defeat the stated security goal. Saying that a cipher has many rounds or looks complicated does not establish that difficulty. Published designs must be examined for mathematical shortcuts, and implementations must be examined for behavior that exposes secrets outside the abstract input-output model.

ChaCha20 has a different interface. An interface states which inputs an operation accepts and which outputs it returns. From a key, a nonce and a block counter, it produces a sequence called a **keystream**. A nonce is a value whose required uniqueness helps separate uses of the same key; it need not be secret. A counter selects a position within that use. The keystream is combined with the message using XOR. Regenerating the same keystream lets the receiver reverse the operation. Thus, the shared key remains central, but the complete inputs and the meaning of an internal block differ from AES. We will specify those differences before looking inside either design.

6.2 Blocks and streams

The phrase “a cipher processes blocks” can hide two different ideas. AES is a block cipher in the mathematical sense just defined: a keyed permutation of sixteen-byte inputs. ChaCha20 generates its stream in chunks of 64 bytes, and its specification calls the chunk-producing procedure a *block function*. That name does not make ChaCha20 a block cipher on 64-byte plaintexts. Its block function takes key material and position information, not a message chunk to permute. The message enters later, at the XOR step. [\[15\]](#)

Let
 M
be a message and let
 Z

be the portion of keystream with the same length. Stream encryption and recovery have the simple form

(6.2)

The symbol

$\oplus$

means position-by-position XOR. Each bit of

Z

is combined with the corresponding bit of

M

; applying that same bit of

Z

a second time cancels it. For a one-byte example, let the message byte be hex 41 and the matching keystream byte be hex B7. The ciphertext byte is F6, because 41 XOR B7 is F6. Recovery gives F6 XOR B7 = 41. These public values illustrate correctness only. They say nothing about whether an observer can predict an actual secret keystream.

This calculation resembles the one-time pad from Chapter [1](#), but the source of

Z

is different. A one-time pad requires an independent, uniformly random secret pad as long as the message, used once. ChaCha20 deterministically expands its finite inputs. Its intended protection is computational, under the relevant assumptions and use conditions. A long output does not create new information-theoretic entropy from a short key, and a random-looking output is not evidence of a fresh physical random event for each byte.

For the IETF form of ChaCha20 described here, write the keystream generation schematically as

(6.3)

The function

B

produces one 64-byte keystream block. Its inputs are the key

K

, nonce

N

and starting block counter

t

. The symbol

$\parallel$

joins byte sequences, and

prefix_ℓ

takes the first

ℓ

bytes, where

ℓ

is the message length in bytes. The displayed continuation stops when enough blocks have been generated and must remain within the permitted counter range. Joining two blocks is concatenation, not integer addition.

A 70-byte message, for example, needs one full 64-byte block and the first six bytes of the next block. The remaining 58 bytes from the second block are not extra ciphertext. At the raw stream-encryption level, the result is still 70 bytes. No padding is required merely to complete that last ChaCha20 chunk. A full protocol may add a tag, a nonce representation or other framing, so the size of a complete protected record can exceed the size of its plaintext. Confusing those layers can lead to incorrect expectations about file and packet lengths.

An AES-based construction can also generate a stream by applying the block cipher to carefully specified counter blocks. In other constructions, the AES input may instead depend on message blocks and earlier results.

These are decisions made by a **mode of operation**, rather than by changing the AES round itself. Different modes have different requirements for message lengths, initialization values and authentication. Chapter 7 will develop those requirements. For now, keep the basic distinction: AES supplies a fixed-block permutation, whereas ChaCha20 supplies a keyed stream-generation method with position information. Neither description automatically solves integrity. XOR recovery will still produce some bytes if a ciphertext byte has been changed. Likewise, a block-cipher inverse can map an arbitrary block to an output without establishing who created it. A complete protected-message interface must define acceptance and failure, not only produce plausible bytes. We can therefore study the internals of these ciphers while keeping a separate question in view: what additional construction will make their outputs safe for the receiver to use?

6.3 The AES state and round structure

AES organizes its sixteen input bytes as a four-by-four array called the **state**. This is a working arrangement of the data, not sixteen separate messages. The first four input bytes fill the first column from top to bottom; the next four fill the second column, and so on. The order is easy to misread if we expect a table to be filled across rows. Figure 6.1 makes the column order visible using the public input bytes 00 through 0F. All byte values in the AES examples below are hexadecimal unless another base is explicitly stated.

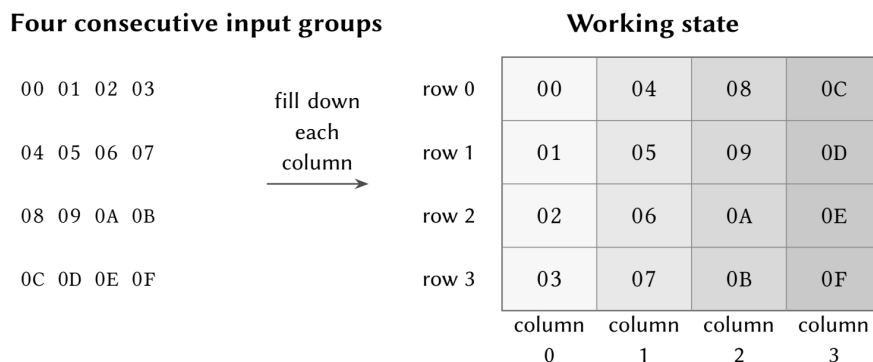


Figure 6.1. Sixteen sequential bytes arranged as an AES state. Each shaded column contains one consecutive group of four input bytes. The row labels and positions show the mapping; shading is only a reading aid. Output bytes are read back down columns in the same order.

With row and column indices starting at zero, the placement rule is

The symbol

$s_{r,c}$

means the byte at row

r

, column

c

. Multiplying

c

by 4 moves to the first byte of that column's input group, and adding

r

selects the position within it. Thus

$s_{2,1}$

is input byte number 6, which is 06 in the figure. The formula describes byte positions. It does not interpret the entire sixteen-byte sequence as one large integer in either endian order. [\[13\]](#)

The initial state is combined with a round key by **AddRoundKey**. Despite its name, the addition here is XOR of corresponding bytes, with no integer carries. The following rounds then apply four named operations. **SubBytes** replaces every byte according to the same invertible substitution rule. **ShiftRows** moves bytes within rows. **MixColumns** combines the four bytes in each column using a fixed field calculation. Finally, **AddRoundKey** combines the current state with that round's key material. Each operation has a particular purpose; repeating only one of them would not give the AES design.

SubBytes changes the relationship between input and output bits in a way that cannot be reproduced merely by XORing fixed selections of the input bits and adding a fixed constant. This property is part of what is meant here by its **nonlinear** behavior. The substitution is still precisely defined and reversible. ShiftRows moves bytes so that the next column-mixing step combines values that previously occupied different columns. MixColumns

then spreads their influence through each column, while the round keys bring in material derived from the secret. These descriptions explain the jobs of the steps; they do not replace analysis of the complete cipher.

The number of rounds depends on the key length, as shown in Table 6.1. AES-128 has an initial AddRoundKey, then nine rounds containing all four operations, and a tenth round containing SubBytes, ShiftRows and AddRoundKey. The last round omits MixColumns. AES-192 and AES-256 follow the same pattern with twelve and fourteen rounds respectively. The initial key addition is separate from the numbered rounds, which explains why the number of round-key blocks is one greater than the round count.

Table 6.1. The three AES variants. Every round-key block is 128 bits; the collection of round keys is derived from the original key.

Variant	Key bits	Block bits	Rounds / round-key blocks
AES-128	128	128	10 / 11
AES-192	192	128	12 / 13
AES-256	256	128	14 / 15

The **key schedule** derives the round keys from the original secret key. It is deterministic, using byte substitutions, movement within words, XOR and specified round constants. A constant is a value fixed by the algorithm rather than a fresh secret chosen for each message. The schedule does not independently generate eleven fresh secret keys for AES-128. Expanding 128 bits into 1,408 bits of round-key material does not create 1,408 bits of independent secret uncertainty. The expanded material depends on the same original input, and needs protection because it can support the cipher's operations just as the original key can.

The **S-box**, or substitution box, is a defined lookup rule that replaces one byte value by another. In AES the rule is invertible, so different input byte values have different output values. An implementation can calculate the rule rather than literally consult a stored box. Its full meaning will become clearer after the field explanation: the substitution uses a field inverse

followed by a fixed bit transformation. For now, the important point is that the S-box is a public rule used in both the cipher discussion and the key schedule.

For AES-128, the first four words of the schedule are the sixteen original key bytes, grouped four at a time. Later words depend on earlier ones. At particular positions, a word is rotated by one byte, its bytes pass through the S-box, and a round constant is combined with it before another XOR. The longer AES keys change the schedule's recurrence, not only the number of repetitions. In particular, AES-256 includes an additional SubWord step at specified positions. It would be incorrect to create AES-256 by taking an AES-128 schedule and simply extending its loop. We will inspect one AES-128 schedule step after learning the byte operations, rather than interrupting the main route with all expanded words.

A **matrix** is a rectangular array of values. Here it specifies how to combine the bytes of one column using the AES field rules; it is not an ordinary table of whole-number products. For fixed key material, every state operation has an inverse. SubBytes uses a bijective byte substitution, meaning a one-to-one replacement of all allowed byte values; ShiftRows can move rows back; the MixColumns matrix is invertible in the AES field; and applying the same round-key XOR a second time cancels it. AES decryption applies the appropriate inverse transformations in the required reverse order. It is not normally the same operation as running AES encryption again. Some modes use only the AES forward transformation even for message recovery, but that is a property of the mode, not a change to the definition of the block cipher's inverse.

An implementation may combine these conceptual steps, process several blocks together or use processor instructions. Its source need not display a loop over a drawn four-by-four array. That does not allow it to change the specified input-output result. The separation is useful when reading real code: the state and rounds describe the algorithm, while memory layouts and optimized routines describe ways of implementing it. Correctly locating a function in a repository is a starting point for checking behavior, rather than evidence that every implementation path has been audited.

6.4 The small finite field inside AES

A **field** is a system of values with addition and multiplication, where subtraction and division by every nonzero value are also defined. Its rules make these operations fit together. A **finite field** has only finitely many values, so its arithmetic must return to that allowed collection. AES uses a field with 256 values. It represents those values by bytes, but its multiplication is not ordinary integer multiplication followed by division by 256. We need the next representation to explain what its multiplication actually does.

A byte can be an unsigned integer between 0 and 255, but AES sometimes gives those same eight bits a different mathematical meaning. It reads them as coefficients of a polynomial whose coefficients are 0 or 1. A **polynomial** is an expression made by adding terms involving whole-number powers of a formal symbol, such as

$$x^2 + x + 1$$

. A **coefficient** is the number multiplying a term: the coefficients in this example are all 1. The rightmost bit gives the constant coefficient, the next bit gives the coefficient of

$$x$$

, and the leftmost bit gives the coefficient of

$$x^7$$

. Here

$$x$$

is a formal symbol used to describe arithmetic; we do not substitute a secret or a large ordinary number for it. This representation gives AES a field with

$$2^8 = 256$$

elements, written

$$\text{GF}(2^8)$$

. The letters GF mean Galois field, another name for a finite field; the expression

$$2^8$$

states that this one has 256 elements. [\[13\]](#)[\[29\]](#)

For example, hex 57 has bit pattern 01010111. The nonzero positions are 6, 4, 2, 1 and 0, so its polynomial is

(6.5)

The arrow connects two representations of one field element. It is not an equation saying that the ordinary integer 87 equals that polynomial for every value of

x

. The coefficient of

x^5

, for example, is zero, so that term is omitted. Leading zero bits remain part of the eight-bit representation even though their polynomial terms need not be printed.

Field addition adds matching coefficients modulo 2. Since

$$1 + 1 = 0$$

in this coefficient arithmetic, equal terms cancel. At the byte level, this is exactly XOR. For example, 57 plus 83 in the field is D4. Ordinary integer addition would give a different answer. Field subtraction is also XOR, because each element is its own additive inverse. None of this makes every AES operation an XOR: the field also has multiplication, and SubBytes uses an inverse followed by an additional transformation.

The **degree** of a nonzero polynomial is its highest power with a nonzero coefficient. For example,

$$x^6 + x + 1$$

has degree six. Multiplying two degree-at-most-seven polynomials may produce terms with powers higher than seven. To bring the result back to a byte, AES reduces by the fixed polynomial

(6.6)

The name

$m(x)$

denotes the polynomial modulus, not a message. Its highest power is eight. Its bit encoding is hex 11B, but the reduction is polynomial division with binary coefficients, not ordinary integer division by 283. The polynomial is irreducible over the two-element coefficient field: it cannot be factored there into lower-degree nonconstant factors. That property gives every

nonzero reduced element a multiplicative inverse, which ordinary arithmetic modulo 256 does not provide.

The simplest useful multiplication is by the element 02, whose polynomial is

$$x$$

. Multiplication by

$$x$$

moves every coefficient up one position. If the original high bit was zero, the resulting polynomial still fits in eight bits. If it was one, a term

$$x^8$$

appears. Modulo

$$m(x)$$

, we may replace that term with

$$x^4 + x^5 + x + 1$$

. This follows because adding those terms to

$$x^8$$

gives the modulus, whose remainder is zero. The subtraction needed to rearrange the relation equals addition in characteristic two.

Call this multiplication `xtime`. An implementation-oriented expression is

$$(6.7)$$

Here

$$b$$

is a byte,

$$b_7$$

is its high bit,

$$\ll 1$$

means a one-position left shift, and the bitwise AND with `FF` retains the low eight bits. The constant `1B` represents the lower-degree terms of the modulus after the departing

$$x^8$$

term has been accounted for. This equation uses integer bit operations to implement the field rule. Merely dropping the high bit, without the conditional XOR, would compute a different operation.

Follow four multiplications beginning with 57. First, shifting 57 gives AE, with no departing high bit. Next, AE has high bit 1. Its shifted nine-bit representation is 15C; polynomial reduction combines it with 11B, giving 047. Equivalently, retain 5C and XOR 1B to obtain 47. The next shift gives 8E without reduction. Finally, shifting 8E gives 11C; reducing with 11B gives 007. The eight-bit intermediate results are therefore

(6.8)

Each arrow is multiplication by the field element 02. The full nine-bit XORs above expose the reductions; the two conditional reductions are not ordinary integer subtractions. This is the multiplication example in Section 4.2 of FIPS 197, expanded here to show why the shortened byte rule works.

These steps also let us multiply by a more general byte. Hex 13 represents

$$x^4 + x + 1$$

, so distributivity gives

(6.9)

The bullet names AES field multiplication. Hex 10 represents

$$x^4$$

, so its product is obtained by the four `xtime` steps already shown. The other product requires one step, and multiplying by 01 leaves the element unchanged. Combining the three partial products uses XOR. This is how a polynomial product can be followed without writing every term of a long polynomial expansion.

The field inverse has the same purpose as the modular inverse in Chapter 4: it undoes multiplication by a nonzero element. Its calculation

uses the new field operations, however. In the AES representation, 53 has inverse CA , meaning that 53



$CA = 01$. One method uses the Euclidean algorithm on polynomials. Another uses the fact that the 255 nonzero field elements form a multiplicative group. Every nonzero

b

satisfies

$$b^{255} = 01$$

, so multiplying

b

by

$$b^{254}$$

gives the multiplicative identity. Therefore

$$b^{-1} = b^{254}$$

. The exponent means repeated field multiplication, which can be organized by successive squaring as in Chapter 4, now reducing polynomials at each step. It is not Python's ordinary integer power followed by a remainder modulo 256 or 283.

Zero has no multiplicative inverse in this field, just as it has none in the familiar rational numbers. AES handles zero by an explicit rule when constructing its S-box; it does not divide by zero. The field is a small, fully public algebraic system used to build an efficient reversible transformation. Its use does not make AES security depend on hiding its modulus or on the difficulty of computing these byte inverses. Indeed, we want those operations to be easy for both the implementation and the reader to check.

6.5 Following one AES transformation

We can now follow the byte operations within an AES round. The first is the S-box, which SubBytes applies independently at each state position. For a nonzero input

b

, AES first computes the field inverse

$$u = b^{-1}$$

. For a zero input, it sets the intermediate byte

u

to zero instead. It then applies an invertible affine transformation: a fixed combination of the bits, followed by addition of a fixed constant. An affine transformation can include a constant offset; the nonlinear part of this construction comes from the preceding inverse operation. [\[13\]](#)

Using eight-bit rotations to express the bit combination, the same S-box rule is

(6.10)

The name

S

means the byte substitution, and

$$\text{rot}_8(u, j)$$

rotates the eight bits of

u

left by

j

positions. Bits leaving the high end return at the low end. This is an eight-bit rotation, distinct from the 32-bit rotations used later in ChaCha20. All six displayed terms are combined with XOR, not ordinary addition. FIPS 197 states the affine step by bit indices; this rotation form gives the same resulting byte.

For input 53, the intermediate inverse is CA . Rotating that byte left by one, two, three and four positions gives 95, 2B, 56 and AC. The complete substitution is therefore

(6.11)

For zero, all the rotated intermediate values are zero, so

$S(00) = 63$

. The result is a substitution of every possible byte with a unique byte. A printed table of 256 outputs is one convenient representation of this rule, but the rule does not require a secret table or a freshly selected substitution alphabet for each message.

Next, consider the start of the AES-128 example in Appendix B of FIPS 197. It uses the input block 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34 and key 2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C. These are published test data. After the initial AddRoundKey, the first column is 19 3D E3 BE, because the four corresponding XORs are

(6.12)

The key bytes fill columns in the same order as the input bytes. The word “add” in the operation’s name has not changed these calculations into integer addition. At this point, each output byte depends on one input byte and one key byte; the later mixing operations will spread that dependence.

After SubBytes is applied to all sixteen bytes, the complete state and its ShiftRows result are

(6.13)

Row zero stays in place. Rows one, two and three rotate left by one, two and three byte positions respectively. For example, the second row begins 27 BF B4 41 and becomes BF B4 41 27; no byte value has been changed by that movement. The first column now contains D4 BF 5D 30, bringing together bytes from four different columns of the preceding state. This is why following only one column through SubBytes would be insufficient to compute the next column after ShiftRows.

MixColumns next multiplies each column by a fixed matrix. If a column’s input bytes, from top to bottom, are

(b_0, b_1, b_2, b_3)

, its output bytes

d_0, d_1, d_2, d_3

satisfy

(6.14)

This is matrix multiplication over the AES byte field. To obtain one output byte, multiply the four entries in its matrix row by the four column inputs and XOR those products. Multiplication uses



from the previous section, and field addition uses XOR. The matrix entries are hexadecimal field elements, not instructions to multiply ordinary integers and keep their last eight bits. Every output byte depends on all four input bytes.

For the example column, multiplying by 02 gives B3, 65, BA and 60, respectively. The first two require reduction: 1A8 XOR 11B is 0B3, and 17E XOR 11B is 065. The other two shifts remain within eight bits. Multiplication by 03 is the XOR of multiplication by 02 and the original byte, because 03 represents

$x + 1$

. The corresponding products are 67, DA, E7 and 50. Substituting these values into the four rows gives

(6.15)

Thus the transformed column is 04 66 81 E5. To check the first line more slowly, B3 XOR DA gives 69; XOR with 5D gives 34; XOR with 30 gives 04. The other three rows follow the same multiplication-and-XOR rule with their own coefficients.

For this published example, the first column of the next round key is A0 FA FE 17. Combining it with the MixColumns result gives A4 9C 7F F2. This completes the first numbered round for that column. The other columns undergo their corresponding calculations, and the following rounds

continue from the whole resulting state. After all ten rounds, including the different final round, the output block read down columns is 39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32. Matching that complete output and the intermediate values is stronger evidence of algorithmic correctness than checking only that encryption and decryption undo one another.

A closer look at one round-key word

Readers who want to connect A0 FA FE 17 to the original key can follow one additional step. The first and fourth words of that key are 2B 7E 15 16 and 09 CF 4F 3C. Rotate the fourth word left by one *byte* to obtain CF 4F 3C 09. Applying the AES S-box independently to these four bytes gives 8A 84 EB 01. XOR with the first round-constant word, 01 00 00 00, gives 8B 84 EB 01. Finally, XOR with the first key word gives A0 FA FE 17. The next schedule words are derived in turn; they are not selected as unrelated random inputs.

This small trace explains the origin of the round key without requiring a full key-schedule table. It also separates three operations that can look similar on a page: rotating a four-byte word by one byte in the key schedule, rotating the eight bits of a byte in the S-box expression, and rotating a 32-bit word by a specified number of bits in ChaCha20. Each has a width and a position rule. Replacing one by another can still produce plausible-looking bytes while computing the wrong algorithm.

6.6 Words, rotations and the ChaCha20 quarter round

ChaCha20 was designed by Daniel J. Bernstein. We use the IETF form described in RFC 8439: a 256-bit key, a 96-bit nonce and a 32-bit block counter. Earlier ChaCha variants use different arrangements, so the name alone is not enough to identify the expected inputs. In this form, the working state contains sixteen 32-bit words. The first four words are fixed public constants, the next eight hold the key, the next one is the counter, and the last three hold the nonce. The full state is therefore 512 bits, but only the eight key words are the secret key.^[15]

Within this state, the **quarter round** updates four selected words, conventionally named

a, b, c, d

. It uses three kinds of operations, often abbreviated **ARX**: addition, rotation and XOR. Addition here is ordinary integer addition modulo 2^{32}

, retaining the low 32 bits. It includes carries between bit positions; only a carry leaving the word is discarded. XOR has no carries. A rotation preserves every bit but changes its position. For example, `FFFFFFFF` plus `00000001` becomes `00000000` modulo 2^{32}

, whereas XOR of those two words gives `FFFFFFFFE`. These operations cannot be substituted for one another.

The complete quarter round is the following ordered calculation:

(6.16)

Every addition in this display is reduced modulo 2^{32}

. Read each row from left to right and the rows from top to bottom. The assignment arrow means “replace the current word with this result,” so the rotation on the first row uses the newly updated

a

. The notation

$\text{rot}_{32}(v, r)$

means a left rotation of the 32-bit word

v

by

r

bits. This ordered update is different from evaluating all right-hand sides using the four original words.

For a fixed word width, a rotation can be written using shifts:

(6.17)

Here

v

is an unsigned 32-bit value,

r

is the rotation distance,

$\gg$

is a logical right shift, and

|

is bitwise OR. The left shift supplies the moved lower bits; the right shift supplies the bits that wrap around. The final mask retains exactly 32 bits. For these valid distances the two retained parts occupy different positions, so the OR joins them. The vertical bar in this bit-operation formula is not the byte-concatenation symbol

||

used earlier.

Use the quarter-round test input from RFC 8439 Section 2.1.1:

(6.18)

The values are hexadecimal words, with leading zeros retained. First, adding

b

to

a

gives 12131415 . XOR of the old

d

with that new

a

gives 13305172. A left rotation by sixteen positions moves the low sixteen bits to the high half and the high sixteen bits to the low half, producing 51721330. This is a half-word exchange, not a reversal of individual bits.

Next, add that new

d

to

c

to get ECFF8273. XOR with the old

b

gives EDFD8177; rotating this value left by twelve positions gives D8177EDF. The remaining two rows repeat the pattern with the updated values and different rotation distances. Table 6.2 separates every addition, XOR and rotation so that each intermediate result can be checked. Only the named word changes in a row of the table; all other words keep their most recent values.

Table 6.2. One complete ChaCha20 quarter round
on the public test input. All values are
hexadecimal; each addition keeps its low 32 bits.

Step	Operation on the current words	New value
1		12131415
2		13305172
3		51721330
4		ECFF8273
5		EDFD8177
Step	Operation on the current words	New

p	words	value
6		D8177EDF
Step	Operation on the current	New
p	words	value
7		EA2A92F4
Step	Operation on the current	New
p	words	value
8		BB5881C4
Step	Operation on the current	New
p	words	value
9		5881C4BB
Step	Operation on the current	New
p	words	value
10		4581472E
Step	Operation on the current	New
p	words	value
11		9D9639F1
Step	Operation on the current	New
p	words	value
12		CB1CF8CE

Step 10 is a useful place to see a discarded carry. The full integer sum $ECFF8273 + 5881C4BB$ is 14581472E. Its low eight hex digits are 4581472E; the leading carry is discarded by reduction modulo 2^{32}

. Carries within those retained positions still contributed to the result. The final words, in order

a, b, c, d

, are EA2A92F4, CB1CF8CE, 4581472E and 5881C4BB, matching the published vector.

The following optional Python example implements only this small public-input calculation. It uses explicit masks because Python integers can grow beyond 32 bits. The XOR and rotation share one assignment in each pair, while the preceding addition is performed first. The program has no key-generation interface and is not an implementation for protecting messages. Its purpose is to connect the displayed operations to reproducible intermediate arithmetic. [\[16\]](#)

Listing 6.1. Checking the published quarter-round result.

```

MASK = (1 << 32) - 1

def rotl32(value, distance):
    return ((value << distance) |
            (value >> (32 - distance))) & MASK

def quarter_round(a, b, c, d):
    a = (a + b) & MASK
    d = rotl32(d ^ a, 16)
    c = (c + d) & MASK
    b = rotl32(b ^ c, 12)
    a = (a + b) & MASK
    d = rotl32(d ^ a, 8)
    c = (c + d) & MASK
    b = rotl32(b ^ c, 7)
    return a, b, c, d

result = quarter_round(0x11111111, 0x01020304,
                       0x9B8D6F43, 0x01234567)
expected = (0xEA2A92F4, 0xCB1CF8CE,
            0x4581472E, 0x5881C4BB)
assert result == expected
print(" ".join(f"{word:08x}" for word in result))

```

The printed line is `ea2a92f4 cb1cf8ce 4581472e 5881c4bb`. Uppercase and lowercase hex digits represent the same values. The assertion checks the fixed result against the specification, while the table exposes the steps behind it. Neither check establishes the security of a full cipher or the side-channel behavior of a Python implementation. In particular, this small routine omits the sixteen-word state setup, the remaining quarter rounds, the final addition, serialization, counter handling and message processing.

From four words to a keystream block

Number the sixteen state words from 0 to 15 and display them in four rows, filled across each row. A column round applies the quarter round to word positions

(0, 4, 8, 12)

,

(1, 5, 9, 13)

,

(2, 6, 10, 14)

and

(3, 7, 11, 15)

. The next round uses the diagonals

(0, 5, 10, 15)

,

(1, 6, 11, 12)

,

(2, 7, 8, 13)

and

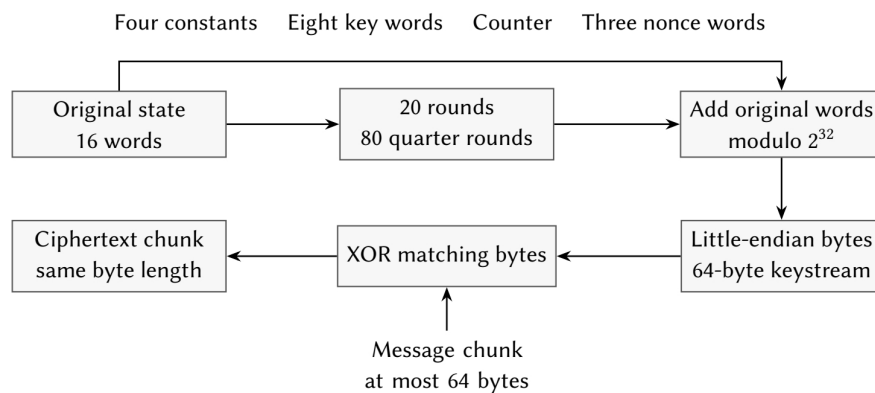
(3, 4, 9, 14)

. The wrap in these diagonals connects positions across the sides of the drawn array. It changes which words interact, rather than changing the quarter-round rule itself. Unlike the AES placement diagram, this displayed ChaCha arrangement advances the word indices across rows.

One column round and one diagonal round form a double round. ChaCha20 performs ten double rounds, hence twenty rounds or eighty quarter rounds. The number 20 therefore does not count individual additions, and one quarter-round test does not test all of ChaCha20. After the twenty rounds, the algorithm adds each original state word to the corresponding transformed word modulo

2^{32}

. This final step is often called **feed-forward**. It uses word addition, not XOR. The resulting sixteen words are then serialized into 64 bytes in little-endian order. Figure 6.2 separates those stages and shows where the message enters.



For the next chunk: keep the key and nonce, advance the counter without wraparound.

Figure 6.2. The route from ChaCha20 inputs to one ciphertext chunk. The original state is retained for the final word additions. A message chunk enters only at the XOR stage, and uses just as many keystream bytes as its length requires. This diagram describes the cipher, without an authentication mechanism.

Byte order matters at both ends. The first four key bytes 00 01 02 03, when present in a test key, form the word 03020100. After the final additions, a result word E4E7F110 produces the four output bytes 10 F1 E7 E4. No bits are reversed inside a byte. The apparent reversal of printed hex pairs comes from writing an integer with its most significant digit first while serializing its least significant byte first.

The full block example in RFC 8439 Section 2.3.2 makes these stages checkable. It uses the 32 consecutive key bytes 00 through 1F, nonce bytes 00 00 00 09 00 00 00 4A 00 00 00 00, and counter 1. Its first state word is the public constant 61707865. After twenty rounds, the corresponding transformed word is 837778AB. Their sum is E4E7F110, so the first four output bytes are exactly those just shown. The first sixteen serialized bytes are 10 F1 E7 E4 D1 3B 59 15 50 0F DD 1F A3 20 71 C4. These bytes are a test keystream, not ciphertext until they have been combined with a message.

The block counter selects successive chunks, but it cannot wrap around and reuse an earlier position under the same key and nonce. Starting at counter

t

, where

$$0 \leq t < 2^{32}$$

, the remaining counter space permits at most

$$(6.19)$$

of raw stream output before wraparound. The factor 64 is the bytes per block. Starting at zero gives

$$2^{38}$$

bytes, or 256 GiB, as a counter-space bound; starting at one leaves one fewer block. A complete construction may reserve blocks or impose additional limits, so this calculation is not universal permission to encrypt that much data in every protocol. The IETF authenticated construction reserves block zero for a different purpose. Managing nonces, counters and message acceptance is the subject of the next chapter.

The collected OpenSSL source provides a useful connection to an implementation. In `crypto/chacha/chacha_enc.c`, the quarter-round macro has the same addition, XOR and rotation pattern. The core applies the column and diagonal groups, adds the saved input words and produces little-endian output. These selected passages make the specification recognizable in real code. A library can also select architecture-specific implementations, so this reading neither identifies every path executed on a machine nor substitutes for a full implementation audit. [\[30\]](#)

6.7 Choosing a cipher within a complete construction

After following both algorithms, it is tempting to ask which one is simply better. The useful question is more specific: which complete, supported construction meets the needs of a particular application on its actual platform? AES can benefit greatly from dedicated processor instructions. ChaCha20's word operations make efficient implementations possible on many platforms without AES-specific instructions. A speed comparison depends on hardware, software, message sizes and the complete operation being measured. A performance number from one historical machine does not establish the fastest choice for every present-day device. [\[15\]](#)

Implementation behavior also matters. A substitution table indexed by secret-dependent values can cause different memory accesses, potentially exposing information through cache and timing observations. AES need not be implemented through such a table: specialized instructions and suitable constant-time software provide different routes. ChaCha20 avoids secret-indexed substitution tables in its round design, but that fact does not make every program using it immune to every side channel. Correct word operations, compiler behavior, memory handling and the surrounding application still need attention. The broader implementation chapter will examine these questions separately from the cipher's abstract mathematical behavior.

The selected OpenSSL AES source illustrates why this distinction is practical. It contains different implementation branches, including a constant-time C path and a table-based path, with architecture-specific code elsewhere in the project. Some routines combine steps that we have

explained separately. Reading one branch does not establish which branch a deployed application selects, and matching a vector does not establish that secrets cannot leak through execution. The repository's low-level AES interfaces are also distinct from its higher-level encryption interfaces. Understanding the core is useful evidence about the algorithm, not a reason to build a new application around an internal routine. [\[30\]](#)

Most importantly, cipher selection must be tied to the intended protection goals. A system that needs to detect modification should use an appropriate authenticated-encryption construction, with its key, nonce, length and verification rules. AES-GCM and ChaCha20-Poly1305 are names for such complete constructions; they are not alternate spellings of AES and ChaCha20 alone. Their authentication mechanisms add a different kind of evidence about a protected message. Even successful authentication does not by itself establish freshness, the real-world identity of a peer or the correctness of data before it was protected. Those remain protocol and application questions.

We can now recognize a meaningful explanation of a cipher. It states the input widths, transformations, order of updates and output representation. It distinguishes key material from public position information and from the data being protected. It checks published outputs as well as recovery, and keeps those correctness checks separate from the security claim. AES gives us a keyed block permutation built from byte operations; ChaCha20 gives us a keystream built from a larger word state. The next step is to understand how constructions use these tools without exposing repeated positions, accepting unauthorized changes or releasing data before the required verification succeeds.

Questions to check your understanding

1. Does AES-256 use 256-bit plaintext blocks? How many round-key blocks does it derive, and are those keys independent random choices?
2. Why is the 64-byte ChaCha20 block function different from a block cipher on 64-byte messages? How many keystream blocks are needed for a 70-byte message?
3. In Figure [6.1](#), which input byte occupies row 2, column 1? Why would filling across rows change the calculation?
4. Show the reduction in
xtime(AE)
. Why would ordinary integer multiplication modulo 256 give a different answer?
5. Use the displayed partial products to check
57 • 13
. What does AES do with zero before the S-box affine step?
6. In the worked AES round, why does the first column after ShiftRows contain D4 BF 5D 30, rather than the first column immediately after SubBytes? What is its MixColumns result?
7. At step 10 of the ChaCha20 trace, why does the result begin 45, although the full integer sum begins 145? Does this mean addition was XOR?
8. How many quarter rounds occur in ChaCha20? What operation combines the transformed state with the original state, and how is E4E7F110 serialized?
9. A raw ChaCha20 program produces the correct test keystream. What has that checked, and what does it leave unestablished about a complete protected-message system?

7. Nonces and Authenticated Encryption

Part III: Working with Secret Keys

An encryption algorithm can produce the right bytes and still be used in a system that exposes messages or accepts unauthorized changes. A sender might restart a counter after a crash while keeping the same key. A receiver might act on decrypted data before checking whether they are authentic. A message header might remain readable, as the network requires, while its meaning is left unprotected. None of these problems requires an attacker to find the secret key by exhaustive search. They arise where a cipher's mathematical transformation meets the rules for using it.

The previous chapter explained AES and ChaCha20 as building blocks. This chapter explains how constructions use such blocks to protect messages with both confidentiality and integrity. Two ideas connect the discussion. First, an encryption operation needs the right context, including a nonce or initialization value with the properties required by its scheme. Second, a receiver needs a defined acceptance rule. Producing candidate plaintext is different from establishing that the protected record should be accepted. A **record** here is one unit protected and checked together, such as one message with its header, nonce and tag.

We will follow these ideas from a short repeated-keystream calculation to two complete message-protection constructions. **AES-GCM** uses AES with Galois/Counter Mode; **ChaCha20-Poly1305** combines ChaCha20 with the Poly1305 authentication calculation. The mathematics will show what a particular failure reveals and how the authentication calculation binds data together. An optional library example will accept a valid record and reject several changes. Throughout, we will distinguish requirements on one message, requirements across a key's lifetime, and decisions made by the application. A secure construction addresses a precise set of questions; it does not remove the need to ask the remaining ones.

7.1 Why a block cipher alone is not enough

Imagine a document divided into sixteen-byte blocks. A tempting method is to encrypt each block independently with AES under the same key, joining the results afterwards. This is the basic idea of Electronic Codebook mode, or ECB. A fixed-key AES permutation maps equal input blocks to equal output blocks. Consequently, repeated plaintext blocks remain recognizable as repetitions in the ciphertext, even when the observer cannot recover their values. The observer learns a relationship that a confidentiality construction is normally intended to hide. Using a longer AES key does not change this deterministic equality property.^[35]

The problem is visible without drawing a recognizable picture. Let a three-block input be

$P_1 || P_2 || P_1$
, where
 P_1
and
 P_2
are distinct blocks and

$||$

joins byte sequences. Independent fixed-key encryption produces

(7.1)

Here
 E_K
is the AES forward transformation with key
 K

. The first and third output blocks are identical because their inputs and key are identical. Distinct inputs cannot collide under that fixed permutation, so the middle block is different. An observer has learned the repetition pattern exactly. This does not prove that every byte of the document is recoverable; it shows a particular kind of information leakage.

A mode of operation specifies how a block cipher is used beyond a single block. It may combine message blocks with earlier results, or use the

cipher to generate a stream from counter blocks. Its definition must cover the inputs, processing rules and permitted message lengths. A mode can improve confidentiality while still providing no authentication. The word *mode* alone therefore does not answer the receiver's question: how can I recognize a changed or invented protected message?

For a raw XOR-based stream construction, the integrity problem is especially direct. If

$$C = M \oplus Z$$

and an attacker changes the ciphertext by a chosen bit pattern

$$\Delta$$

, the receiver's raw recovery gives

$$(7.2)$$

The symbols

$$M$$

,

$$C$$

and

$$Z$$

are equal-length message, ciphertext and keystream portions. The symbol

$$\Delta$$

names the changes, with a 1 at each bit position to flip. XOR is associative, and the two copies of

$$Z$$

cancel. The attacker can therefore cause matching plaintext-bit changes without learning the keystream. Knowing the message format may make a particular change meaningful. Without that knowledge, the attacker can still corrupt it.

This property is called **malleability**: an observer can turn a protected value into another one whose recovered data have a controlled relation to the original. It is not the same as recovering the encryption key. An authentication mechanism is meant to make an unauthorized modification

fail the acceptance check, rather than merely turn it into altered plaintext. AES-GCM and ChaCha20-Poly1305 combine stream-style encryption with such a mechanism, but that mechanism has its own assumptions and input rules. [\[31\]\[15\]](#)

Adding familiar-looking structure is not a substitute for a construction. A receiver might recognize a file marker, a plausible date or a repeated word in decrypted bytes, but those features are not secret evidence that an authorized participant created the record. Similarly, taking an encryption algorithm and an authentication algorithm that are individually well studied does not make every way of joining them secure. The order of operations, covered data, key separation and failure behavior need an argument. We will use specified authenticated constructions rather than inventing a rule from the appearance of their outputs.

7.2 Nonces, IVs and counters

A **nonce** is an extra input used to distinguish encryption operations, usually without being kept secret. An **invocation** means one call of the encryption procedure. For the constructions studied here, a nonce has to be different for each invocation under the same key. For the AES-GCM profile and IETF ChaCha20-Poly1305 construction used in this chapter, the sender must avoid using the same nonce again with the same key. That rule concerns encryption invocations, not attempts to decrypt a received record. A receiver necessarily supplies the original nonce to recover the message and may need to examine a record more than once. The sender's uniqueness obligation does not mean that a nonce must disappear after the receiver has seen it. [\[34\]](#)

The phrase *with the same key* defines the scope. Two different, appropriately generated keys can use the same nonce value without thereby repeating the same key-and-nonce pair. In contrast, two devices that share one encryption key do not obtain separate nonce spaces merely because their processors or files are different. Their combined encryption activity must respect the key's requirements. The same issue arises when one key is shared across both directions of a conversation, across several workers or across a restored copy of an application. A nonce is not generally a second secret key. It can be sent with the ciphertext, stored beside it or

reconstructed from agreed information. Predictability is also different from repetition. A correctly managed sequence of nonce values can be predictable and still unique. A random choice can be hard to predict and still collide with an earlier choice. Keeping a nonce hidden does not repair repetition of a keystream or an authentication key derived from it.

An **initialization vector**, or IV, is an input used to initialize a mode's processing. GCM uses the term IV for an input that plays the nonce role. Other modes have different conditions. For example, Cipher Block Chaining (CBC), a confidentiality mode, requires an unpredictable IV, rather than allowing an unchanged public sequence number to be used directly as its IV. NIST SP 800-38A distinguishes this condition from the uniqueness requirement for other modes. A rule for GCM nonces must not be transferred to CBC merely because both inputs are called IVs. This comparison explains the terminology; it is not a recommendation to add unauthenticated CBC to a new application. [\[35\]](#)

A **counter** is one way to track positions. There may be more than one counter in a construction, and they need not have the same width or purpose. A message counter may help form a new nonce for each record. A block counter then selects successive keystream blocks *within* that record. Advancing the internal block counter does not allocate a safe nonce for the next record, and allocating a new record number does not permit the internal counter to wrap during a message.

One possible nonce layout combines a sender field and a record counter:

(7.3)

Here

S

identifies an encryption context assigned a distinct 32-bit value under the key, and

j

is that context's 64-bit invocation counter. The function

enc_w

writes a nonnegative integer that fits in exactly

w

bits, using an agreed byte order; we use big-endian order in this illustration. Thus

$$0 \leq s < 2^{32}$$

and

$$0 \leq j < 2^{64}$$

, and the counter must not wrap or restart while its key and sender field are retained. The joined result has

$$32 + 64 = 96$$

bits, or twelve bytes. The layout resembles the deterministic construction described by NIST; the formula alone does not allocate identifiers, preserve state or choose a permitted lifetime for the key. [\[31\]](#)

For example, sender value 7 and counter 258 give the nonce bytes 00 00 00 07 00 00 00 00 00 00 01 02. The first four bytes encode 7 and the last eight encode

$$258 = 256 + 2$$

. A second sender could use its own assigned sender value with counter 258, giving a different complete nonce. Reusing sender value 7 and counter 258 under the same key would recreate the original pair. Randomly picking a short sender identifier without collision control would not establish that the assignment is unique.

Persistence and concurrency are part of this rule. A sender that writes its updated counter only after transmitting can lose that update in a crash. Restoring an old backup can restore an old counter and an old key together. Two workers can read the same next value before either records its use. A counter design therefore needs a way to reserve values safely, divide ranges or change the key when continuity cannot be guaranteed. The relevant requirement is never to allocate an already used pair, not to make the counter display a convenient consecutive sequence with no gaps. Skipping a reserved value is generally easier to handle than silently using it twice.

Random nonces replace some state-management work with a collision probability that must be controlled. For

q

independent uniform selections of
 n
 bits, the union bound from Chapter 5 gives

$$(7.4)$$

There are

$$q(q-1)/2$$

pairs of selections, and each pair matches with probability

$$1/2^n$$

. Adding those probabilities gives the bound; it does not claim that all pair-match events are independent. When the right-hand side exceeds 1, it is not a useful probability estimate and the bound 1 is better. The equation assumes independent uniform sampling, which a repeating or badly seeded generator may fail to provide.

For

$$n = 96$$

and

$$q = 2^{32}$$

, the displayed bound is

$$2^{-33} - 2^{-64}$$

, just below

$$2^{-33}$$

. It is small, but not zero. This calculation is not a recommendation to send
 2^{32}

messages with every 96-bit-nonce system. A scheme or protocol can impose stricter limits because of its authentication bounds, total processed data, number of users or generation method. In particular, the finalized GCM specification has specific invocation constraints tied to its IV constructions. A nonce's bit length is only one input to a complete usage policy.

A timestamp alone is usually not a convincing uniqueness mechanism. Several messages may be created in one clock interval; a clock may move backwards; and several senders may see the same time. Combining fields can be useful when the construction and allocation rules are explicit.

Simply adding more labels, or assuming that events happen at different real times, does not prove that the resulting encoded byte strings will be distinct.

7.3 What nonce reuse reveals

Return to a stream construction whose keystream positions are determined by the key, nonce and internal counter. If two encryption operations repeat those inputs for overlapping positions, they repeat the same keystream bytes. Let the two message portions be

M_1

and

M_2

, and let the reused keystream be

Z

. Their ciphertext portions satisfy

(7.5)

The subscripts distinguish the two messages, and all portions in the equation have the same length. Rearranging XOR pairs the two identical copies of

Z

, whose XOR is zero. The ciphertexts therefore reveal the XOR of the plaintext portions without requiring the observer to know the key. For unequal message lengths, the equation applies to the overlapping reused positions, not to nonexistent bytes beyond the shorter portion.

Table [7.1](#) shows a seven-byte teaching example. The common keystream is 10 32 54 76 98 BA DC; it is a public demonstration mask, not an output claimed for a particular cipher key. The two ASCII messages differ in one digit. Their ciphertexts differ at the matching position, even though most ciphertext values do not look like the original letters.

Table 7.1. Two messages combined with the same seven keystream bytes. This deliberately incorrect reuse illustrates a relation exposed by the ciphertexts.

Value	Bytes in hexadecimal
	50 41 59 3D 31 30 30
: PAY=100	
	50 41 59 3D 39 30 30
: PAY=900	
	40 73 0D 4B A9 8A EC
	40 73 0D 4B A1 8A EC
	00 00 00 00 08 00 00

At byte index 4, counting from zero, the ASCII values for the digits 1 and 9 are 31 and 39. Their XOR is 08, which is also A9 XOR A1 in the corresponding ciphertext bytes. The remaining positions cancel to zero because the messages have equal bytes there. The observer has learned where the messages differ and the XOR relation at that position. That alone need not identify both messages uniquely; many pairs of bytes can have XOR 08.

If the observer knows one message portion, the other can be recovered exactly:

$$(7.6)$$

The right-hand side uses the known portion
 M_1
and two observable ciphertext portions. Substituting the relation from Equation [7.5](#) cancels
 M_1
and leaves
 M_2
. Alternatively,

$$M_1 \oplus C_1$$

first recovers the reused keystream portion, which can then be combined with

$$C_2$$

. A predictable file header, a known request format or information from another channel can therefore turn an exposed relation into recovered content.

Raw stream encryption is also vulnerable to the controlled modification discussed earlier. In this example, toggling ciphertext byte A_9 by 08 changes the recovered digit from 1 to 9. That manipulation does not require a second ciphertext; it follows from the XOR construction's lack of authentication. Reuse and malleability are different failures, although both can be understood through XOR cancellation. A correct authentication mechanism is meant to reject the changed ciphertext, but only within its own use conditions.

AES-GCM nonce reuse can undermine both confidentiality and authentication. With the same key and 96-bit nonce, the message counter stream repeats. The masking value used in the authentication tag also repeats, exposing relations involving the internal hash subkey. Suitable observations can enable forgeries, even when the attacker has not recovered the AES key. Precisely what can be recovered from a particular transcript depends on the messages, associated data, lengths and tag information available. It would be inaccurate to claim that every repeated pair immediately reveals the AES key or all messages ever protected with it. [\[31\]](#) In ChaCha20-Poly1305, repeating the key and nonce repeats both the message keystream and the derived Poly1305 one-time key. The authenticator was not designed to reuse that one-time key for different protected inputs. Repetition therefore removes assumptions needed for its protection as well as exposing the stream relation. Changing only the associated data does not repair either construction's repeated keystream: associated data are not an extra secret key or an independent source of stream positions. [\[15\]](#)

Using a new nonce later cannot make already exposed relations disappear. A response to suspected reuse needs to consider the affected key, stored ciphertexts and the integrity of messages already accepted. The important lesson for design is earlier: treat allocation of nonce values as

part of the protected operation, and handle failure before silently restarting a sequence. Authentication is essential, but it does not make a nonce-dependent construction safe under arbitrary nonce repetition.

7.4 Protecting ciphertext and associated data

Authenticated encryption with associated data, abbreviated **AEAD**, combines two protection goals. It aims to hide plaintext content under a stated confidentiality model and make unauthorized protected records fail an authenticity check. The associated data are supplied to that same check while remaining unencrypted by the AEAD operation. A routing header, content type, protocol version or record number may need that treatment because the receiver must interpret it or a network must read it without opening the encrypted message. Phillip Rogaway's 2002 work formalized this combined problem, including the need to bind clear data to the particular encrypted message. [\[32\]](#)

For teaching purposes, we will display an AEAD output as a ciphertext body

$$C$$

and an authentication tag

$$T$$

:

(7.7)

The subscript

$$K$$

names the secret key,

$$N$$

is the nonce,

$$A$$

is the associated-data byte string, and

$$M$$

is the plaintext. The function

Enc

performs the selected complete construction, rather than one raw AES block transformation. The tag is verification data produced with secret material. It need not be secret itself. Some interfaces package the body and tag together as one ciphertext string; our separation makes their roles visible and does not impose a universal wire format.

Authenticated decryption is an operation with a choice of outcomes:

(7.8)

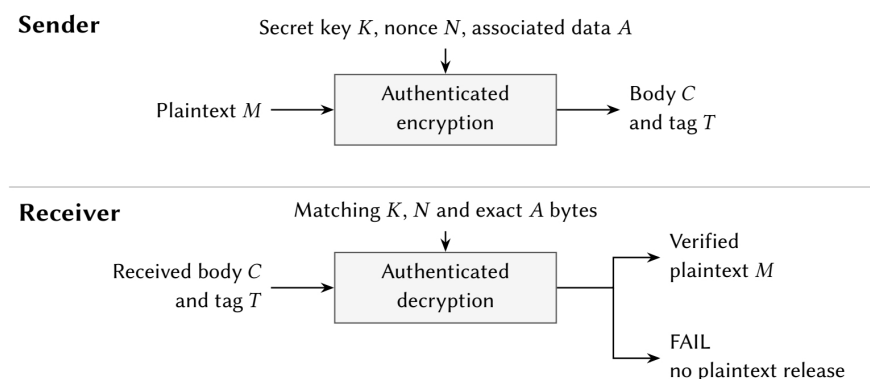
Here

FAIL

is a distinct failure indication, not an empty message or a message containing zero bytes. For data generated correctly with the matching key, nonce and associated data, correctness requires recovery of

M

. For an unauthorized new protected input, the security goal is to make acceptance infeasible within the stated attack model and usage bounds. This does not mean that an impossible mathematical guarantee replaces probability or computational assumptions. Figure 7.1 shows the two operations and their different roles. [\[34\]](#)



A tag check is an acceptance step, not a substitute for the application's context checks.

Figure 7.1. The same key, nonce and associated-data bytes connect authenticated encryption to authenticated decryption. The receiver's usable output is either verified plaintext or a distinct failure indication. Transport of the nonce, associated data, body and tag is arranged by the protocol; the secret key is not sent with the record.

Suppose Nora sends an encrypted note with clear associated data identifying version 1, a particular conversation and record number 7. The receiver must use those exact encoded bytes for verification. Substituting version 2 or moving the ciphertext under a different authenticated conversation label should cause the original tag to fail under the construction's assumptions. Omitting the header from authentication would leave room for changes in how the same body is interpreted. Protecting the body and header together addresses that connection, rather than merely attaching an unrelated authentication result to each part.

Associated data remain visible wherever they are transmitted in clear form. Including a person's name, a diagnosis or a location there would not make those values confidential. The protection is also about exact bytes, not the intended meaning of a sentence. The text bytes for `version=1` are not the same input as `Version=1`. Two visually similar Unicode strings may have different encodings, as Chapter [3](#) explained. The protocol must specify the representation and any normalization before the authenticated inputs are formed.

Combining several associated-data fields requires unambiguous framing. Joining the text fields `ab` and `c` gives the same bytes as joining `a` and `bc`. A valid tag over `abc` cannot tell the receiver which division the sender intended. A fixed schema, explicit lengths or a suitable structured encoding can preserve the boundary. This is a formatting problem before authentication, not a reason to change the tag algorithm. The receiver must agree on the fields' interpretation as well as provide the same final byte string.

Including a label is also not enough if the receiver never checks whether the label is appropriate. A record that is authentic for an old conversation should not automatically be accepted as a command in a new one merely because the packet carries its own old conversation identifier. The application needs an expected context and an acceptance policy. AEAD can bind the supplied context to the ciphertext; it cannot decide which conversation, operation or permission the receiver intended unless the surrounding system makes that decision.

Authentication under a shared key is evidence relative to possession of that key. It does not establish which one of several key holders produced a

record, provide a publicly verifiable signature, or prove that a sensor's input was truthful. Nor does a previously valid record become a forgery when an attacker sends exactly the same record again. AEAD verification can succeed again on the unchanged inputs. Detecting that replay requires protocol state, a suitable freshness rule or another specified mechanism, even if a record number is included in the associated data.

The research on composition explains why these distinctions should be built into the construction. Bellare and Namprempre analyzed ways of joining confidentiality encryption with a message authentication code, or MAC. Their results distinguish integrity of plaintexts from integrity of ciphertexts and depend on the exact unforgeability property assumed for the MAC. An encrypt-then-MAC composition with appropriate separate keys and strong authentication properties has useful guarantees; those guarantees do not establish that every concatenation of two secure-looking functions works. We will study MACs in the next chapter. Here the practical consequence is to use a specified AEAD interface rather than invent a composition from its ingredients. [\[33\]](#)

Finally, a tag length is not a complete security statement. In an idealized experiment where an unknown valid tag is uniform among

$$2^\tau$$

possibilities, one blind guess succeeds with probability

$$2^{-\tau}$$

. With at most

$$v$$

guesses, a union bound is

$$(7.9)$$

The symbol

$$\tau$$

is the tag length in bits, and

$$v$$

counts verification attempts. This is a baseline guessing calculation under the stated ideal assumption, not the full forgery bound for GCM or Poly1305. Actual bounds also depend on the construction, processed lengths, key use and the attacker’s information. A 128-bit tag does not make those other quantities irrelevant or make the sender’s nonce rules optional.

7.5 AES-GCM and ChaCha20-Poly1305

It helps to separate two jobs before following either construction. The encryption part turns plaintext bytes into ciphertext bytes. The authentication part calculates a tag from the protected inputs so that the receiver can check the record before using it. **GHASH** is GCM’s internal keyed hash calculation; its result is combined with another value to form the tag. **Poly1305** is an authenticator that uses a one-time key to calculate a tag. Neither name means that an ordinary public file digest can be used as a substitute.

Table 7.2. The AEAD constructions as used here. The twelve-byte nonce and full sixteen-byte tag are deliberate choices for this explanation; broader specifications and other profiles must be read separately.

Property Key	AES-GCM 128, 192 or 256 bits with AES; protocol dependent	ChaCha20-Poly1305 256 bits in the IETF construction
Property Nonce used here	AES-GCM 96 bits; no repeated value under a key	ChaCha20-Poly1305 96 bits; no repeated value under a key
Property Ciphertext body	AES-GCM AES counter stream XOR plaintext	ChaCha20-Poly1305 ChaCha20 stream XOR plaintext
Property Authentication	AES-GCM GHASH with an AES-derived masking value	ChaCha20-Poly1305 Poly1305 with a derived one-time key
Property Tag used here	AES-GCM 128 bits	ChaCha20-Poly1305 128 bits
Property Authentication lengths	AES-GCM Bit lengths, encoded big-endian	ChaCha20-Poly1305 Byte lengths, encoded little-endian

How AES-GCM combines its parts

For a 96-bit nonce

N

, GCM forms a 128-bit pre-counter block

J_0

by appending the four bytes 00 00 00 01. It applies AES to incremented counter blocks to generate the message keystream, beginning with the block whose final four bytes are 00 00 00 02. The cipher input is a counter block, not the plaintext itself. XOR with the plaintext produces the ciphertext body. Partial final blocks use only the required keystream bytes, so this body has exactly the plaintext's length. [\[31\]](#)

The value

$E_K(J_0)$

has another role: it masks the authentication result. It is not reused as the first message-keystream block. A separate hash subkey is

(7.10)

The input

0^{128}

is a block containing 128 zero bits. AES with the selected key

K

transforms that fixed block to the 128-bit value

H

. This derivation does not make

H

public just because the derivation procedure is public; its result depends on the secret key. GCM's security analysis accounts for how these values are derived together. Arbitrarily reusing the same application key for unrelated cryptographic purposes is a different question.

GHASH combines formatted associated data and ciphertext into one block. It uses the binary field with

2^{128}

elements and reduction polynomial

$x^{128} + x^7 + x^2 + 1$

, with the bit-to-polynomial convention defined by GCM. Here

u

is a formal polynomial variable, and its superscripts are powers; each coefficient is either 0 or 1. The polynomial specifies how to reduce a product to the field representation, rather than an ordinary integer modulus. This is different from AES's internal byte field, which has only

2^8

elements and a different reduction polynomial. The familiar pattern of field addition by XOR still applies, but copying the byte-level `xtime` rule from Chapter 6 would not implement GHASH multiplication.

The GHASH calculation carries one working value from block to block. This value is an **accumulator**: it summarizes the blocks processed so far under the defined arithmetic. At each step, the next formatted block is combined with the current value, and the result becomes the new working value. The recurrence below is a compact way to say repeat this same step for each block. It does not ask the reader to solve a new equation for every block.

Let

$X_1, \dots, X_m$

be the formatted 128-bit blocks. GHASH starts from a zero block and repeats

(7.11)

The integer

i

counts the input blocks,

m

is their total count, and

Y_i

is the intermediate accumulated result after block

i

. The bullet here means multiplication in the GCM 128-bit field. At each step, XOR the next block into the previous result, then multiply by

H

. For two blocks, this gives

$$Y_2 = X_2 \bullet H$$

and

$$Y_2 = (Y_1 \oplus X_2) \bullet H$$

. The last result,

Y_m

, is the GHASH output, which we will name

S

.

Formatting is part of this calculation. GCM first takes the associated data, adds enough zero bits to complete its last 128-bit block, then does the same for the ciphertext. It appends a final block containing two 64-bit big-endian integers: the associated-data length in *bits*, followed by the ciphertext length in bits. A part already aligned to a block needs no extra padding block. These padding bits belong to the authentication input; they do not extend the transmitted ciphertext body. For example, three associated-data bytes and five ciphertext bytes require thirteen and eleven zero-padding bytes respectively. The final length block contains 24 and 40, since

$$3 \times 8 = 24$$

and

$$5 \times 8 = 40$$

. Its two halves are 00 00 00 00 00 00 00 18 and 00 00 00 00 00 00 00 28. Hex 18 represents decimal 24 and hex 28 represents decimal 40. The authentication input has three blocks in this case: one padded associated-data block, one padded ciphertext block and one length block. The explicit lengths preserve distinctions that zero padding alone would lose.

For the full 128-bit tag used here, GCM combines the hash result with the nonce-dependent mask:

(7.12)

Here
 T
is the transmitted tag,
 S
is GHASH over the specified formatted inputs, and

$E_K(J_0)$

is one AES output block. Both inputs to the XOR are 128 bits. GHASH is a keyed algebraic component within this construction, not a substitute for the general-purpose cryptographic hashes in the next chapter. Its output is not sent unmasked as an independent message-authentication method.

An all-zero public test case makes these stages concrete. The AES-128 key is sixteen zero bytes, the nonce is twelve zero bytes, the plaintext is sixteen zero bytes, and the associated data are empty. Table 7.3 gives the intermediate values and final outputs. The ciphertext and tag appear in the collected OpenSSL tests, which identify the GCM proposal's vectors as their source. The intermediate values can be reproduced from the finalized GCM equations. These zeros are published test inputs, not advice for selecting a real secret key.^[30]

Table 7.3. A known-answer AES-128-GCM example. Each displayed value is one 128-bit block written as 32 hexadecimal digits.

Quantity	Hexadecimal value
Hash subkey	66e94bd4ef8a2c3b884cfa59ca342b2e
Ciphertext	0388dace60b6a392f328c2b971b2fe78
GHASH result	f38cbb1ad69223dcc3457ae5b6b0f885
Mask	58e2fccefa7e3061367f1d57a4e7455a
Tag	ab6e47d42cec13bdf53a67b21257bddf

Since the plaintext block is zero, its XOR with the first counter-stream block leaves that stream block as

C

. The GHASH input consists of

C

and a length block: the associated-data length is zero, and the ciphertext length is 128 bits. The last byte of that length block is therefore hex 80.

Applying Equation [7.11](#) to those two blocks gives the displayed

S

. XOR of the last two input quantities in Equation [7.12](#),

$E_K(J_0)$

and

S

, gives the displayed tag. Matching these stages checks more than matching only a round trip.

How ChaCha20-Poly1305 combines its parts

The IETF construction takes its 256-bit key and 96-bit nonce and first runs the ChaCha20 block function with counter zero. The first 32 output bytes supply a Poly1305 one-time key. Its first sixteen bytes determine a value conventionally named

r

, after specified bits have been cleared in a step called **clamping**. The next sixteen bytes determine a second value

S

. Message encryption starts at counter one, keeping those key-generating bytes separate from the message stream. This division is a defined part of the construction, not an optional offset selected for convenience.^[15]

Poly1305 authenticates the associated data and *ciphertext*, not the recovered plaintext. The construction pads each of those byte strings separately to a multiple of sixteen bytes, then appends two eight-byte length fields. Here the lengths count *bytes* and are encoded little-endian. For the

three-byte and five-byte illustration above, the length halves are 03 00 00 00 00 00 00 00 and 05 00 00 00 00 00 00 00. The padding still needs thirteen and eleven zero bytes, giving three formatted blocks, but the final length bytes are different from GCM's. Interchanging the two formats would compute the wrong tag.

The authentication arithmetic also changes. Poly1305 uses ordinary integers modulo the prime

$$(7.13)$$

The letter

p

names the arithmetic modulus. The exponent means a power of two, and subtracting five gives the particular prime used by the algorithm. This is integer modular arithmetic of the kind introduced earlier in the book, rather than binary-polynomial field arithmetic. Addition has ordinary carries before reduction. The fact that both GCM and Poly1305 use polynomial-style accumulation does not make their multiplication or byte representation interchangeable.

For the full sixteen-byte blocks in this AEAD authentication input, read block

i

as a little-endian integer and add

2^{128}

; call the resulting integer

n_i

. That additional high bit records the block's extent in the Poly1305 calculation. With

V_i

naming the accumulator after block

i

, the recurrence is

$$(7.14)$$

Start with zero. Add the encoded next block

n_i

to the current accumulator, multiply that sum by the clamped secret value

r

, and keep its remainder on division by

p

. The operations occur in that order. Every block of this AEAD-formatted input has sixteen bytes, including its final length block. Raw Poly1305 on a shorter final block has a corresponding different high-bit position; the full-block rule here is not a universal rule for all raw messages.

After the last formatted block, the tag value is

$$(7.15)$$

The index

m

is the number of formatted blocks, and

s

is the second half of the one-time key interpreted as a little-endian integer.

Add

s

to the final accumulator, retain the low 128 bits, and serialize that value as sixteen little-endian bytes to obtain

T

. Notice the two moduli: the recurrence uses

$$2^{130} - 5$$

, while the final tag extraction uses

$$2^{128}$$

. The second modulus controls the output width; it does not replace the prime in every accumulator step.

For the complete AEAD vector in RFC 8439 Section 2.8.2, the associated data contain twelve bytes and the ciphertext body contains 114 bytes. Their padding lengths are four and fourteen bytes. Adding the sixteen-byte length block gives

authentication-input bytes, or ten blocks. The published tag is 1ae10b594f09e26a7e902ecbd0600691. This vector checks the assembly of the components as well as their individual operations; the main explanation does not require reproducing its entire message and key on the page.

Limits belong to the construction and the interface

The constructions' counter widths impose per-invocation ceilings, but their ceilings differ. The finalized GCM specification limits plaintext length to

$2^{39} - 256$

bits. Dividing by eight gives

bytes. With the 96-bit nonce arrangement explained above, that corresponds to at most

sixteen-byte message blocks. IETF ChaCha20-Poly1305 reserves counter zero and can use the remaining

64-byte blocks, giving

message bytes. These are algorithm-level per-message ceilings, not recommendations for record sizes or total lifetime traffic.

A protocol, library, authentication target or application can impose much smaller limits. A limit on message count addresses a different quantity from

a limit on total processed blocks or failed verification attempts. Counting all of them across the relevant key's use is part of key management. The examples here use full 128-bit tags and twelve-byte nonces; they do not teach tag truncation. NIST's GCM revision work is also recorded in the sources so that a proposal is not silently presented as a finalized requirement. At the date of this draft, the official revision page describes a pre-draft consultation rather than a replacement final specification. [\[37\]](#)

7.6 Verification failure and message release

The receiver's important boundary is between candidate bytes produced internally and authenticated plaintext released for use. A low-level implementation may calculate candidate plaintext before it has finished verifying the tag. The application must not treat those bytes as accepted data merely because they have appeared in an output buffer. An unauthenticated instruction could cause harm when parsed or executed even if the program later displays a verification error. The AEAD contract is plaintext *or* failure, not plaintext followed by an advisory warning.

An interface that returns the plaintext only after successful verification makes this boundary easier to maintain. In a streaming interface, the caller must understand whether intermediate output is unauthenticated until a final check succeeds. Before that success, it must not be passed to an interpreter, used to update trusted records or displayed as verified content. Internal buffering and temporary storage require their own handling; they do not change the acceptance condition. Selected PyCA documentation makes this distinction explicit for low-level GCM, while its higher-level AEAD decryption methods report an invalid tag through an exception. [\[36\]](#)

Failure does not uniquely diagnose a malicious modification. A wrong key, incorrect nonce, mismatched associated-data bytes, truncated packet or corrupted tag can also make verification fail. The receiver should not recover by silently ignoring the tag, trying an unauthenticated mode or treating the failure as an empty valid message. Error handling should preserve the rejection and avoid disclosing unnecessary internal distinctions to an untrusted peer. Local diagnostics can still help an operator identify configuration and transport faults without accepting the failed record.

The optional program below uses the higher-level `AESGCM` interface from the Python `cryptography` package. Each run creates a fresh key and performs exactly one encryption. Under that one-use key, the all-zero twelve-byte nonce is distinct by construction: it is used only once for encryption. Keeping that nonce while adding further encryptions under the same key would violate the example's condition. Its purpose is to show that uniqueness, not visual randomness, is the essential nonce property in this setting.

The associated data are a fixed clear header for this demonstration, and the plaintext is a short ASCII note. The returned `sealed` byte string contains the ciphertext body followed by a sixteen-byte tag. The program first checks valid recovery twice to demonstrate that AEAD alone does not reject an exact replay. It then changes one ciphertext-body bit, one tag bit and the associated-data input in separate decryption attempts. The failure exception is handled without returning or using candidate plaintext.

Listing 7.1. Valid recovery and rejection through an AEAD library interface.

```
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

key = AESGCM.generate_key(bit_length=128)
aead = AESGCM(key)
nonce = bytes(12) # One encryption under this fresh key.
message = b"Meet at 09:00."
aad = b"notes:v1;sender=Nora;record=7"
sealed = aead.encrypt(nonce, message, aad)

for _ in range(2):
    assert aead.decrypt(nonce, sealed, aad) == message
print("valid record accepted twice")

changed_body = bytes([sealed[0] ^ 1]) + sealed[1:]
changed_tag = sealed[:-1] + bytes([sealed[-1] ^ 1])
cases = (
    ("ciphertext", changed_body, aad),
    ("tag", changed_tag, aad),
    ("associated data", sealed, aad + b"x"),
)
for label, candidate, context in cases:
    try:
        aead.decrypt(nonce, candidate, context)
    except InvalidTag:
        print(label, "rejected")
    else:
        raise AssertionError("unexpected acceptance")
```

The output reports `valid record accepted twice`, followed by `ciphertext rejected`, `tag rejected` and `associated data rejected`. The key and ciphertext vary across runs, so the program does not print a fixed ciphertext. The independently checked known-answer examples earlier in the chapter serve that different purpose. The program was checked with the documented library interface; it is an acceptance demonstration, not a complete file format, persistent nonce allocator or network protocol.

Several details connect its behavior to the chapter's model. Changing the tag is enough to fail even when the body has not changed. Changing associated data is enough to fail even when the entire sealed byte string is unchanged. Repeating a valid decryption succeeds because the library has not been given a record of already accepted message numbers. The application would need to check its expected conversation and replay state before delivering the recovered note as a new event. Under concurrency, accepting and recording a new record number must also be coordinated so that two workers cannot both deliver the same event.

Large messages make the release boundary harder to manage. A single authenticated record may require temporary storage until verification completes, and a late failure invalidates that whole record. A protocol can instead define a sequence of smaller authenticated records, each with its own properly allocated nonce and bound position. That is a protocol design, not simply cutting an existing ciphertext into pieces. The receiver must also be able to detect reordering, missing records and an unexpected end, using authenticated context and appropriate state. Independently valid chunks do not automatically prove that the complete intended file or conversation arrived.

The chapter's central lesson is that encryption is an operation with a context and an acceptance rule. A nonce keeps the construction within its intended pattern of use; associated data bind clear meaning to the protected body; and verification determines whether plaintext can be released. The mathematics explains specific failures, while the construction and interface specify how to avoid accepting them. We can now turn to hashes and message authentication with a clear distinction between a digest, a keyed check and an authenticated-encryption operation.

Questions to check your understanding

1. An AES-ECB message contains three blocks

$$P_1 \parallel P_2 \parallel P_3$$

- a. Which relation is visible after encryption, and would an AES-256 key remove it?
2. Can a predictable counter be a suitable GCM nonce source? What must two senders sharing a key coordinate, and what can go wrong after a backup is restored?
3. In Equation [7.3](#), how many nonce bits come from each field? Explain the twelve bytes for sender 7 and counter 258.
4. Why does changing only associated data fail to repair a repeated message keystream? Given the table's

$$M_1$$
,
$$C_1$$
and
$$C_2$$
, how can
$$M_2$$
be recovered?
5. A clear header is included as associated data. Is it now secret, and does successful verification alone establish that its conversation identifier is the one the receiver expected?
6. For three associated-data bytes and five ciphertext bytes, compare the GCM and ChaCha20-Poly1305 padding counts, length units and byte orders.
7. In GCM, what are the distinct roles of

$$E_K(0^{128})$$
, the first message counter block and

$$E_K(J_0)$$
? Why must GHASH multiplication not use the AES byte-field rule?
8. Which ChaCha20 block produces the Poly1305 one-time key, and which begins message encryption? Which two different moduli appear

in the Poly1305 calculation?

9. The library accepts the same valid record twice and rejects a changed tag. Why are both results consistent with AEAD's role? What should happen to candidate plaintext when verification fails?

8. Hashes and Message Authentication

Part III: Working with Secret Keys

A downloaded file may contain millions of bytes, yet a hash function can calculate a short value from them. This value is called a **digest**. It is a compact reference to the file's bytes, rather than an encrypted or compressed copy of the file. A software tool can recompute that short value and compare it with a reference before using the file. A different system may attach a short tag to a readable message so that a receiver can check whether someone with the right secret key produced it. Both situations involve compact outputs, but they answer different questions. A public digest summarizes the bytes according to a public rule. A message authentication code adds secret material and an acceptance condition.

The distinction becomes important when an attacker can change more than one field. Replacing a file and its unprotected digest is easy if the digest algorithm is public. Producing a valid authentication tag for an unauthorized new message should be difficult without the key, even after seeing tags for other messages. Neither operation necessarily hides the message, and neither decides on its own whether an old valid message should be accepted again. The protection comes from the defined construction and its use, not from the output being long or difficult to read.

This chapter develops the three main hash-security questions, explains the structure of SHA-2 and SHA-3, and follows **HMAC**, a message authentication code built from a hash function and a secret key, through a published test case. We will use a deliberately weak toy function to make collisions visible, then separate that example from the assumptions behind real cryptographic hashes. The formulas will identify both their byte inputs and their security meaning. The final section returns to representation and context, because even a correct cryptographic calculation cannot recover a field boundary or a meaning that was never encoded.

8.1 A digest of a message

A **hash function** maps an allowed input to an output called a **digest** or **hash value**. In the fixed-output cryptographic functions discussed first, a message can have many possible lengths while the digest has a specified number of bits. A message here means a bit string or, in the interfaces we use, a byte string. It can represent text, executable code, a photograph or a serialized record. The function does not need to understand that representation's human meaning. [\[38\]](#)

For a fixed-output hash, write

(8.1)

The name
 H
 identifies the selected hash algorithm,

M
 is its input message, and

d
 is the digest. The set

$\{0,1\}^n$
 consists of all strings of exactly

n
 bits. For SHA-256,

$n = 256$
 , so the output has 32 bytes. Writing those bytes in hexadecimal uses 64 characters, because each hex character represents four bits. Those 64 display characters are an encoding of the digest; they are not 64 original digest bytes.

Hashing is deterministic. The same algorithm applied to exactly the same input bytes gives the same result, regardless of whether the bytes arrived in one operation or several correctly ordered pieces. It does not require a random seed to select a new digest for every call. If two implementations disagree on the result, the cause may be different bytes, a different algorithm or an implementation error. A digest is not supposed to vary because one program read the file in larger chunks.

Table 8.1 compares two three-byte ASCII inputs. The final character changes from c, hex 63, to b, hex 62; only one input bit changes. The SHA-256 outputs differ in 124 of their 256 bit positions. This example illustrates a large change in output from a small change in input. It does not establish that every one-bit input change flips exactly half the output bits, and a similar-looking effect alone would not prove security.

Table 8.1. SHA-256 of two precisely specified inputs, without a newline. Each digest is displayed in two consecutive 32-character halves; join the halves to read its 64-character hexadecimal representation.

Input bytes	SHA-256 digest in hexadecimal
61 62 63 ASCII abc	ba7816bf8f01cfea414140de5dae2223 b00361a396177a9cb410ff61f20015ad
61 62 62 ASCII abb	715edf8ba8729420cd4d1ce85ed61954 a9f531f8c548df728c407effe839296d

A digest is useful as a compact reference to content, but it is not a lossless compressed copy. When the possible input set is larger than the output set, several inputs must share an output. There is no general decompression procedure that can identify the original arbitrary file from its digest alone. This counting fact does not make hashing useless. It explains why the security question concerns the difficulty of finding suitable matching inputs, rather than a claim that matching inputs cannot exist.

The fixed output also does not make a low-uncertainty message secret. If a hidden value is known to be one of the 10,000 decimal strings from 0000 to 9999, an observer with its unkeyed digest can hash those candidates and compare the results. A 256-bit output does not turn four decimal digits into a uniformly chosen 256-bit secret. The earlier probability chapter explains why the input distribution and available information matter. Password protection needs additional design choices, developed in Chapter 9.

Many programs calculate a file digest incrementally. They preserve an internal hash state while reading successive chunks, then finalize the computation according to the algorithm’s padding rules. Hashing the concatenation

$A\|B$
in this way is not generally the same as hashing
 A
and

B

independently and joining their digests. Nor is finalizing after

A

and treating its displayed hex digest as an ordinary input to the next call the same operation. These are different constructions, even though all can be described casually as hashing pieces of a file.^[44] A file's digest covers the bytes actually submitted. It does not automatically cover the filename, path, version label, owner or time shown beside the file. If those facts matter, a surrounding format must bind them appropriately to the content. Before deciding whether a digest is trustworthy, we must therefore ask two questions: which exact bytes were hashed, and why should this particular reference digest be trusted? The second question will lead from public hashing to authentication.

8.2 Preimages, second preimages and collisions

The main hash-security tasks differ in what the attacker receives and what they may choose. Confusing those tasks can make an otherwise correct calculation misleading. A **preimage** problem starts with a target digest

y

, for example one generated from an unknown input, and asks for an input

X

such that

$$H(X) = y$$

. The attacker need not identify the original input if several inputs have that digest. The goal is to find a matching allowed input. A **second-preimage** problem starts instead with a particular message

M

and asks for a distinct message

M'

such that

$$H(M') = H(M)$$

. The original message is fixed as part of the challenge. A **collision** problem gives the attacker more freedom: choose any two distinct messages

X
 and Y
 whose digests match. Computing the same message twice is not a collision. The two input strings must be different, even when they are displayed or stored under different names.

For an ideal n -bit hash with enough distinct candidate inputs, classical generic-search scales are summarized in Table 8.2. Here an ideal model means that new distinct inputs behave like independent uniform choices among the 2^n output values. Actual algorithms are fixed public functions, and attacks can exploit their structure. The ideal comparison is a baseline for reasoning, not a proof that every named hash achieves those costs in every setting.

Table 8.2. Different search tasks in an ideal fixed-output hash model. The work scales count classical generic hash queries, with sufficient candidate inputs; they are not exact runtimes or universal guarantees for concrete algorithms.

Task	Given in advance	Find	Work scale
Preimage	Target digest	An with	
Second preimage	Target message	with the same digest	
Collision	Freedom to choose the pair	with matching digests	

The difference comes from the comparison being made. Against one fixed ideal target, each new independent candidate digest matches with probability $1 / 2^n$. Finding any matching pair allows each new digest to be compared with all earlier digests. There are

$$q(q-1)/2$$

pairs after

$$q$$

distinct queries. The birthday reasoning from Chapter 5 places the collision scale near the square root of the output-space size:

$$(8.2)$$

The square root is the number whose square is

$$2^n$$

. Taking it halves the exponent. For an ideal 256-bit output, the generic collision scale is therefore

$$2^{128}$$

queries, while a fixed-target preimage scale is

$$2^{256}$$

. These numbers describe different tasks. They are not claims that a real computer can be assigned a useful completion date from the exponent alone.

The second-preimage row also needs its model. A fixed ordinary target message in an ideal function is different from every possible long, structured input to an iterated concrete hash. Some analyses of those hashes give message-length-dependent bounds. Similarly, a small or biased candidate space changes a preimage search, as the four-digit example showed. Security statements must identify the algorithm, input setting, resources and attack model rather than applying the table as an unconditional label.

A toy calculation makes the existence of collisions easier to see. Define an explicitly insecure byte-sum function

$$(8.3)$$

Here

$$M$$

contains

ℓ

bytes, and

m_i

is the ordinary unsigned integer value of byte

i

. The summation adds those values, and the remainder modulo 256 fits in one byte. For ASCII_{AB}, the sum is

$$65 + 66 = 131$$

, or hex 83. For ASCII_{BA}, it is

$$66 + 65 = 131$$

again. The different strings collide immediately because addition ignores their order.

For this same function, adding a zero byte also leaves the output unchanged, and increasing one byte while decreasing another by the same amount can preserve the sum. These are structural weaknesses, not a search near

$$2^{8/2}$$

. The toy function is useful only because its weakness is visible. It is not a cryptographic hash, a recommended file-authentication method or a small version of SHA-256.

Real collision resistance asks that finding a suitable distinct pair be computationally infeasible under the relevant conditions. It does not deny the counting argument that collisions exist. Preimage resistance and collision resistance are also not interchangeable. Demonstrating a pair selected freely by an attacker does not necessarily give an input matching an independently fixed target digest. This distinction will matter again when we ask whether a collision result about a public hash automatically creates a forgery against a keyed MAC.

8.3 SHA-2 and SHA-3 at the right level

Table 8.3. Selected fixed-output functions. An input step is a compression block for these SHA-2 functions and the rate portion for these SHA-3 functions. It is not a required message length.

Function	Digest bits / bytes	Input step bits / bytes	Internal state bits
SHA-256	256 / 32	512 / 64	256 chaining bits
Function	Digest bits / bytes	Input step bits / bytes	Internal state bits
SHA-512	512 / 64	1024 / 128	512 chaining bits
Function	Digest bits / bytes	Input step bits / bytes	Internal state bits
SHA3-256	256 / 32	1088 / 136	1600
Function	Digest bits / bytes	Input step bits / bytes	Internal state bits
SHA3-512	512 / 64	576 / 72	1600

SHA-256: blocks and a chaining state

SHA-256 begins from a fixed public initial state of eight 32-bit words. It pads the message, divides the padded input into 512-bit blocks and processes them in order. A **compression function** combines each message block with the current state to produce the next state. The state is the algorithm’s current working information, rather than the whole file or a secret key. Think of the process as carrying a defined summary forward after each block. After all blocks have been processed, the eight resulting words form the 256-bit digest. A schematic description is

(8.4)

Here
IV
is the specified initial value, not a fresh per-message encryption IV. The padded message has blocks

$B_1, \dots, B_k$

;

F

names the specified SHA-256 compression operation; and

S_i

is the chaining state after block

i

. This schematic is specialized to SHA-256's full-state output. The term compression refers to its fixed-size state update, not to a reversible file-compression program.

Within each block's update, SHA-256 expands sixteen 32-bit input words into a schedule of sixty-four words. Its sixty-four rounds use rotations, shifts, XOR, bitwise logical functions and addition modulo 2^{32}

. The working words are then added to the incoming chaining words as the specification requires. The arithmetic resembles operations already encountered, but their arrangement and constants define a different algorithm. An implementation cannot replace a rotation with a shift or rearrange the round updates because both appear to mix bits.

Padding is precise and public. If the original message length is

L

bits, append one 1 bit, then the smallest nonnegative number

Z

of zero bits such that

(8.5)

Finally, append the original length

L

as a 64-bit big-endian integer. The congruence reserves the last 64 positions of a block for that length, so the completed input has a length divisible by 512. SHA-256's allowed original bit length is less than

2^{64}

. The encoded length is the original message length, not its length after padding.

For abc ,

$$L = 3 \times 8 = 24$$

. The equation gives

$$z = 423$$

, since

$$24 \times 1 + 423 = 447$$

. Appending the 64-bit length gives 512 bits in total. At the byte level, the padded block contains $61\ 62\ 63\ 80$, then 52 zero bytes, then the eight bytes $00\ 00\ 00\ 00\ 00\ 00\ 00\ 18$. The byte 80 contains the appended 1 bit followed by seven of the zero bits. Hex 18 represents decimal 24. When that block is read as sixteen big-endian 32-bit words, its first word is 61626380 , its last word is 00000018 , and the intervening words are zero. Processing the specified rounds gives the SHA-256 value for abc shown earlier. The example explains why a final newline, a different character encoding or an incorrect length field changes the computation. Padding is not a secret salt and is not an instruction to add arbitrary spaces to the visible text.

SHA-512 follows a related design with 64-bit words, 1024-bit blocks and eighty rounds. Some family members use a truncated result with a different specified initial state. In particular, SHA-512/256 is a defined SHA-2 function; it is not obtained by simply taking the first 256 bits of ordinary SHA-512. Naming the exact function is part of specifying a hash operation.

SHA-3: absorbing and squeezing

SHA-3 uses a different framework, called a **sponge**. This name describes two phases: absorb input into a working state, then squeeze output from that state. The following explanation defines the working state and the steps in each phase. Its working state has 1600 bits, organized in twenty-five 64-bit lanes. A specified permutation mixes the full state through twenty-four rounds. The construction first absorbs input into part of the state, applying the permutation between the required steps, and then squeezes out the requested output from that part. The entire internal state is not published as the digest.

Two widths describe this arrangement:

(8.6)

The rate
 r

is the number of state bits used for input and output in each step. The capacity

c

is the remaining portion, which is not directly exposed through those input/output steps but is mixed by the permutation. For SHA3-256,

$r = 1088$

and

$c = 512$

. For SHA3-512,

$r = 576$

and

$c = 1024$

. Capacity is a structural parameter in the security analysis, not a count of additional user data stored beside the message.

Starting from an all-zero state, the sponge XORs a padded rate-sized input block into the rate portion and then applies the permutation. It repeats that absorption for every input block. To obtain output, it reads the rate portion after absorption; if more output is needed, it applies the permutation again before reading another portion. SHA3-256 returns only 256 output bits. Its digest therefore does not reveal all 1088 rate bits, much less the complete 1600-bit state. [\[39\]](#)

Domain-identifying suffix bits and a defined padding rule are also part of the function. SHA-3 does not consist merely of using a particular permutation somewhere in a program. A function labeled Keccak-256 can use different domain or padding rules from standardized SHA3-256 and consequently give different outputs on the same bytes. Their shared underlying design does not make their names interchangeable. This is

another reason to record the exact algorithm rather than only the digest length.

The same standard defines the extendable-output functions SHAKE128 and SHAKE256. An **extendable-output function**, or **XOF**, takes an input and supplies as much of its output stream as the caller requests. For the same SHAKE variant and message, a shorter result is the prefix of a longer result. Requesting twice as many bytes does not create an independent second hash result. The numbers 128 and 256 in the SHAKE names identify their intended security-strength levels, rather than fixed digest lengths.

Output length still matters. Requesting only eight bits from an XOF gives just 256 possible outputs, regardless of a large underlying state or a strength number in its name. Longer output must be interpreted within the construction's security bounds; it is not an unlimited source of independent secret information. Nor can a variable-output function simply be substituted wherever a particular fixed-output hash is required. The HMAC interface discussed next needs a defined fixed digest size.

Both SHA-2 and SHA-3 are public algorithms. Their use in a protocol depends on the required properties, exact parameters and appropriate implementation. The larger family number does not make SHA-2 automatically invalid, and a large state is not by itself a proof of security. The calculations in this chapter use SHA-256 for the main digest and HMAC examples, while the family comparison explains what changes when a different specified hash is selected.

8.4 Why a bare hash does not authenticate a sender

Suppose Nora sends Sam a public message and its SHA-256 digest. Sam hashes the received message and compares the result with the received digest. If the two match, he has checked a relation between those two received objects. He has not established who supplied either one. An attacker able to replace the message can also calculate the digest of the replacement. The attacker needs no collision, no knowledge of the original message's preimages and no secret key, because none was used.

The situation changes if Sam already has a trustworthy copy of the expected digest. For example, an authenticated channel or a correctly

verified signature on a manifest can bind a file digest to its intended source and release. A replacement file must then match that independently trusted value. The security question is no longer whether the attacker can send an arbitrary matching message/digest pair. It concerns changing the file while preserving a particular trusted reference. The source of trust must be identified rather than attributed to the public hash operation itself.

This distinction explains both the value and the limit of published file checksums. A digest retrieved from the same compromised location as the file can be replaced with it. A digest obtained through an independent authenticated route can be useful evidence about the downloaded bytes. A noncryptographic checksum, such as a CRC, addresses another problem: detecting specified patterns of accidental corruption efficiently. It is not designed to resist a person who deliberately constructs modifications. Calling all three objects a check value can hide very different assumptions.

It may seem that adding a secret key to the start of the message solves the authentication problem. Consider the proposed tag

(8.7)

Here
 K
 is a secret byte string and
 M

is the message. The attacker can no longer calculate the displayed
 expression directly without
 K

. Nevertheless, this particular construction has a structural weakness. SHA-256 publishes its complete final chaining state, which can be used as a starting state for further block processing. The length field in its padding does not prevent that continuation.

To see what is continued, let
 P
 denote the exact SHA-256 padding bytes for the total byte length of

$K||M$

, and let

X

be some additional bytes. Given the tag and a correct guess of that total length, an attacker can compute a digest for

$$(8.8)$$

The attacker starts from the published chaining state and accounts for the bytes already processed, including

P

. They process

X

and the new final padding for the extended input. They do not recover

K

, and they do not simply obtain the hash of

$K||M||X$

. The old padding remains embedded between

M

and

X

.

If the receiver uses Equation [8.7](#) to verify arbitrary byte messages, the attacker can submit

$M' = M||P||X$

together with this calculated tag. The receiver's own hash calculation inserts the secret key and reaches the same result. This is called **length extension**. Its direct applicability to an application depends on details such as the key-length guess and whether the message format accepts those embedded padding bytes. Rejecting that particular byte pattern is not a substitute for using a properly specified MAC.

The explanation applies to the complete-state output of SHA-256 and to analogous secret-prefix use of full-output SHA-512. It is not a claim that every SHA-2 variant or every hash has the same attack. Truncated-state variants require a separate analysis, and standardized SHA-3 does not

expose its complete sponge state in this way. Nor does the absence of this particular attack justify inventing a new keyed construction from another hash. We need an analyzed relationship among the secret input, the exposed output and the attacker's permitted queries.

HMAC supplies such a construction. Its output is a second, outer hash of an inner digest, with separately derived key pads in the two layers. Continuing the exposed outer hash does not produce the required outer hash for a longer message inside the inner layer. This observation explains why the simple continuation above does not carry over unchanged. The complete security argument still depends on properties of the underlying functions and on the precise construction; a diagram alone is not a proof.

8.5 MACs and HMAC

A **message authentication code**, or **MAC**, uses a secret key to produce a tag for a message. A party holding the matching key can check the tag against the exact received bytes. The intended security goal is that an attacker without the key cannot create an acceptable tag for a new message, even after seeing tags on other messages and, in the relevant model, choosing messages to be tagged. A scheme that works only while an attacker has never seen an authenticated message would not meet this usual goal.

Authentication here means evidence relative to possession of the shared secret and the scheme's assumptions. If Nora and Sam both know the key, either can create a valid tag. Sam cannot use that shared-key tag alone to convince a third party that Nora, rather than Sam, created the message. Other people who have received the same key have the same capability. A MAC is therefore different from a public-key signature, and key distribution determines the group of parties to which the authentication claim applies.

A MAC also leaves its message readable. An HMAC tag on a public document can protect its integrity under the shared-key model without encrypting one byte of its contents. Replaying an unchanged message and tag can still pass verification. These are separate properties, just as they were for authenticated encryption. We first examine the keyed calculation and then connect it to the application's format and freshness rules.

The name **HMAC** means hash-based message authentication code. It uses a hash function in a particular two-layer calculation, with secret-key material included in both layers. The first hash processes a prepared key and the message. The second processes another prepared-key value and the first hash result. The constants and preparation rules below define those two layers exactly. This structure is the important difference from inventing a tag by putting a secret in front of a message and hashing once.

The two HMAC layers

HMAC was developed by Mihir Bellare, Ran Canetti and Hugo Krawczyk. Its original analysis appeared in 1996; RFC 2104 described the construction in 1997. The historical examples in that RFC include older hash algorithms. Our worked instance is HMAC-SHA-256, with an explicitly named modern hash, rather than an adoption of those older algorithm choices. [\[41\]](#)[\[40\]](#)

Let

B

be the hash's input block size in *bytes*, and let

L

be its digest size in bytes. For SHA-256,

$$B = 64$$

and

$$L = 32$$

. HMAC first prepares a block-sized version of the key. If the supplied key is longer than

B

bytes, hash it once; otherwise retain it:

(8.9)

In this equation, vertical bars give a byte length, and

$$\text{zeros}(j)$$

means a string of

j

zero *bytes*. Thus
 K_0

always has exactly
 B

bytes. The hash choices discussed here have

$L \leq B$

, so hashing a long key leaves a result that can be padded to that width. The padding is part of HMAC's internal construction. It does not turn a weak human password into a strong key.

Next define two public block-sized strings:

$ipad$

consists of

B

repetitions of byte hex 36, and

$opad$

consists of

B

repetitions of byte hex 5c. The letters stand for inner and outer pad. These fixed constants are neither random nonces nor additional secret keys.

HMAC uses them as follows:

(8.10), (8.11)

First XOR the prepared key with the inner pad, append the message and hash that byte string to obtain

I

. Then XOR the same prepared key with the outer pad, append the raw inner digest and hash again to obtain

T

. XOR operates on two equal-sized blocks; concatenation joins successive byte strings. The two operations must not be interchanged. The inner digest

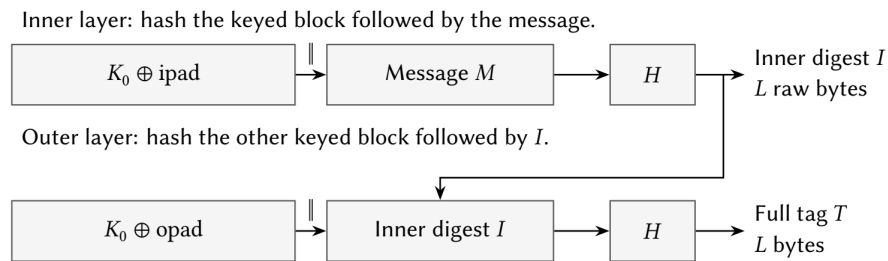
is

L

binary bytes, not the

$2L$

printable characters of its hexadecimal display.



B is the keyed block width; the same K_0 is used in both layers.

Figure 8.1. HMAC's inner and outer inputs. The message enters the inner hash; the outer hash receives its fixed-length digest. Both keyed blocks are derived from the same prepared secret key.

The block size must come from the selected algorithm. For HMAC with SHA3-256 it is 136 bytes, corresponding to the sponge rate, rather than its 32-byte digest or its complete 200-byte state. For SHA3-512 it is 72 bytes. This convention is specified for HMAC use of the SHA-3 hashes. A variable-output SHAKE function is not a drop-in choice for this fixed-digest HMAC definition. [\[39\]](#)

A complete HMAC-SHA-256 example

We can inspect each layer with Test Case 1 of RFC 4231. The supplied key is twenty repetitions of byte `0B`. The message is the eight ASCII bytes of `Hi There`, with hex encoding `48 69 20 54 68 65 72 65`. The space is one byte, and there is no final newline. These are public test inputs, not a key to use for protecting real messages. [\[43\]](#)

Because twenty bytes is less than

$B = 64$

, the key is not hashed during preparation. Append forty-four zero bytes to obtain

K_0

. In the inner keyed block, each of the first twenty bytes becomes $0B \text{ XOR } 36 = 3D$. Each of the remaining forty-four bytes becomes $00 \text{ XOR } 36 = 36$. Appending the eight-byte message gives a 72-byte inner input before SHA-256 adds its own padding.

Hashing that input gives the following inner digest. The two displayed lines are consecutive halves of one value:

```
92ab4d9a1f3b6152bca9dd9e69af43f4
ce99e42fd4e30ff972c48025b9f9cfef
```

This intermediate value is calculated from the HMAC definition for the teaching trace; the RFC's published expected result is the final tag below. The intermediate has thirty-two bytes. Its sixty-four hexadecimal characters are shown only so that a calculation can be compared without printing control characters.

For the outer keyed block, the first twenty bytes are $0B \text{ XOR } 5C = 57$, followed by forty-four bytes $5C$. Append the thirty-two raw bytes of

I

to that 64-byte block. The outer input therefore has

$64 + 32 = 96$

bytes before hash padding. Its SHA-256 result, matching the RFC's published HMAC-SHA-256 tag, is

```
b0344c61d8db38535ca8afceaf0bf12b
881dc200c9833da726e9376c2e32cff7
```

Again the lines are joined directly. The result is thirty-two bytes, not sixty-four bytes and not a text message to hash once more. A different message encoding, an extra newline or use of the inner digest's hex text would produce a different computation.

For this example, SHA-256 pads the 72-byte inner input into two blocks and the 96-byte outer input into two blocks. Their bit lengths are

$72 \times 8 = 576$

and

$96 \times 8 = 768$

, respectively. These are the lengths encoded by the hash padding at each layer, not the eight-byte message length alone. Implementations can save the hash states after processing the keyed blocks to avoid repeating that work for every message. Such an optimization must preserve the same result, and those saved states are sensitive key-related material.

The collected OpenSSL source provides a useful connection to implementation. Its selected HMAC routine hashes an oversized key, fills the key block, applies the two pad constants and finishes the outer hash with the raw inner digest. The higher-level `EVP_MAC` interface exposes HMAC to applications without requiring them to reproduce these internal steps. This reading confirms how the specified layers are represented in selected source passages; it is not a review of every optimized path or of the whole library. [\[45\]](#)

What the security statement does and does not say

HMAC security is not simply the collision resistance of the public hash. An unkeyed collision gives two messages with equal results from the hash's public initial state. It does not automatically give equal inner states after an unknown HMAC keyed block. The original analysis and later research study the precise keyed construction. Bellare's later analysis removes an earlier collision-resistance assumption through stated properties of the underlying compression function. Applying the results to HMAC's key-derived inputs also requires the specified conditions on those related inputs; it is not a theorem about every function called a hash. [\[42\]](#)

The practical conclusion is to separate the questions. A collision attack on a hash does not by itself constitute an HMAC forgery, but this distinction is not permission to choose obsolete algorithms for new designs. Likewise, a 256-bit digest is not a proof of 256-bit security for every protocol that uses it. The actual key strength, construction, number and length of processed messages, tag length, implementation and attack model affect the assurance that can be claimed.

Tag guessing makes the role of output length clear. If a tag behaved as an independent uniform

t

-bit value unknown to an attacker, one blind guess would match with probability

2^{-t}

. A union bound on

v

such verification attempts is

(8.12)

Here

v

counts attempted candidate tags, and the bound describes the stated ideal guessing model. It is not a complete HMAC security theorem. For instance, reducing a tag from 32 bytes to 16 bytes reduces

t

from 256 to 128 in this particular calculation. Real bounds can contain additional terms. Finding a repeated tag among observed messages is also a different event from predicting a valid tag for a new message.

Our example retains the full 32-byte tag. Where a protocol specifies truncation, the exact retained length and the relevant limits must be fixed by that protocol, rather than selected by an unauthenticated request. A verifier must not silently accept whatever short prefix happens to arrive. Key length is another separate choice: padding a one-byte key to 64 bytes leaves at most 256 original key candidates. For a new secret key, use an appropriate cryptographic generation or key-establishment interface, not a memorable word or the fixed key in a test vector.

Unlike the one-time Poly1305 key inside the preceding chapter's construction, an HMAC key can authenticate many messages within its intended usage limits. HMAC itself does not require a nonce parameter. A record protocol may still need an authenticated counter, challenge or

identifier to establish freshness. The absence of a nonce in HMAC's formula says nothing about whether an application can safely process the same authenticated instruction twice.

An optional library check

The following Python example checks the published tag and then uses a verification function for the full-length binary tag. Its key is intentionally the public RFC test key. The fixed known-answer comparison is a test of expected behavior, while the function's later comparisons model how candidate tags are checked. The code performs no encryption and makes no claim to implement a complete network protocol.

Listing 8.1. A published HMAC-SHA-256 result and three verification outcomes.

```
import hmac

key = bytes.fromhex("0b" * 20) # Public test key only.
message = b"Hi There"
expected = bytes.fromhex(
    "b0344c61d8db38535ca8afceaf0bf12b"
    "881dc200c9833da726e9376c2e32cff7"
)
tag = hmac.digest(key, message, "sha256")
assert tag == expected
print(tag.hex())

def verify(candidate_message, candidate_tag):
    if not isinstance(candidate_tag, bytes):
        return False
    if len(candidate_tag) != 32:
        return False
    wanted = hmac.digest(key, candidate_message, "sha256")
    return hmac.compare_digest(wanted, candidate_tag)

changed_tag = bytes([tag[0] ^ 1]) + tag[1:]
assert verify(message, tag)
assert not verify(message + b"!", tag)
assert not verify(message, changed_tag)
assert not verify(message, tag[:16])
print("valid tag accepted")
print("changed message and tag rejected")
print("short tag rejected")
```

The first output line is the 64-character tag printed above. The next three lines are valid tag accepted, changed message and tag rejected and short tag rejected. Python's `hmac.digest` returns bytes; `tag.hex()` is used only for display. The verification function computes the tag over the

supplied message and rejects an incorrectly sized candidate before comparison. [\[44\]](#)

The comparison uses `compare_digest` to avoid the content-dependent early exit of an ordinary equality check on a secret-dependent result. The supplied candidate and calculated value have the same public type and length. This interface choice reduces a particular timing risk; it does not establish constant-time execution of the whole application or protect an exposed key. Parsing limits, key storage, error handling and replay state belong to the surrounding system. Checking a published vector and rejecting these changes are useful implementation tests, but they are not a security proof for all possible inputs.

8.6 Length, encoding and context

A cryptographic function takes bytes, while an application often starts from names, numbers, text fields and commands. Agreement about those meanings requires an exact representation. Choosing SHA-256 or HMAC-SHA-256 is only one part of that agreement. The participants must also specify character encoding, field boundaries, length units, numeric byte order, tag representation and the context in which a key and message are valid. A perfectly implemented MAC can authenticate an ambiguous byte string. Consider a message formed by concatenating two fields. The field pairs `ab, c` and `a, bc` both produce `abc`. Consequently, hashing or authenticating their plain concatenation gives identical results under the same algorithm and key. No hash collision has been found: the cryptographic input is literally the same. An attacker may still exploit a disagreement between the meaning assigned to the fields at the sender and at the receiver.

One illustrative solution is to prefix each field with a fixed-width byte count:

(8.13)

Here

A

and

B

are byte strings;

$|A|$

and

$|B|$

count their bytes; and

$\text{be32}(j)$

is the four-byte big-endian encoding of the nonnegative integer

j

. Each field must be shorter than

2^{32}

bytes to fit. The decoder reads four bytes, then exactly the stated number of bytes for that field, and repeats for the second field. It must reject incomplete data, excessive declared lengths and unexpected trailing data for this two-field format.

For the first pair, the bytes are

00 00 00 02 61 62 00 00 00 01 63.

For the second pair, they are

00 00 00 01 61 00 00 00 02 62 63.

The byte strings now differ even though their unframed payloads both read `abc`. A properly specified format may use another unambiguous scheme instead. The requirement is that accepted encodings have the intended interpretation, not that every protocol adopt this particular four-byte prefix.

Hash padding does not solve this application problem. It records an overall bit length according to the algorithm, not the boundaries between an application's fields. Incremental hash and HMAC calls do not preserve those boundaries either: updating with `ab` and then `c` processes the same data as updating with `a` and then `bc`. The split between library calls is not a

cryptographic separator. Framing must enter the authenticated byte string itself.[\[44\]](#)

Text representations require the same care. A visible line ending may be encoded as LF or as CR followed by LF. A displayed accented character can have the distinct Unicode encodings discussed in Chapter 3. If a format requires normalization or canonical representation, both parties must apply the specified rule at the specified stage before generating and checking its authenticated bytes. A receiver must not verify one interpretation and then execute another after an unaccounted transformation. File formats that permit several equivalent text layouts often need a precise signing or MAC convention.

A binary digest and its hexadecimal display are also different inputs. SHA-256's binary result contains 32 bytes, while ASCII encoding of its hex display contains 64. Uppercase and lowercase hex can decode to the same binary digest under an appropriate decoder, but hashing those literal text strings gives different inputs. The chosen representation must be validated before comparison. A protocol should not leave the receiver to guess whether an incoming field is binary, hex, Base64 or a different encoding.

Context identifies which meaning is being authenticated. Suppose one key is used in a test system and in a live service, or for messages in two opposite directions. A valid message might be copied into a setting for which it was never intended. An application can bind an explicit protocol identifier, version, direction and operation to the authenticated input using an unambiguous encoding. The receiver must also compare those values with its own expected context. A valid tag on a field that says `test` does not make that record a valid live instruction.

Separate keys for separate purposes can strengthen this separation, and the next chapter explains key derivation. Public context labels do not make extra secret entropy. They state an intended domain that must be covered by the MAC and enforced by the application. Nor should an incoming label alone be allowed to downgrade the algorithm or tag length. Supported algorithms, versions and lengths are governed by the protocol's rules and authenticated negotiation where applicable.

Freshness adds a related requirement. If a command has an authenticated sequence number, an exact replay retains a valid tag. The receiver must

remember which numbers have already been accepted, or apply another defined freshness mechanism. It should also bind the number to the appropriate connection, sender and direction. HMAC can authenticate that number's bytes without deciding whether the number is new. Under concurrent processing, the acceptance and state update must be coordinated before the same operation is delivered twice.

For a file manifest, the same reasoning extends beyond its file digests. A digest of file bytes does not by itself identify a filename, release version, intended platform or set of accompanying files. Those fields must be encoded and bound to an authenticated manifest when the application relies on them. The receiver still needs the right verification key or shared secret and a trusted basis for associating it with the intended source. A content hash is a useful component of that design, rather than the whole design.

We can now distinguish four operations that are often confused. Encoding chooses how information is represented. Hashing computes a public digest with specified security goals. A MAC uses a secret key to authenticate specified bytes. A protocol binds those bytes to parties, context and acceptance rules. Keeping these roles separate makes the mathematics easier to interpret and helps explain why a correct local calculation can coexist with an insecure system. The next step is to examine where secret keys come from when the starting material is a password or an already established secret.

Questions to check your understanding

1. How many bytes and hexadecimal characters represent a SHA-256 digest? Why does a one-bit input change not prove that exactly half of the digest bits must always change?
2. Distinguish finding a preimage, a second preimage and a collision. Under the stated ideal classical model, why does a 256-bit output give a collision-search scale near 2^{128} rather than 2^{256} ?
3. Calculate the toy byte sum for AB and BA . Why does this example fail as a cryptographic hash, despite determinism and a fixed output width?
4. For SHA-256 input abc , explain the 423 zero bits and the final eight bytes. For SHA3-256, which parts of the 1600-bit state are the rate and capacity?
5. Sam receives a file and its digest from an attacker-controlled location. What can the attacker replace? In the secret-prefix SHA-256 example, why does the forged message contain old padding bytes?
6. In the HMAC test case, why are K_0 , the inner input, I and the outer input 64, 72, 32 and 96 bytes? What goes wrong if I is appended as its hexadecimal text?
7. Does knowing an unkeyed hash collision automatically give an HMAC forgery? What does the ideal bound $v/2^t$ describe, and what does it leave out?

8. Why do the two unframed field pairs ab, c and a, bc authenticate the same input? Does splitting them across two update calls fix the problem?
9. HMAC verification accepts the same tagged command twice. Does HMAC need a random nonce to repair this behavior, or what must the application's authenticated format and acceptance state establish?

9. Passwords and Key Derivation

Part III: Working with Secret Keys

When a person opens an account, a service needs a way to recognize the password on a later visit. Keeping a readable copy would make that recognition easy, but a stolen database would then expose every stored password immediately. Encrypting the copies moves much of the risk to the encryption key: whoever obtains both the database and that key can recover them. A password verifier takes a different approach. It stores the result of a specified calculation and checks whether a later candidate produces the same result, without needing a routine that decrypts the original password.

That approach still leaves a serious problem. A person may choose a familiar expression, reuse a password from another service or make a small change to a common word. An attacker who obtains a verifier can calculate results for likely candidates. The result can be long and mathematically well defined while the original choice remains easy to guess. Password hashing therefore needs to address the cost of each trial as well as the representation of the stored value.

Key derivation raises a related but different question. A protocol may already possess secret material and need separate keys for encryption, authentication or communication in opposite directions. Here the goal is to derive appropriate keys from that material, rather than deliberately make every calculation expensive. We will distinguish these two uses. **Argon2id** is the password-hashing method we will use to make password guesses costly to test. **HKDF** is a key-derivation method that prepares keys from suitable existing secret material. We will follow both, including a published numerical example of HKDF. Both constructions use public parameters, but neither can manufacture an unpredictable secret from input that an attacker already knows.

9.1 A password is not a random key

A uniformly generated 128-bit key is selected from

possibilities with equal probabilities in the ideal generation model. A password typed by a person does not normally follow that model. People use language, memorable events, keyboard patterns and familiar replacements. The number of strings that a password field could accept is therefore different from the distribution of strings that people actually choose. An attacker benefits from that distribution by trying plausible choices before unlikely ones.

Length and character variety can help, but they do not alone assign an entropy value to an individual password. A long quotation available in a public text may be much easier to predict than its character count suggests. Replacing a letter with a familiar digit can also be included in a guessing strategy. Conversely, a password generator or a genuinely random word-selection process can make choices according to a specified distribution. The strength of that process depends on its actual randomness and independence, not on whether its output resembles ordinary words.

The probability model from Chapter 5 makes the distinction precise. Suppose an attacker orders distinct candidates as

$$w_1, w_2, \dots$$

, from most to least likely given their information. If the password is the random variable

$$W$$

, the probability of success within the first

$$q$$

trials is

$$(9.1)$$

Here

$$q$$

is the number of distinct candidates tested, and each term is the probability that the password equals that particular candidate. The equation assumes that a correct trial can be recognized and that the attack tests this fixed list.

It adds the probabilities of mutually exclusive choices. Human choices need not be equally likely, so replacing every term with a reciprocal alphabet size would change the model.

For example, imagine a small illustrative distribution in which the first three candidates have probabilities

0.20

,

0.10

and

0.05

. Trying those three covers

$100 \times 0.20 + 100 \times 0.10 + 100 \times 0.05$

, or 35 percent of that distribution. We have not measured real users or claimed that these numbers apply to a particular service. The example explains why the order of guesses matters. A long output from a hash or KDF does not change the original probabilities.

Let a deterministic derivation have at most

D

possible password inputs once its public salt and parameters are fixed. It can produce at most

D

corresponding outputs, even when every output has 256 bits. For a four-digit decimal input,

$D = 10^4 = 10,000$

, far below

2^{256}

. The derivation can make each candidate costly to evaluate, but it cannot turn those 10,000 possibilities into an independently selected secret with

2^{256}

equally likely choices. Public randomness in a salt does not become additional *secret* password uncertainty.

Good password policy must also consider the person using it. In its remote-authentication setting, the July 2025 NIST SP 800-63B-4 requires at least fifteen characters for a password used as the sole factor and permits a minimum of eight when it is used only within a multi-factor process. It

favors support for long inputs, password managers and checks against commonly used or compromised choices. It rejects mandatory mixtures of character types and routine forced changes without evidence of compromise. These are requirements and recommendations within that document's scope, not a formula that makes every compliant choice equally strong. [\[46\]](#)

A service still needs an authenticated protected connection when receiving a password. Password hashing protects a stored representation against a particular later guessing problem; it does not stop a false login page from collecting what the user types. It also does not undo password reuse. An attacker who already learned the same password elsewhere may submit the correct value immediately. Different safeguards address those different routes to compromise.

9.2 Offline guessing and salts

An **online** guessing attack sends candidates to the real service. The service can count failures, limit their rate and apply additional authentication checks. An **offline** attack begins with a copy of enough stored information to test candidates elsewhere. The attacker can run the verification calculation without asking the service for permission each time. A login rate limit on the original website cannot regulate calculations performed on the attacker's own equipment.

A **password verifier** is the stored result used to test a later password candidate. It is not a copy intended to be decrypted back into the password. A **salt** is additional data supplied to the password calculation and normally stored openly with the result. A fresh random salt gives each newly created record its own input to that calculation, so the same candidate password must be tested separately for different salts. The cost settings are **parameters**: values that specify such matters as the calculation's memory use and number of passes. The formula below simply names these pieces of one stored record.

For a basic password-hashing design without an additional server secret, write the stored record as

(9.2)

Here

P

is the encoded password,

S

is a **salt**, and

F

is the selected password-hashing function. The collection

θ

contains its cost and output parameters. The record

R

stores an algorithm identifier

a

, an algorithm version

v

, those parameters, the salt and the result

V

. The semicolon separates the main data inputs from parameters in our notation; it is not a byte that the implementation inserts into the password.

On a later login, the verifier obtains the record and applies the same function to the submitted password with its recorded salt and parameters. It checks the result through the library's verification interface. It does not choose a fresh salt for that comparison, because a different salt would define a different calculation. A fresh salt is generated when creating a new record, including after an appropriate password change or upgrade.

The salt is normally public and stored with the verifier. Its purpose is to make the calculation specific to that record. Without a salt, one candidate calculation can be compared against many accounts using the same function and parameters. Identical passwords also give identical stored results. With different salts, the corresponding inputs differ, so an attacker generally has to perform a separate expensive derivation for each record being tested. The salt frustrates this reuse of work and precomputed tables; it does not prevent

a fresh dictionary attack against one known record. As a small conceptual example, two users may independently choose the same password. With salts

$$S_A$$

and

$$S_B$$

, their records use

$$F(P, S_A; \theta)$$

and

$$F(P, S_B; \theta)$$

. Distinct salts do not constitute a mathematical guarantee that these finite outputs can never collide. The important operational fact is that they define separate derivations rather than one common password-to-result table. An attacker who knows a candidate and both salts can still test that candidate against both accounts.

The Argon2 example below uses a randomly generated sixteen-byte salt, following the RFC's recommended password-salt length. Sixteen bytes contain 128 bits. For

$$n$$

independently uniform salts of that width, the birthday union bound is

$$(9.3)$$

There are

$$n(n-1)/2$$

distinct pairs, and each pair matches with probability

$$2^{-128}$$

. Multiplying gives the displayed bound. At

$$n = 2^{32}$$

generated records, it is less than

$$2^{-65}$$

. This is a calculation under the independent uniform model, not a claim that a broken generator or copied salt history satisfies it. Password records and HKDF salts will also have different reuse rules, despite sharing the same word.

The database needs protection even though its salts are not secret. The verifier results allow offline trials; account identifiers and metadata can be sensitive; and write access may let an attacker replace records or account bindings. Public parameters mean that secrecy of those parameters is not the intended defense. They do not mean that publishing the complete authentication database is harmless.

Some systems add a separate secret, often called a **pepper**, to protect against a database-only leak. NIST describes an additional keyed operation with a verifier-held secret stored separately from the password hashes. If that secret remains inaccessible, possession of the database alone no longer gives all the information used by that design. This is an additional key-management assumption. Placing the secret in the same stolen table removes that separation, and compromising the running verifier may expose it or allow its use. A public salt and a separately protected secret serve different purposes.^[46]

9.3 Memory and time as defensive costs

An ordinary cryptographic hash aims to process data efficiently. That is useful for files, signatures and many protocol operations, but it also makes each password trial inexpensive. A password-hashing function deliberately consumes configurable resources. The legitimate service normally evaluates it for the submitted password; an offline attacker may need to evaluate it for many candidates. The defensive advantage comes from making repeated trials expensive within an acceptable legitimate workload.

A simple cost model helps without pretending to predict a real attack. Let

Q

be the number of candidates an attack tests, and let

R

be its aggregate evaluation rate in candidates per second for the chosen function and parameters. If that rate stays constant, elapsed time is approximately

$$(9.4)$$

The numerator counts completed trials and the denominator counts trials per second, so

$$T$$

is in seconds. For a hypothetical million trials, a rate of

$$10^8$$

trials per second gives

$$10^8 / 10^8 = 0.01$$

seconds. A hypothetical rate of

$$10^3$$

trials per second gives

$$10^3 / 10^8 = 0.00001$$

seconds, or 16 minutes and 40 seconds.

Those rates are assumptions for arithmetic, not measurements of SHA-256, Argon2 or a particular GPU. The comparison does not promise a fixed slowdown between algorithms. Nor is testing a million candidates the same as successfully recovering every password: success depends on where the actual choice falls in the candidate list. An attacker may use many devices, different implementations or a different strategy. Actual evaluation needs measured costs, hardware assumptions and the relevant password distribution.

Increasing a computation count raises work per trial, but many small independent computations can fit on specialized parallel hardware. A **memory-hard** design also aims to make the working memory important to efficient evaluation. If an attacker uses less memory, recomputing missing information should impose additional costs. This is a time–memory trade-off: one resource can sometimes be reduced by spending more of another. Memory hardness is therefore a goal supported by analyses of specific constructions, not a claim that an attacker is physically forced to allocate exactly the defender’s amount.

For an illustrative capacity calculation, a worker using 64 MiB of working memory and a device with 1 GiB available give

The floor brackets mean round down to an integer. Under the stated full-memory-per-worker model, at most sixteen such memory allocations fit before allowing for other overhead. This is not a throughput estimate or a lower bound on an attacker’s memory. Bandwidth, processing units, scheduling and memory-saving strategies all affect the real result. Multiplying sixteen by an assumed processor speed would not supply those missing facts.

The defender also pays the cost. A busy service may perform many password checks at once. Its settings must account for concurrent memory use, response time and deliberate overload attempts, rather than only the speed of one isolated test. A desktop file-encryption operation and a heavily loaded authentication service can have different acceptable delays. Tests on the actual deployment platform are therefore necessary before selecting a cost profile, and the chosen settings need review as software and hardware change.

Existing schemes expose different controls. PBKDF2, short for Password-Based Key Derivation Function 2, repeats a pseudorandom-function calculation using an iteration count; it does not provide Argon2’s tunable large working-memory parameter. Scrypt is another memory-hard password-based design. Compatibility, available implementations and applicable approval requirements can constrain a system’s choice. We focus on Argon2id to explain memory-aware password hashing, without treating one profile as suitable for every environment or implying that its appearance in an RFC establishes NIST validation.^[49]

The original Argon2 paper by Alex Biryukov, Daniel Dinu and Dmitry Khovratovich studies this resource problem and the trade-offs around memory access. Its 2016 design discussion is a historical research source, with specific models and experimental conditions. The later RFC 9106 specifies version 1.3 and includes Argon2id. We use that RFC for the modern variant and numerical parameters, rather than lifting a historical performance claim into a promise about current equipment.^{[47][48]}

9.4 Argon2 and password verification

Argon2 is a family of password-hashing methods. We use the variant named **Argon2id**, which combines the two styles of memory access explained below. A **memory-hard** design aims to make an attack pay for substantial working memory or for the extra computation needed to reduce that memory. The purpose is to raise the cost of repeated guesses, rather than to keep the salt or the algorithm secret. To understand its parameters, picture one working area that is revisited in passes and divided into lanes. A pass revisits the area; a lane is one of its organized chains of blocks.

Argon2 organizes a working area into blocks and repeatedly computes values that depend on previously available blocks. Each memory block has 1024 bytes. Several lanes divide the computation into chains with specified synchronization rules. The result depends on the password, salt, version, variant and parameters; it is not merely a fast hash followed by a delay. A correct implementation must perform the specified computation and encode its inputs exactly.

The variants differ in how they select memory references. Argon2d uses data-dependent addressing, so references depend on intermediate data. Argon2i uses data-independent addressing. Argon2id combines them: the first half of the first pass uses Argon2i-style addressing, and the remainder uses Argon2d-style addressing. This choice balances concerns about observed memory-access patterns and time–memory trade-offs. It does not by itself certify the complete program against every side channel. RFC 9106 makes Argon2id the primary variant that implementations of the document must support. [\[48\]](#)

Reading the parameters

For the instance used here,

m

names the requested working memory in kibibytes,

t

the number of passes and

p

the number of lanes. One kibibyte, written KiB, is 1024 bytes; one mebibyte, written MiB, is

$$1024^2$$

bytes. Thus

$$m = 65,536$$

KiB corresponds to 64 MiB. This memory total is shared across all

$$p$$

lanes. It is not 64 MiB for each lane when

$$p = 4$$

.

The specification rounds the number of one-KiB blocks to a multiple of

$$4p$$

:

$$(9.6)$$

Here

$$m'$$

is the actual block count and

$$q$$

is the block count per lane. The floor operation rounds down. The allowed minimum is

$$m \geq 8p$$

. With

$$m = 65,536$$

and

$$p = 4$$

, we have

$$4p = 16$$

, so no rounding is needed:

$$m' = 65,536$$

and

$$q = 16,384$$

. Each lane is divided into four equal segments of 4096 blocks. These counts describe the specified work area; implementations can need additional bookkeeping memory.

An initial 64-byte hash binds the input values and lengths together. It includes the requested

m

, not only the rounded allocation, as well as

t

,

p

, the output length, version, variant, password and salt. Optional secret and associated-data inputs also have defined length fields. The algorithm uses BLAKE2b-based hashing, where BLAKE2b names a particular hash function, and its specified block-compression operations; it is not a generic request to run any hash over a large buffer.

The first two blocks in each lane are derived from that initial value and the lane and block indices. The remaining blocks are filled using references selected by the variant’s indexing rules. Later passes also incorporate the previous value of the block being updated. After the last pass, the final blocks from the lanes are XORed and a specified variable-length hash produces the requested output. These are dependencies that memory-saving analysis has to consider; allocating a large buffer and ignoring it would not provide the same property.

Table 9.1. The two Argon2id profiles recommended in RFC 9106. These are identified reference profiles, not measured performance figures or universal application settings.

RFC profile	Memory	Passes	Lanes
First	KiB = 2 GiB	1	4

RFC profile	Memory	Passes	Lanes
Second		3	4

KiB = 64
MiB

A record that can be verified later

The output can be stored in a PHC-formatted string, named after the Password Hashing Competition. This format places the algorithm, version, parameters, salt and verifier in one encoded record. For example, the following original teaching record was calculated for the 22 ASCII bytes of Example password only., using salt bytes 00 through 0F, the second profile above and a 32-byte result. The fixed salt and disclosed password make this reproducible test data; real registrations use a fresh random salt.

```
$argon2id$v=19$m=65536,t=3,p=4$  
AAECAwQFBgcICQoLDA0ODw$  
VHzFLJBTag+qCI1layKT334arJp7xm71yCcXws0NBNo
```

The displayed lines are joined directly with no spaces or newline. The salt and output are encoded in unpadded Base64 in this format. They are not hex strings. The version field `v=19` is decimal 19, the value hex `13` used by Argon2 version 1.3. The twenty-two-character salt field decodes to sixteen bytes, while the forty-three-character result field decodes to thirty-two bytes.

This example has no optional Argon2 secret or associated-data input. A design using either must specify how it is supplied again; it must not assume that the illustrated PHC fields carry those additional values. In particular, a separately protected server secret is not made public by adding it to the stored salt field.

Table 9.2 explains the record's roles. An account identifier and application policy exist outside this string and still need a protected association with it. The record supplies what the password-hashing library needs to reproduce the calculation; it does not establish that the database row belongs to the intended account or that its cost settings remain acceptable.

Table 9.2. Fields in the illustrated password-verification record. None of these fields is the original password.

Field	Meaning
argon2id, v=19	The algorithm variant and version to interpret.
m=65536, t=3, p=4	Memory in KiB, passes and lanes.
Salt field	The sixteen public bytes used for this record.
Result field	The 32-byte verifier, displayed in Base64.

A password verifier must not be reused as an encryption key stored in that same public record. If the stored bytes are the key required to decrypt a file, stealing the record supplies that key directly, without guessing the password. Password verification and password-based encryption can use related mathematical tools, but their stored information and security goals differ. A file-encryption design keeps its derived secret key secret and stores the necessary public derivation metadata instead.

An optional registration and verification check

This program uses the documented Argon2id interface from Python's `cryptography` library. It creates a record with a fresh random sixteen-byte salt and then verifies the same public teaching password. An incorrect candidate must raise the verification failure exception. The cost settings match the RFC's second profile; running the example requires the corresponding memory and a library backend that supports Argon2id.^[49]

Listing 9.1. Create a password record, then check a correct and an incorrect candidate.

```
from secrets import token_bytes
from cryptography.exceptions import InvalidKey
from cryptography.hazmat.primitives.kdf.argon2 import Argon2id

password = b"Example password only." # Public teaching input.
kdf = Argon2id(
    salt=token_bytes(16),
```

```

length=32,
iterations=3,
lanes=4,
memory_cost=64 * 1024,
)
record = kdf.derive_phc_encoded(password)
Argon2id.verify_phc_encoded(password, record)
print("correct password accepted")

try:
    Argon2id.verify_phc_encoded(b"Wrong teaching input.", record)
except InvalidKey:
    print("wrong password rejected")
else:
    raise AssertionError("unexpected acceptance")

```

The two output lines are correct password accepted and wrong password rejected. The encoded record varies between executions because the salt is random, so the program does not print a fixed record. The preceding fixed-salt example provides a separate reproducible value. During verification, the class method reads the recorded parameters and salt rather than generating new ones. It uses a new verification operation; a derivation object is not an indefinitely reusable state for repeated password guesses.

The record is an input to an expensive computation. An application must accept records only from its intended storage path, validate their supported form and enforce resource bounds before evaluation. A string requesting an enormous memory allocation is not safe merely because it resembles a PHC record. Malformed records and unavailable memory also need controlled handling. The short program demonstrates the successful and wrong-password paths for its own generated record; it is not a complete login service, parser policy or overload defense.

The collected libsodium code shows the same separation at another interface. Its password-string function generates a random salt and creates an encoded Argon2id record; a different function checks the password against that record. Its documentation also provides a way to ask whether an existing record needs an updated cost profile. Parameters must still be interpreted in that interface's units: libsodium's memory-limit argument is in bytes, whereas the Python example's `memory_cost` is in KiB. Copying the same integer between them without conversion changes the requested resources. [\[52\]](#)

9.5 Extracting and expanding key material

A protocol often starts with secret material that is not yet in the form its encryption or authentication algorithm expects. A key exchange, for example, can produce an encoded mathematical value with a particular distribution and structure. Using that representation directly as an AES key would leave unanswered questions about its length, distribution and intended purpose. A **key derivation function**, or **KDF**, supplies a specified way to turn suitable input material into the keys the protocol needs.

The word suitable is essential. The input must have enough uncertainty from the attacker's perspective, under the assumptions of the construction and protocol. The algorithm must also be used with its required input relationships. No deterministic calculation can repair a fully known secret, and no public salt makes a known value unknown. Derivation is a way to process and separate existing key material, not a substitute for obtaining it securely.

Hugo Krawczyk's research on HKDF distinguishes two stages: **extraction** and **expansion**. Extraction processes suitable source material into a compact intermediate key. Expansion uses that key to generate the required output bytes with a specified context. The research defines computational security goals and studies assumptions for different kinds of source. It does not claim that every distribution with a large-looking numeric value is automatically an acceptable secret source. [\[50\]](#)

HKDF is the **HMAC-based Extract-and-Expand Key Derivation Function** described in RFC 5869. Extraction prepares an intermediate key from suitable input material; expansion derives the requested output from that key. We use HMAC-SHA-256 for the numerical example. Let

IKM

denote the **input keying material**, and write its extraction step as

(9.7)

The result

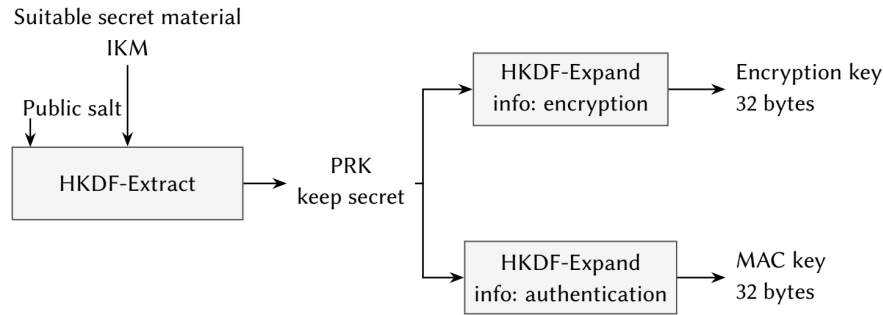
PRK

is the **pseudorandom key** passed to expansion. The subscript identifies HMAC's key argument, so the salt is used as the HMAC key and IKM as its message input. This order matters. Swapping them defines another calculation, even if both inputs have the same length. With SHA-256, PRK contains 32 bytes. [\[51\]](#)

The salt's position can initially seem surprising because the previous chapter explained MACs with a secret HMAC key. Here HMAC is being used as a component of an extraction construction under a different security argument. The salt can be public; the secrecy to be processed comes from suitable IKM. We are not asking a public salt to authenticate a sender. Security claims for HMAC in one role cannot be transferred to another role merely by giving an argument the name key. For HKDF, a random or pseudorandom salt of the hash-output length is an ideal available choice in the RFC's guidance. The specification also defines use without a supplied salt: it substitutes a string of hash-output-length zero bytes. This makes the function defined when no salt is available, rather than declaring every source and protocol equally strong without one. Salt values are assumed independent of the input material in the relevant analysis; when protocol participants supply them, the protocol must prevent inappropriate attacker manipulation.

HKDF salt reuse also differs from the per-record password-salt rule. The RFC permits the same HKDF salt with multiple suitable inputs under its assumptions. Its public salt is not an AEAD nonce subject to a universal never-repeat rule, and it is not a request to reuse one password salt across all accounts. Identifying which construction uses a field is more useful than applying one rule to every field called salt.

Expansion uses PRK to produce the requested amount of output through HMAC. With suitable inputs and assumptions, these derived keys are intended to behave like secret random keys to a feasible attacker. They are nevertheless deterministic functions of the source and public derivation inputs. Deriving many keys does not create the same information-theoretic uncertainty as independently sampling all those keys from fresh secret randomness.



Public labels separate purposes; they do not supply secret entropy.

Figure 9.1. One extraction followed by separate expansions with distinct public context strings. The diagram describes a key schedule, not a password-verification record or a complete authenticated key exchange.

This computational distinction is useful. A system can use one properly established secret to obtain several keys without exposing the master material to each use. Knowing one derived key should not reveal the other keys under the applicable security assumptions. However, someone who obtains PRK can recompute every expansion whose public inputs they know. Keeping PRK secret and limiting its exposure remain important.

HKDF does not include the deliberate memory and time costs needed to slow password guesses. Applying HKDF directly to a human password gives an attacker a comparatively inexpensive derivation to repeat. RFC 5869 explicitly separates this case from its general key-material processing. A design may first apply an appropriate password-based KDF and then derive separate purpose keys from the result, if its specified format calls for that arrangement. The expensive password step still supplies only cost per guess, rather than new secret entropy.

9.6 HKDF inputs and separate key purposes

The expansion step takes PRK, a public byte string called `info`, and a requested output length. The info string binds the key to an intended application context, such as a protocol version, direction, algorithm and purpose. It is distinct from extraction's salt. The chosen fields and their encoding must follow a protocol; arbitrary concatenation can lose boundaries in exactly the way shown in Chapter [8](#).

Let

h

be the selected hash's digest length in bytes and

L

the requested output length in bytes. In this section,

L

means the total derived output length, not a hash block size. HKDF-Expand defines

$$(9.8), (9.9), (9.10)$$

The ceiling brackets in

N

mean round up, so enough complete HMAC blocks are generated. The first block has no previous bytes because

$T(0)$

is empty. Each later block includes the *complete* preceding block, the same info string and one counter byte. The operation

first_L

retains exactly the first

L

bytes of their concatenation. OKM stands for **output keying material**.

The counter is a single binary byte, starting at value 1. It is not the ASCII character 1, and it does not continue to a two-byte representation at 256. Consequently the specified limit is

$$L \leq 255h$$

. For SHA-256,

$$h = 32$$

and the maximum is

$$255 \times 32 = 8160$$

output bytes in one expansion. That is a format limit, not 8160 bytes of newly created independent entropy.

Following a published derivation

Test Case 1 in RFC 5869 uses SHA-256 with the following exact inputs. IKM is twenty-two bytes, each hex 0B. The salt is thirteen bytes, hex 00 01 02 03 04 05 06 07 08 09 0A 0B 0C. Info is ten bytes, hex F0 F1 F2 F3 F4 F5 F6 F7 F8 F9. The requested output has

$$L = 42$$

bytes. These are public known-answer data, not secret inputs for a real key schedule.

Extraction places the salt in HMAC's key argument and IKM in its message argument. The published 32-byte PRK is shown below in two consecutive hex halves:

```
077709362c2e32df0ddc3f0dc47bba63
90b6c73bb50f9c3122ec844ad7c2b3e5
```

The displayed sixty-four characters are decoded into thirty-two binary bytes before being used as the expansion key. Using their literal text would change the key. No password stretching has occurred: this step is one HMAC evaluation over the specified public example inputs.

The required block count is

$$N = \lceil (L / 11) \rceil = 2$$

. For the first block, append counter byte 01 to the ten info bytes. HMAC's message input therefore contains eleven bytes. Calculating

$$T(1)$$

gives

```
3cb25f25faacd57a90434f64d0362f2a
2d2d0a90cf1a5a4c5db02d56ecc4c5bf
```

This is a full 32-byte block. For

$$T(2)$$

, append the ten info bytes and counter byte 02 to the *raw*

$$T(1)$$

bytes. Its HMAC message input has

$$32 + 10 + 1 = 43$$

bytes. The calculated second block is

```
34007208d5b887185865b4b0a85a993b
89b9b65683d60f0106d28fff039d0b6f
```

The two intermediate expansion values are calculated here to expose the steps. The RFC publishes the PRK and final OKM for comparison. HMAC applies its own inner and outer processing to each input; the eleven and forty-three bytes above count its message argument, not those internal padded hash inputs.

Joining the two blocks gives sixty-four bytes, but the request is for forty-two. Keep all thirty-two bytes of

$T(1)$

and the first ten bytes of

$T(2)$

, since

42 - 32 = 10

. The result matches the RFC's published output:

```
3cb25f25faacd57a90434f64d0362f2a
2d2d0a90cf1a5a4c5db02d56ecc4c5bf
34007208d5b887185865
```

The three consecutive lines contain eighty-four hex characters, representing forty-two bytes. This example checks the extraction argument order, empty initial block, binary counters, previous-block dependence and final truncation. Simply calculating HMAC once and padding its output with zeros would not implement HKDF-Expand.

Making purposes explicit

Suppose an application needs an encryption key and a separate MAC key from one PRK. It can specify separate 32-byte expansions with fixed info strings such as `notes:v1:send:enc:32` and `notes:v1:send:mac:32`. These particular literal labels illustrate purpose separation; they do not define a complete new communication protocol. The same labeling logic can distinguish the receive direction or a different protocol version. If a deployed protocol defines a different schedule, its specified labels and encodings must be followed. Changing an appropriately defined context

makes a different derivation input. The security goal is that the resulting keys can be used separately under the construction's assumptions, rather than exposing one purpose's raw key as another's. Different labels are not a mathematical guarantee that finite output strings can never coincide. Seeing different outputs in a test is also not a proof of their computational independence. Both outputs remain functions of the same protected PRK.

Output length alone does not supply this separation. For the same PRK and info, the first sixteen output bytes of a 32-byte HKDF expansion equal a sixteen-byte expansion. The length controls how many bytes are retained; it is not automatically inserted into the HMAC message. If a design needs the key's intended length bound into its context, the specified info encoding can include that length. This is why the illustrative labels above end in 32.

The info input should describe context independently of the IKM value, as RFC 5869 requires. It must not contain the secret merely as a supposed label, nor should a program expose IKM in logs while constructing it. Variable fields need an unambiguous representation, and the receiver must enforce the expected context. A key derived under an attacker-chosen old version is not appropriate for a new session merely because both programs successfully performed HKDF.

Derivation also does not authenticate a key exchange. If an active intermediary replaces unauthenticated exchange messages and establishes a separate secret with each endpoint, each side may derive perfectly well-formed keys from the wrong relationship. Authentication and transcript binding belong to the surrounding protocol. HKDF processes the material it is given; it does not identify the person who supplied it.

An optional library check

The following program checks the complete RFC example through the library's HKDF interface. It then demonstrates separate labeled expansions using the published PRK as public test data. The labels are fixed bytes, and the expected value is decoded from hexadecimal. This keeps the code close to the notation without asking the reader to implement HMAC or a new KDF.

Listing 9.2. A published HKDF-SHA-256 result and two labeled expansions.

```

from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF, HKDFExpand

ikm = bytes.fromhex("0b" * 22)
salt = bytes(range(13))
info = bytes(range(0xf0, 0xfa))
expected = bytes.fromhex(
    "3cb25f25faacd57a90434f64d0362f2a"
    "2d2d0a90cf1a5a4c5db02d56ecc4c5bf"
    "34007208d5b887185865"
)
result = HKDF(
    algorithm=hashes.SHA256(), length=42, salt=salt, info=info
).derive(ikm)
assert result == expected
print("RFC 5869 test case matched")

prk = bytes.fromhex(
    "077709362c2e32df0ddc3f0dc47bba63"
    "90b6c73bb50f9c3122ec844ad7c2b3e5"
)
def expand(label, length=32):
    return HKDFExpand(
        algorithm=hashes.SHA256(), length=length, info=label
    ).derive(prk)

enc_label = b"notes:v1:send:enc:32"
mac_label = b"notes:v1:send:mac:32"
assert expand(enc_label) != expand(mac_label)
assert expand(enc_label, 16) == expand(enc_label)[:16]
print("purpose and prefix examples checked")

```

The program reports RFC 5869 test case matched and purpose and prefix examples checked. Its inequality checks these particular inputs, while the prefix check illustrates the exact expansion rule. Neither is a general security proof. The public IKM and PRK must be replaced by properly established secret material in a real design; keeping these test values and changing only the labels would not produce secret keys. [\[49\]](#)

The collected libsodium HKDF-SHA-256 routine provides a direct source-code connection. Its extraction initializes HMAC with the salt and supplies IKM as data. Its expansion initializes HMAC with PRK, includes the previous complete output block after the first iteration, appends context and a one-byte counter, and copies only the needed final bytes. The routine also checks its maximum output length. Reading those selected operations connects the equations to an implementation, without claiming that every implementation path or side channel has been reviewed. [\[52\]](#)

9.7 Parameters, upgrades and recovery

A password-hashing scheme is not finished when the first account is registered. The service must retain enough information to verify old records, decide when their settings should change, and handle failures and recovery. Recording the algorithm, version and parameters makes that process possible. It also avoids silently interpreting an older verifier as if it had been produced by the current default. Library defaults can change; stored data need a stable, explicit meaning. An ordinary upgrade can happen after a successful login. First verify the submitted password with the old record's settings. If the record is outside the current approved policy, calculate a new record from that successfully supplied password with the new settings and a fresh salt, then replace it through a coordinated update. There is no general operation that converts an old verifier into the preferred new verifier for the same password without that password or another specifically designed migration mechanism. Hashing the old stored result again is not equivalent to running the new scheme on the original password.

This distinction also matters when comparing parameters. A different lane count affects Argon2's specified calculation, not just a scheduler option that can be changed while keeping the old output. A new memory or pass count gives a new record. Applications need an explicit accepted-policy check rather than a rule that every numerically larger field is always better. Legacy records can be recognized without allowing an untrusted request to select a weak algorithm or an excessive resource allocation.

NIST's guidance calls for increasing practical password-hashing costs over time as computing performance changes, while considering verifier performance. That guidance does not establish an Argon2-specific NIST approval, a permanent iteration number or one safe delay for every service. Systems subject to particular standards or validation requirements must choose an approved scheme and implementation for those requirements. Our Argon2id example and the NIST authentication guidance answer related questions with different scopes.

Password processing rules also need continuity. A password string must be turned into bytes consistently, including any specified Unicode normalization, and verified in full. Truncating a long password silently can make visibly different inputs equivalent. Changing encoding or normalization without a migration plan can instead reject a password that previously worked. The example programs avoid ambiguity by using

explicit ASCII byte strings; a real user interface needs a defined text policy rather than copying that simplification.

Where a separate server secret is used, its lifecycle must be planned too. Loss can make stored verifiers unusable; disclosure can remove the additional protection expected from a database-only compromise. The location of the secret, permitted access, backups, version identifiers and rotation procedure affect recovery. It should not be confused with the public salt. The exact construction determines whether an update can be performed from protected stored data or requires the user to supply the password again.

Resetting access is not decrypting a password

An account-recovery process usually establishes sufficient authority to replace a password or authenticator. It should not email the old password or claim to retrieve it from the verifier. A properly protected stored verifier still permits guessing, but it is not intended to be a reversible password archive. Recovery therefore relies on another trusted mechanism, such as an established authenticator or a controlled recovery channel, with its own abuse protections. A weak recovery path can bypass a strong normal login path.

Recovery codes are credentials too. Their generation, delivery, storage, use limits and replacement need protection appropriate to the system. A reset token must be checked for the intended account and action, expire according to policy and be invalidated when consumed. Reusing an old token must not silently reset a later password. Those acceptance rules are part of account security; choosing Argon2id for stored passwords does not implement them. [\[46\]](#)

Encrypted data introduce another boundary. A service can reset an account login while being unable to decrypt a file whose only key came from the forgotten password. Deriving a key from a new password gives a different key, not recovery of the old one. If no other authorized recovery path or key copy exists, strong encryption may make the data unavailable. If a service can recover it, some additional capability must have been designed and protected; that capability is part of the security model. A

common architectural option separates a random data-encryption key from the password-derived key used to protect it. The data are encrypted under the random key, and an authenticated wrapping operation protects that key under a separately derived wrapping key. Changing a known password may then require rewrapping the data key rather than encrypting the entire data set again. The design still needs specified derivation metadata, authenticated context, nonce management and a recovery policy. This is an architectural distinction, not a complete wrapping format or permission to compose cryptographic operations without analysis.

Stored password verifiers must remain separate from that decryption capability. Publishing the very derived key that unwraps the data key would bypass the password step. Similarly, sending a stored verifier as a reusable login credential can make it a password equivalent: an attacker may present it directly instead of recovering its original input. The protected channel, login protocol and stored representation must agree on what counts as proof of access.

We have now connected three different forms of protection. A salt makes a password record specific to its derivation; an appropriate password-hashing cost makes each guess more expensive; and a key schedule gives suitable secret material separate, defined uses. Recovery and upgrades determine how those choices survive changes over time. None removes the need for secret generation, authenticated communication or careful acceptance rules. With that foundation, the next part of the book turns to public-key methods and the problem of establishing a shared secret in the first place.

Questions to check your understanding

1. Three password candidates have probabilities

0.20

,

0.10

and

0.05

. What probability mass do their trials cover? Would a 256-bit derived result make those choices uniformly random?

2. Why can a website's login rate limit fail to control an offline attack? What does a public per-record salt change, and why must verification reuse the stored salt?

3. In the hypothetical model, how long do one million trials take at 1000 trials per second? Why does 64 MiB per worker on a 1 GiB device not establish an actual cracking rate?

4. With Argon2id

$m = 65,536$

KiB and

$p = 4$

, how much memory is requested in total, how many blocks are in each lane, and what does

$t = 3$

count?

5. Which values does the PHC record store, and why is the encoded result not suitable as a publicly stored file-decryption key? Is its $v=19$ field a count of passes?

6. In HKDF-Extract, which HMAC argument is the salt and which is IKM? Why does this use of a public salt not make HKDF a replacement for a memory-hard password KDF?

7. For SHA-256 HKDF with

$L = 42$

and ten info bytes, how many expansion blocks are needed? Explain their eleven-byte and forty-three-byte HMAC message inputs and the

final retained bytes.

8. Why should an encryption key and a MAC key have different specified purpose labels? Does requesting sixteen rather than thirty-two bytes with otherwise identical inputs give an unrelated key?
9. After verifying a password against an old record, how can the service upgrade it? Why does resetting an account password not necessarily recover data encrypted under a key derived from the forgotten password?

IV. Public Keys

10. Agreeing on a Secret

Part IV: Public Keys

Nora wants to send a private note to Sam over a network that other people can observe. They know how to encrypt and authenticate data once they share a suitable secret key. The missing step is how to obtain that key. Sending it beside the ciphertext would reveal it to the same observer, while deriving it from a public phrase would leave that observer able to repeat the calculation. The problem concerns what the participants know that an outsider does not.

Public-key methods introduce an asymmetry between information that can be distributed and information that must remain secret. A participant can publish one value while retaining related private material. Another participant can use that public value in a specified calculation without learning the private material itself. In a key-agreement protocol, each side can then obtain a common secret that a passive observer should not be able to compute under the relevant assumptions. The secret is not created by concealing the algorithm or by asking the observer to ignore a transmitted key.

We will develop this idea through Diffie–Hellman arithmetic, then change the network model to let an attacker replace messages. That change exposes the need for authentication. We will also examine key-encapsulation mechanisms and follow the components of hybrid public-key encryption through a published example. Each stage answers a different question: obtaining matching secret material, relating it to the intended party, deriving suitable purpose keys and deciding whether a protected message can be accepted.

10.1 The key-distribution problem

Symmetric encryption assumes that the intended parties already possess an appropriate secret. A cryptographic generator can create that secret locally, but local generation does not place it safely at the other endpoint. If

Nora sends a newly generated key over an observable unprotected channel, the observer learns it. If she encrypts it under an older shared key, the next question is how that older key was established. A complete design needs a starting point for trust or secrecy, rather than an endless chain of unexplained keys.

For a small closed system, a secure meeting or an established provisioning process can supply shared keys in advance. This approach has practical value and is not made obsolete by public-key cryptography. Its scaling depends on how keys are organized. Chapter 1 introduced the fully pairwise arrangement: one distinct shared key for every unordered pair. Equation 1.5 gives its count as

$n(n-1)/2$
, where
 n

is the number of participants. At

$n = 1000$
, this is

$499,500$
distinct keys, with 999 associated with each participant. These numbers count relationships, not the total number of stored copies. Both endpoints normally need access to their pair's key. Removing a participant, replacing a compromised credential or changing who may communicate requires a management process for the affected relationships. A stolen pairwise key and a stolen collection of one participant's keys also have different consequences. The example's count assumes this particular organization; another architecture can change both its storage requirements and its trust assumptions.

A trusted key-distribution service can use another arrangement. Each participant may share a long-term secret with that service, which helps distribute session keys. This changes storage and provisioning requirements, but it also introduces trust in the service and dependence on its protection and availability. A compromise can have a different scope from compromise of one isolated pairwise key. Architecture changes the security question as well as the number of stored keys.

Public-key methods offer a different arrangement. Sam can keep a private key and distribute a corresponding public key. Nora need not obtain

Sam's private key to use that public key in an appropriate encryption or key-establishment construction. Other users can use the same published recipient key without all sharing one common private secret. A directory can distribute public material efficiently, provided that the association between keys and intended recipients is protected against substitution. This last condition is easy to miss. A public key can be public without being safely replaceable. If an attacker substitutes their own public key under Sam's name, Nora can perform correct cryptography and protect her note for the attacker. A certificate, a previously trusted key, an authenticated directory or a suitably verified out-of-band comparison may provide the needed association. Each mechanism has its own assumptions. Merely downloading a file labeled `Sam's key` does not establish them.

The original Diffie–Hellman paper connected public-key work to this distribution problem. Its public-file discussion also recognized that integrity of the directory matters. We use its central algebraic idea here, while separating a modern explanatory exchange from its historical parameters and security estimates. The paper's numbers and its account of the best attacks in 1976 are not current parameter recommendations. [\[4\]](#)

It helps to distinguish three related terms. **Key establishment** is the broader task of making suitable keying material available to the intended parties. **Key agreement** describes methods in which participants contribute to establishing that material, as in the Diffie–Hellman exchange below. A **key-encapsulation mechanism** provides a public-key interface that generates a shared secret and a corresponding transmitted encapsulation. None of these names alone specifies how an application's party identities have been authenticated.

A session key also has a different lifecycle from a long-term credential. A long-term key may help identify a party over many connections. A session key protects a particular exchange or set of records. Separating them can limit exposure and allow fresh session material, but those benefits depend on the construction, generation and deletion rules. Calling a key temporary does not ensure that copies have disappeared from memory, logs or backups.

10.2 Diffie–Hellman step by step

Diffie–Hellman key agreement is a method for two parties to calculate matching secret material while sending only public values over the network. Each party makes a private choice, publishes a value derived from it, then combines the received public value with its own private choice. The resulting equality is what we will calculate first. The method does not encrypt the note itself, and an unauthenticated exchange does not prove who supplied the public values. Keeping those tasks separate makes the purpose of the following powers easier to see.

Diffie–Hellman starts from a group in which taking powers is efficient while appropriate reverse problems are believed difficult for an observer. Chapter 4 introduced cyclic groups and distinguished discrete logarithms from the computational and decision Diffie–Hellman problems. Here we use that background to follow the complete exchange. The small group makes every operation checkable, so it provides no practical secrecy.

Let

p

be a prime modulus and let

g

generate a chosen subgroup of order

q

. Recall that the modulus is the number used when retaining a remainder. A subgroup is a smaller allowed group within a larger one; its order

q

counts its elements, and a generator

g

reaches them by taking powers. The modulus tells us how to represent and multiply residues. The subgroup order tells us the period of powers of

g

and the corresponding exponent range. Nora chooses a private exponent

a

, and Sam chooses a private exponent

b

, independently using an appropriate secret-generation process in a real implementation. They compute public shares

(10.1)

The values

p

,

q

,

g

,

A

and

B

can be visible. Nora sends

A

and retains

a

; Sam sends

B

and retains

b

. The exponent is not sent as a second field. After receiving the other share, each participant combines it with their own private exponent:

(10.2)

The subscripts identify Nora and Sam. The letter

Z

denotes the raw shared group value, before the encoding and key-derivation steps needed by a complete construction.

The two results agree because the exponent rules give

$$(10.3)$$

The equalities are group equalities, represented modulo

p

in this example. Multiplication of the integer exponents commutes, so

$$ba = ab$$

. Each participant can reach the common value because they know one private exponent. The observer sees both public powers but lacks both private exponents. The algebra proves agreement for correctly formed inputs; it does not prove that a particular group makes the observer's problem hard.[\[17\]](#)

A complete small-number exchange

Choose

$$p = 23$$

,

$$g = 2$$

and subgroup order

$$q = 11$$

. The prime-order subgroup generated by 2 has eleven elements. We can check the order:

$$2^{11} \equiv -8 \pmod{23}$$

, and 2 is not the identity. Since 11 is prime, its order is therefore 11 rather than a proper nontrivial divisor. The exponent range used in this teaching execution is

1

through

$$q - 1 = 10$$

.

Nora uses

$$a = 6$$

and Sam uses

$$b = 9$$

. Nora obtains

$$A = 7 \text{ and } 23 = 0$$

. Repeated squaring makes the calculation visible:

Sam computes

$$B = 7 \text{ and } 23 = 0$$

. Squaring

$$2^4 = 16$$

gives

$$7^2 = 49 = 10 + 23 = 0$$

; multiplying its remainder by 2 gives

$$7^2 \text{ and } 23 = 10 + 23 = 0$$

. These public results can now cross the network.

Nora raises the received 6 to her private exponent 6:

Each reduction removes a multiple of 23 while preserving the residue needed in the next multiplication. Nora's result is

$$Z_N = 12$$

. She has not needed Sam's private exponent to calculate it.

Sam raises the received 18 to his private exponent 9:

His result is

$$Z_S = 12$$

as well. A direct check is

$ab = 18 \times 6 \pmod{23}$
, so

$$g^{10} = 2^{10}$$

has the same group value as

$$2^{10} = 1024 \\ \equiv 12 \pmod{23}$$

Exponent reduction uses subgroup order 11, while the represented group multiplications use modulus 23. Those are different roles.

An observer cannot reproduce this calculation merely by multiplying the two public shares. Here

$$AB = 18 \times 6 = 108 \\ \equiv 16 \pmod{23}$$

which represents

, not

. That observation does not make the example secure. Trying the ten possible nonzero exponents identifies

from

or

from

immediately. The example teaches the asymmetry of the operations, not a safe parameter set.

What a security claim needs

Correctness is the equality we just checked. A passive-secrecy claim asks what can be learned from the public parameters and shares without changing them. Computing

g^{ab}

from

g^a

and

g^b

is the computational Diffie–Hellman task. Distinguishing that value from an independent group value is a related decision task with its own assumption. The original paper already noted that a discrete-logarithm attack breaks its method without proving the converse. Modern reasoning must retain these distinctions rather than infer every security property from one difficult-looking exponentiation.

A group element is also not automatically a uniformly distributed string of the exact length an encryption interface accepts. Its encoding and distribution have structure. A specified key-derivation step processes suitable secret material and binds the required context, as Chapter [9](#) explained. Encoding the toy result as one byte 0C or as the text 12 gives different bytes, but neither encoding adds secrecy to its tiny candidate set.

Received shares need construction-specific handling. In a finite-field protocol that uses our chosen prime-order subgroup, checks can reject an

invalid encoding, the identity and elements outside the subgroup. For example,

$$Y^Y \bmod p = 1$$

tests membership among nonzero represented elements, while rejecting

$$Y = 1$$

excludes the trivial shared value it would create. These checks do not authenticate the person who supplied a valid share. They also must not be copied unchanged into a different group or an X25519 implementation, whose specified representation and validation rules differ.

Fresh private contributions are useful, but freshness must be implemented. Reusing an exponent changes which sessions depend on the same secret. Disclosure of it can expose more than one recorded exchange. **Ephemeral** means intended for one exchange or a short period, rather than retained as a long-term credential. In an appropriate authenticated ephemeral exchange, later disclosure of a separate long-term authentication key need not reveal earlier session secrets, provided the necessary ephemeral secrets and derived material have been erased and the computational assumptions hold. This property is called **forward secrecy**. It is a property of a protocol under a stated compromise model, not a result of applying the word ephemeral to one variable.

The following optional program checks only the small public teaching values. It also calculates the substituted exchanges developed in the next section. Python's modular exponentiation makes the arithmetic concise; its use here is not a recommendation to implement a real secret-exponent protocol with this script.

Listing 10.1. Check the honest and substituted small-number exchanges.

```
p, g = 23, 2
a, b = 6, 9
A, B = pow(g, a, p), pow(g, b, p)
assert (A, B) == (18, 6)
assert pow(B, a, p) == pow(A, b, p) == 12
print("honest shared value:", pow(B, a, p))

u, v = 3, 4 # Attacker's two private exponents.
U, V = pow(g, u, p), pow(g, v, p)
nora_value = pow(U, a, p)
sam_value = pow(V, b, p)
assert nora_value == pow(A, u, p) == 13
assert sam_value == pow(B, v, p) == 8
assert nora_value != sam_value
```

```
print("Nora and attacker:", nora_value)
print("Sam and attacker:", sam_value)
```

The output gives honest shared value: 12, then Nora and attacker: 13 and Sam and attacker: 8. Every input to this program is intentionally public. The attack succeeds through message substitution, not because the program secretly recovered either participant's exponent.

10.3 The active intermediary problem

A passive observer records the exchange without changing it. An **active intermediary** can intercept a share and supply another. Suppose Mallory sits between Nora and Sam and controls the delivery of the messages. Each honest participant believes the received share came from the intended other person, but neither has checked that belief against a trusted credential or authenticated conversation.

Mallory chooses two private exponents,

$$u = 3$$

and

$$v = 4$$

. She calculates

$$U = g^u \text{ and } B = h$$

for the side facing Nora, and

$$V = g^v \text{ and } B = h$$

for the side facing Sam. She intercepts Nora's public share

$$A = 18$$

and sends

$$V = 16$$

to Sam instead. She intercepts Sam's

$$B = 6$$

and sends

$$U = 8$$

to Nora instead. Both replacements are valid nonidentity elements of the intended subgroup, so membership checking alone does not stop them.

Nora now calculates a value shared with Mallory:

(10.4)

Nora knows

a

and received

U

; Mallory knows

u

and intercepted

A

. Both can obtain 13 without learning the other's exponent. Sam separately calculates

(10.5)

Sam knows

b

and received

V

; Mallory knows

v

and intercepted

B

. Their common value is 8. The two honest participants have not established a common secret with each other, even though each has completed a locally correct computation.

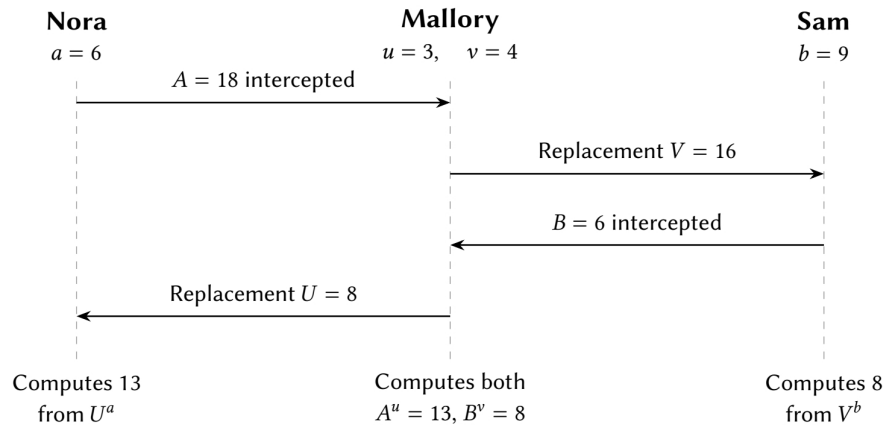


Figure 10.1. Mallory substitutes two valid shares and establishes a separate secret with each participant. Arrows show the messages actually delivered; the intended direct exchange has been replaced.

If the following encryption and authentication steps use only these unauthenticated derived secrets and public context, Mallory can follow both key schedules. She can decrypt a note from Nora, create a replacement and encrypt it for Sam, with the appropriate tags for each side. An additional independent authentication mechanism could stop this behavior; the attack description assumes that no such mechanism protects the exchange. The weakness does not depend on the small group: increasing the modulus alone does not prevent an intermediary from making her own legitimate shares. This explains why a successful AEAD tag check after bare Diffie–Hellman is insufficient to identify the peer. The tag shows consistency with the key established on that side. In the altered exchange, Mallory knows that key. Similarly, a confirmation MAC made only under the unauthenticated new secret can confirm possession relative to the substituted exchange. It does not introduce a trusted connection to Sam’s identity that the exchange never had.

Authenticating a conversation

An **authenticated key exchange** binds the resulting key to the intended party and the relevant conversation under a defined model. A signature-based construction can use long-term verification keys whose associations are already trusted. A pre-shared-key construction starts from a different kind of credential. The authentication may be one-way or mutual,

depending on whether one or both parties require that assurance. These choices determine what the protocol proves and which credentials it needs. The material authenticated must be defined carefully. Protecting only an isolated share can leave its peer, role or session ambiguous. A **transcript** is the specified record of the relevant handshake messages and context, represented as exact bytes. Protocols can bind the participants, roles, shares, algorithm choices and fresh session information to that conversation. A transcript hash is useful as a compact input, but the public hash alone does not authenticate it; it must be connected to the protocol's authentication mechanism.

Adding signatures to a few fields is not a general proof of a secure protocol. The keys must be associated with the intended parties, and the signed material must prevent substitution or confusion between roles and sessions. An attacker can also copy, delay or relay valid messages. Faithfully forwarding an authenticated conversation is not automatically a confidentiality break: a network relay may carry protected data without learning it. The question is whether the attacker can make a party accept a false relationship, learn a protected fresh key or violate another specified goal. [\[56\]](#)

Bellare and Rogaway's full 1993 paper provides a foundational example of defining these goals rather than judging a protocol only by its message diagram. It models an adversary controlling delivery and starting multiple protocol instances, and distinguishes fresh session keys from exposed ones. Its detailed constructions address the symmetric two-party setting, where long-term shared secrets already exist. We use its separation of goals and adversarial capabilities; we do not present that paper as a proof of the unauthenticated Diffie–Hellman exchange or of every later public-key handshake. [\[53\]](#)

Confirmation, identity and compromise

Key confirmation provides evidence that the other side possesses matching keying material in the relevant exchange. It can be explicit, through an appropriate confirmation message, or integrated into subsequent authenticated operations. It does not mean that a message has been approved by a human, and its identity meaning depends on the

authentication already established. In our substituted example, both confirmations can succeed with Mallory unless another credential binds the exchange to the intended peer.

The distinction also affects which endpoint has evidence at a particular time. Nora may encrypt to a trusted recipient public key while Sam is offline. That does not tell Nora that Sam has received, decrypted or processed the note. A suitably authenticated response can supply additional evidence according to its protocol. Confidential delivery capability, sender authentication and proof of actual receipt are separate properties.

Finally, compromise has a time and a scope. A model may allow disclosure of one session key while requiring another session to remain protected. It may distinguish loss of a long-term authentication key from disclosure of an ephemeral secret during the exchange. A claim of forward secrecy must say which of those events it covers. No protocol can keep a plaintext secret from a recipient device that is already handing that plaintext to an attacker. These boundaries make a security statement useful rather than weaker: they identify exactly what the construction is intended to protect.

10.4 Key-encapsulation mechanisms

A **key-encapsulation mechanism**, abbreviated **KEM**, packages public-key establishment behind a small interface. Sam generates a recipient key pair and makes the public key available through an appropriate distribution process. Nora uses that public key to produce both a shared secret and a transmitted encapsulation. Sam uses the encapsulation and his private key to obtain the matching secret. An **encapsulation** is the public data sent for this recovery step; it is not the secret itself. In the interface, **encapsulate** names the sender's operation and **decapsulate** names the receiver's. The application message is not an input to this basic KEM operation.

Write

pk_R

for the recipient's public key and

sk_R

for its private key. The letters pk abbreviate public key; sk abbreviate secret key, which in this public-key setting is the private key. The pair lists the

public value first and the private value second:

(10.6), (10.7), (10.8)

The subscript

R

identifies the recipient. KeyGen creates the key pair; Encaps uses the public recipient key and internal randomness; Decaps uses the private recipient key and the transmitted encapsulation

c_K

. The arrow indicates an algorithm result and allows randomized algorithms where specified. In this notation the shared secret is

Z

, and

c_K

is its public encapsulation. It is distinct from the later ciphertext that carries the note.

For honestly generated inputs, the KEM's correctness condition is that

$Z' = Z$

except with the construction's permitted negligible failure probability. Some constructions have exact correctness for valid inputs, while others define a small possible failure probability. This statement concerns agreement, not secrecy. An insecure algorithm could publish its secret inside

c_K

and still make both outputs match every time. NIST SP 800-227 uses this separation between definitions, correctness and security when explaining KEMs. [\[54\]](#) The basic interface normally generates its secret during encapsulation rather than accepting an application-chosen secret as another argument. It is therefore not simply a command to encrypt a password supplied by Nora. Certain constructions implement encapsulation by generating a random value and applying public-key encryption to it; other constructions obtain the secret through different mathematics. Both can fit a KEM interface if they meet its required properties. Their byte formats, assumptions and security analyses can differ.

Connecting a KEM to Diffie–Hellman

A DH-based KEM gives the earlier arithmetic another arrangement. Let Sam’s recipient exponent be

r

, with public key

$$R = g^r$$

in an illustrative finite-field group. Nora generates a fresh ephemeral exponent

e

and publishes its share

$$E = g^e$$

as the encapsulation. She computes

$$R^e$$

; Sam computes

$$E^r$$

. Their raw group values agree:

$$(10.9)$$

The recipient key pair may already exist, so Sam need not send a new live share before every message. Nora can encapsulate for him while he is offline. Her ephemeral private exponent must still be generated securely and protected for the time it is needed.

A Diffie–Hellman-based KEM, abbreviated DHKEM, does more than return the raw group value. It encodes inputs, applies required validation and derives the KEM shared secret with defined context. In HPKE’s base DHKEM, this context contains the encoded ephemeral public key and the encoded recipient public key. The context binds the derivation to the particular encapsulation and recipient key, rather than treating the raw DH bytes as an interchangeable application encryption key. The exact labeled derivation is part of the construction. [\[55\]](#)

The finite-field explanation is conceptual. HPKE’s numerical example below uses X25519, an elliptic-curve Diffie–Hellman function whose byte

representation and operations differ from powers modulo 23. Its public input and raw output use 32-byte encodings, but they are not obtained by replacing

p

with a large decimal number in our toy script. Chapter 12 develops the curve mathematics. Here we rely on the specified function and library interface to keep the roles visible without introducing all its point arithmetic at once. [\[58\]](#)

This arrangement also explains a limitation. Anyone with Sam’s public key can perform a fresh base encapsulation and know the secret they generated. Sam’s successful decapsulation alone does not identify Nora as the encapsulator. Authentication needs an additional specified relationship, such as an authenticated KEM mode, a pre-shared secret or an appropriate enclosing protocol. The existence of a recipient private key does not imply that the sender also proved possession of a credential.

Security goals and malformed input

KEM secrecy asks whether the shared secret is protected relative to the public key, encapsulation and the attacker’s allowed operations. A chosen-ciphertext model can permit the attacker to obtain decapsulation results for other encapsulations while excluding the challenge itself. That is stronger than giving the attacker only one public encapsulation to observe. The precise security notion and construction determine what can be concluded when the KEM is embedded in another protocol.

A recipient must also follow the specified rules for received keys and encapsulations. An invalid length, unsupported parameter set or prohibited public value must not be reinterpreted casually as some nearby valid input. On the other hand, a KEM is not generally a stand-alone test of whether arbitrary received bytes came from the intended sender. A changed encapsulation can produce a failure or a different shared value, depending on the algorithm and input; subsequent authenticated processing must then follow its defined acceptance rules.

Errors can themselves become observations. Distinct timing, messages or other behavior may expose information about private operations. A

mathematically correct KEM with a leaking implementation can fail its intended secrecy goal. NIST's recommendations distinguish algorithm, implementation, device, channel and key-use requirements for this reason. Correctness of one decapsulation result does not certify the endpoint or the application.^[54]

The recipient public key needs validation and an appropriate association with its owner. Its private counterpart needs protection and a defined lifecycle. Fresh encapsulation randomness is necessary even when the recipient key is intentionally reused. A KEM can output a suitable fixed-length secret key, but an application may still need a specified KDF to obtain its required key sizes and distinct purposes. That post-processing follows the application schedule; it is not permission to add an arbitrary hash or reuse one secret for every role.

10.5 Hybrid encryption and context binding

A KEM handles compact key establishment. A symmetric authenticated-encryption scheme handles the application data. Combining these roles gives **hybrid public-key encryption**: use the recipient's public key to establish secret material, then use derived symmetric material to encrypt the note. This avoids asking a public-key operation to carry every byte of a large document directly. The combination must be a specified construction, because the component names alone do not define secure encodings, key schedules or acceptance behavior.

Hybrid Public Key Encryption, or **HPKE**, is the construction described in RFC 9180. It combines a key-encapsulation mechanism (KEM), a key derivation function (KDF) and authenticated encryption with associated data (AEAD). The KEM gives a shared secret and an encapsulation, the KDF creates the context's keys and related values, and the AEAD protects the plaintext with its associated data. A **suite** is the specified combination of these algorithms and their settings. Both endpoints must select the same supported suite and interpret all inputs consistently. The RFC is an Informational IRTF/CFRG publication with concrete algorithms and test vectors; its existence does not constitute a general certification of every application using those primitive names.^[55]

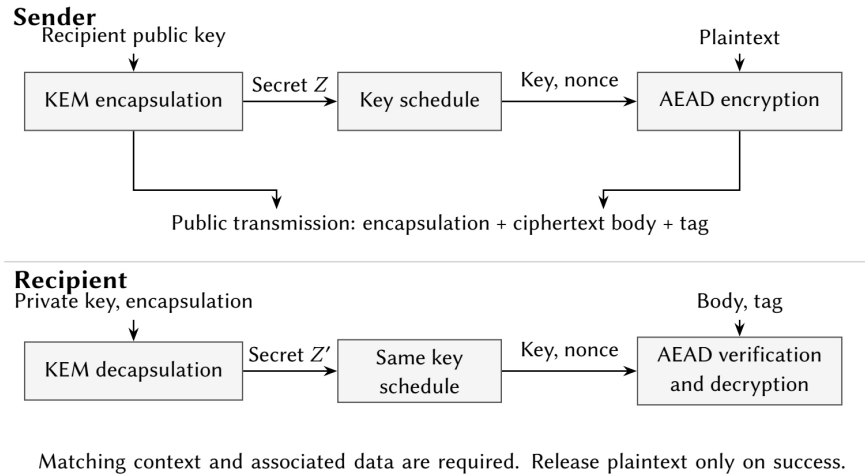


Figure 10.2. The roles in base hybrid public-key encryption. The public transmission contains an encapsulation and protected data; the secret KEM result and derived AEAD key stay inside the endpoints.

The diagram shows a one-way construction from sender to recipient. A response channel needs an appropriate separately specified schedule and context. An application must not reverse the roles of an existing sender context and start encrypting with the recipient's copy of the same key and nonce sequence. That would reuse values across directions. HPKE provides defined mechanisms for building on its context, but a complete two-way channel needs additional protocol design.

Binding the suite, keys and purpose

HPKE uses labeled versions of HKDF's extraction and expansion rather than unstructured calls over raw DH. Its labels include the literal version string `HPKE-v1` and an identifier for the relevant suite. The KEM derivation uses a KEM-specific identifier; the enclosing key schedule identifies the KEM, KDF and AEAD together. The labeled expansion also encodes its requested length in two bytes. These details prevent specified uses from being silently treated as one common unlabeled derivation.

For base DHKEM, a context is formed as

Here `enc` is the serialized ephemeral public key, and the second term is the recipient public-key encoding. Their lengths and interpretations are fixed by the selected KEM. This is not a concatenation of arbitrary usernames whose boundaries the receiver has to guess. The specified labels, secret input and context together determine the KEM shared secret.

HPKE's enclosing key schedule then incorporates a mode byte, a labeled hash of its pre-shared-key identifier and a labeled hash of the application's info input. In base mode, there is no pre-shared secret, but the defined empty values and labels still matter. From the resulting schedule it separately derives an AEAD key, a base nonce and an exporter secret. The exporter supports defined further derivations; it is not the ordinary ciphertext or a value to publish. This provides two relevant places for application context. The `info` input is bound into context establishment and hence its derived material. The AEAD associated-data input binds clear per-message bytes to a particular protected message. Neither field automatically checks what the application expected. The receiver must supply the intended info and associated data, and enforce its own expected identity, role, operation and session rules. A different purpose string should not be accepted merely because a sender supplied it as a suggestion.

Binding algorithm identifiers also does not by itself secure an unauthenticated negotiation. If an attacker can force both participants to select an undesired older suite before the binding occurs, the resulting calculation can be internally consistent with that forced choice. Allowed-suite policy and negotiation authentication remain part of the surrounding protocol. Domain separation and protection against negotiation downgrade answer different questions.

Modes and the meaning of authentication

HPKE defines four modes, summarized in Table 10.1. The base mode uses the recipient public key but no sender credential. PSK means pre-shared key. Auth uses an authenticated encapsulation operation with sender private-key material, where the chosen KEM supports it; AuthPSK combines that operation with the specified pre-shared input. The table describes the sources of additional assurance, conditional on trusting their associations and following the mode’s requirements.

Table 10.1. HPKE modes and additional sender-side material. Every mode still requires the intended recipient key and correct application context. Support for authenticated encapsulation depends on the KEM.

Mode	Additional material	What it can bind
Base	None	No sender credential is established.
PSK	A shared secret and its identifier	Possession of that pre-shared secret.
Auth	Sender private key; trusted sender public key at the recipient	The authenticated encapsulation’s sender-key relationship.
AuthPSK	Both of the preceding inputs	Both relationships, within the mode’s assumptions.

AEAD inside base HPKE detects unacceptable modifications relative to the established key, but that is different from identifying a particular sender. A stranger can obtain Sam’s public key and create a fresh valid base-mode encryption for him. The new message may decrypt correctly while being unauthorized by the application. Authenticated modes add specific possession relationships; they do not turn an arbitrary displayed name into a verified real-world identity or produce a public digital signature on every plaintext. Likewise, a sender’s use of Sam’s trusted public key does not prove that Sam has already received the message. Base encapsulation requires no reply. The difference between intended-recipient confidentiality and recipient participation matters in asynchronous delivery. If the application needs confirmed receipt, it must define the authenticated response and what event that response acknowledges.

A published suite and its byte lengths

The first base-mode test in RFC 9180 uses DHKEM with X25519 and HKDF-SHA-256, then HKDF-SHA-256 as the enclosing KDF and AES-128-GCM as the AEAD. Its numeric identifiers are KEM 32, KDF 1 and AEAD 1, with mode byte 0. The encapsulation contains 32 bytes; the KEM shared secret contains 32 bytes; the AES key contains 16 bytes; and the GCM nonce contains 12 bytes. These are different values even where their lengths happen to agree.

Table [10.2](#) gives selected published results. Its public test inputs include fixed ephemeral and recipient keys and a twenty-byte info value. Those keys are disclosed known-answer data. Reusing them in a product would not establish secrecy. The table’s longer hex values are displayed as two consecutive halves to keep them readable.

Table 10.2. Selected published values from RFC 9180 Appendix A.1.1. Join consecutive hex halves within a row. The test values are public and are not recommended keys.

Value	Hexadecimal bytes
Encapsulation 32 bytes	37fda3567bdbd628e88668c3c8d7e97d 1d1253b6d4ea6d44c150f741f1bf4431
KEM secret 32 bytes	fe0e18c9f024ce43799ae393c7e8fe8f ce9d218875e8227b0187c04e7d2ea1fc
AEAD key 16 bytes	4531685d41d65f03dc48f6b8302c05b0
Base nonce 12 bytes	56d890e5accaaf011cff4b7d

First the DHKEM applies its labeled extraction and expansion to the raw X25519 result, binding enc and the recipient key. That produces the table’s KEM secret. The enclosing base-mode schedule then binds the mode and info to produce the listed AES key and base nonce. A 32-byte raw curve output is therefore not simply shortened to sixteen bytes to obtain this particular AES key. Both derivation layers and their labels are part of the published calculation.

HPKE’s stateful record interface obtains each message nonce from the base nonce and the current sequence number:

The function

!XORP(i, 12)

encodes the nonnegative integer

i

in exactly twelve big-endian bytes. XOR combines those bytes with the base nonce. The initial sequence number is 0, whose twelve bytes are all zero, so the first nonce equals the base nonce. For sequence 1, its last byte changes from hex 7D to 7C; for sequence 2, it becomes 7E. This is XOR, not ordinary addition to the nonce.

The published first plaintext has 29 bytes and its first associated-data input has seven bytes, the ASCII encoding of `Count=0`. AES-GCM produces 29 ciphertext-body bytes and a sixteen-byte tag, giving 45 AEAD output bytes. When `enc` is prepended, the simple combined representation has

32 + 45 = 77

bytes, before any application headers or framing. The nonce is derived, so that representation does not add a separate twelve-byte transmitted nonce field.

A stateful HPKE context increments its sequence as specified and must fail at its defined message limit rather than wrap. Its sender and receiver must stay in the required order, and unrecoverable message loss requires discarding the associated context. Algorithm and library limits also apply. The single-shot interface used below creates a separate context for each encryption, so it does not ask a beginner to manage an HPKE sequence manually. Fresh encapsulation randomness is still necessary; repeating the same complete setup can repeat the derived key and initial nonce.

An optional HPKE library example

The installed `cryptography` library exposes a single-shot HPKE suite using these three algorithms. The recipient key pair in the program is generated locally, which fixes which public key belongs to the private object in this demonstration. A real sender obtaining that public key over a network would still need an appropriate trusted association with the

intended recipient. The program uses the base mode exposed by this interface and the info input to bind its fixed application context. [\[57\]](#)

Listing 10.2. Base-mode HPKE: correct decryption, failed alterations and no sender identity.

```
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives.asymmetric import x25519
from cryptography.hazmat.primitives.hpke import Suite, KEM, KDF, AEAD

suite = Suite(KEM.X25519, KDF.HKDF_SHA256, AEAD.AES_128_GCM)
recipient = x25519.X25519PrivateKey.generate()
public_key = recipient.public_key()
info = b"notes:v1:delivery"
message = b"Meet at 09:00."
sealed = suite.encrypt(message, public_key, info=info)

assert len(sealed) == 32 + len(message) + 16
for _ in range(2):
    assert suite.decrypt(sealed, recipient, info=info) == message
    print("valid packet accepted twice")

changed = sealed[:-1] + bytes([sealed[-1] ^ 1])
for packet, context in ((changed, info), (sealed, info + b"!")):
    try:
        suite.decrypt(packet, recipient, info=context)
    except InvalidTag:
        pass
    else:
        raise AssertionError("unexpected acceptance")
print("changed tag and context rejected")

other = suite.encrypt(b"Another sender.", public_key, info=info)
assert suite.decrypt(other, recipient, info=info) == b"Another sender."
print("another sender also accepted")
```

The program reports valid packet accepted twice, changed tag and context rejected, and another sender also accepted. The fourteen-byte message produces a 62-byte combined packet: thirty-two bytes of encapsulation, fourteen encrypted body bytes and sixteen tag bytes. The generated keys and encapsulation vary across runs, so no fixed ciphertext is printed. This example uses the same suite as the published vector, with different generated inputs and the single-shot interface's context treatment. The final line is an intended result. The encrypt operation needs only the recipient public key, so a different caller can create another acceptable packet. A valid base-mode ciphertext is not proof that Nora sent it. Similarly, decrypting the same combined packet twice creates the matching receiver context each time; the interface is not storing an application's replay history. Applications that require one-time processing must enforce that requirement.

The public single-shot interface here carries auxiliary authenticated application information through info; the lower-level RFC record vector also has a separate associated-data input. Those are distinct interface choices, not a reason to ignore context. The known-answer checks compare the published low-level inputs against their expected results, while this printed program checks the documented high-level behavior. Successful execution does not implement public-key distribution, sender authorization, storage policy or a full delivery protocol.

Two meanings of hybrid, and what remains outside the construction

Hybrid public-key encryption combines public-key establishment with symmetric data protection. A second use of *hybrid* combines multiple key-establishment mechanisms, often a traditional method with a post-quantum one. The goal of a carefully designed post-quantum/traditional combination is to retain the required protection if an appropriate component remains secure under the combination's assumptions. This is not achieved merely by concatenating arbitrary outputs and declaring success. NIST SP 800-227 describes multi-algorithm KEMs with specified combiners that bind the component secrets and relevant public inputs. Its X-Wing example combines ML-KEM and X25519. That combination is a different architectural layer from choosing a KEM, KDF and AEAD for message encryption. An application can use both forms of hybrid at once. Our X25519-only HPKE example illustrates public-key/symmetric hybrid encryption and does not supply post-quantum protection for its key establishment. [\[54\]](#)

Forward secrecy remains a separate question. In HPKE's defined modes with a long-term recipient key, later compromise of that recipient secret allows decryption of recorded past encapsulations and ciphertexts encrypted under it. A fresh sender ephemeral key alone does not prevent this: the attacker can combine the recorded public encapsulation with the compromised recipient key. This differs from an appropriate authenticated exchange in which both sides' necessary ephemeral secrets were erased and neither can be reconstructed from the later compromised credential.

Metadata and replay also need their own treatment. The visible recipient routing, packet length and timing may reveal information even when the

plaintext is protected. A recorded single-shot packet can be delivered again, and correct decryption alone does not make it a new instruction. Context binding, authenticated application identifiers and acceptance state help the application distinguish intended events, but their definitions must be part of its protocol rather than inferred from the AEAD tag.

The central achievement is now clearer. Public-key establishment lets participants create matching secret material without sending a readable session key. Authentication connects that result to the intended relationship. A specified KDF separates its uses, and AEAD protects the resulting data under an acceptance rule. These components can support strong systems when their assumptions and interfaces fit together; none performs every task by itself. The next chapter examines public-key encryption more directly through RSA and ElGamal.

Questions to check your understanding

1. Why does the fully pairwise arrangement for 1000 participants need 499,500 distinct keys? Does a public-key directory remove the need to protect the association between a key and its owner?
2. In the group generated by 2 modulo 23, what are the roles of modulus 23 and order 11? With
$$a = 6$$
and
$$b = 9$$
, calculate
$$A$$
,
$$B$$
and both shared results.
3. Why does multiplying
$$A$$
and
$$B$$
give 16 rather than the shared value 12? What does the agreement equality prove, and why does the toy group's size give no practical secrecy?
4. Mallory supplies
$$U = 8$$
to Nora and
$$V = 16$$
to Sam. Which two values can she calculate, and why do valid subgroup checks or later MACs under those substituted secrets not identify the intended peer?
5. What does key confirmation establish? Why is faithfully relaying an authenticated conversation different from establishing separate attacker-known secrets with the two participants?

6. What does Encaps return besides its transmitted encapsulation?
Does a correct KEM necessarily hide that value, authenticate its sender or accept an application-chosen plaintext?
7. In the published HPKE suite, distinguish the 32-byte encapsulation, 32-byte KEM secret, sixteen-byte AEAD key and twelve-byte base nonce. Why is the first 29-byte plaintext represented by 77 combined bytes before framing?
8. Why does the library example accept a packet created by another caller and accept an exact packet twice? What changes when a protocol requires an authenticated sender or one-time processing?
9. Does X25519 plus HKDF and AES-GCM constitute a post-quantum/traditional hybrid? Does a fresh sender ephemeral key make recorded HPKE messages forward secret against later compromise of the long-term recipient key?

11. RSA and ElGamal

Part IV: Public Keys

Nora knows Sam's public key and wants to send him a confidential message while he is offline. She cannot ask him to contribute a new share in a live exchange, but she can use an encryption scheme designed for his published key. Sam later applies his private key to the received ciphertext. This arrangement separates the ability to create a protected message from the ability to read it. It also introduces a question that ordinary arithmetic cannot answer by itself: what must happen between the readable message and the public-key calculation so that an observer cannot test guesses, alter its meaning or learn from the receiver's responses?

RSA is a public-key method named after Rivest, Shamir and Adleman. In its encryption use, anyone with the recipient's public key can create a ciphertext, while recovery requires the corresponding private key and the specified decoding steps. RSA uses multiplication, whole-number powers and remainders. **ElGamal encryption**, named after Taher ElGamal, provides another public-key method using the group calculation behind Diffie–Hellman and a new secret choice for each encryption. We will explain their calculations from the inputs onward and distinguish the basic mathematics from the complete schemes that provide a stated security guarantee. [\[5\]](#)[\[60\]](#)

We will first build a small RSA key and carry a number through encryption and recovery. Then we will examine what factoring explains, why raw RSA leaks information, and how OAEP changes the message before the RSA operation. Finally, a small ElGamal example will show both the value of fresh randomness and the limits of encryption without protection against modification. The small numbers let us check each step; they provide no useful secrecy. The optional programs repeat the teaching calculations and demonstrate a documented RSA-OAEP interface, without requiring programming to follow the chapter.

11.1 Public and private operations

A public-key encryption scheme has three tasks: generate related keys, encrypt with the public key and recover with the private key. The public key need not be concealed from an observer. It does need a trustworthy association with the intended recipient, as Chapter [10](#) explained. If Nora accepts an attacker's key as Sam's, mathematically correct encryption can deliver her message to the attacker. The public-key calculation does not repair the mistaken association. Likewise, anyone who knows Sam's public encryption key can normally create a new ciphertext for him. Successful decryption therefore does not, by itself, establish who sent a message or whether that person was authorized to send it.

The basic RSA calculation can be described in three steps. First, the recipient creates a key pair using two secret prime numbers. Second, a sender prepares a message as a number and applies a calculation using the public key. Third, the recipient uses related private information to reverse that calculation and check the prepared message. A **prime** is a whole number greater than 1 with no positive divisors except 1 and itself. We use primes because their arithmetic lets us construct the matching reverse operation. We do not ask the sender to discover the two secret primes before encrypting.

For the two-prime RSA construction considered here, the public key contains a modulus

n

and a public exponent

e

. The modulus is the product of two distinct odd primes,

p

and

q

. A related private exponent

d

permits the reverse operation. An **exponent** tells us how many times to use a value as a factor:

5^3

means

$$5 \times 5 \times 5$$

. The **modulus** tells us which remainder to keep after the calculation. A **representative** is the integer used to stand for the prepared input. We first use an integer representative

$$m$$

in the range

$$0 \leq m < n$$

, rather than a sentence or a file. The basic operations are

$$(11.1)$$

Read the first formula as: take the message number, raise it to the public exponent, then keep the remainder on division by

$$n$$

. Read the second as: take that result, raise it to the private exponent, and again keep the remainder on division by the same

$$n$$

. The number

$$m$$

stands for the prepared input, not yet for an arbitrary sentence. How to prepare the input safely is part of the OAEP discussion later in this chapter.

The symbol

$$c$$

denotes the ciphertext representative. The public operation raises

$$m$$

to the power

$$e$$

and retains its remainder on division by

$$n$$

. The private operation does the corresponding calculation with

$$d$$

. Exponents

e

and

 d

are related through the prime factors; they are not arbitrary passwords. These formulas describe the RSA encryption and decryption *primitives*, not a secure way to encrypt an application message directly.[\[59\]](#)

The distinction between an integer and an application message will matter throughout the chapter. An encryption scheme specifies how to prepare bytes, introduce randomness, convert between bytes and integers, and reject invalid encodings after the private operation. An application then specifies identities, permitted algorithms, message framing and acceptance rules. For example, recovering bytes that say `pay 100` does not establish whether an account exists or whether the instruction has already been processed. The original RSA research made public and private transformations concrete; later scheme definitions address security requirements that the bare transformation does not meet. We preserve that historical contribution while using the complete scheme's definition when discussing its protection.[\[5\]](#)

A similar arithmetic pattern appears in RSA signatures, but a signature is not properly described as *encrypting with the private key*. Signing and verification have different goals, encodings and interfaces from encryption and decryption. RSA-OAEP is an encryption scheme; RSA-PSS is a signature scheme. A verifier does not need to recover a hidden message to check a signature, and a public signature is not confidential. We will develop signatures in their own chapter. Here, keeping the roles distinct prevents an attractive algebraic symmetry from becoming an incorrect description of a real protocol.

11.2 How RSA builds a key

Here **arithmetic** means the rules and calculations with numbers that let us create the two matching operations. Multiplying

 p

and

q

gives the public modulus

n

. The remaining task is to choose exponents

e

and

d

that undo one another under that modulus. We cannot just divide 1 by

e

as an ordinary fraction: our calculations keep integer remainders. Instead, we use a period related to both primes. We will first explain how to find this period, then use it to choose the exponents.

A **multiple** is a whole-number product: the positive multiples of 4 begin 4, 8, 12, while those of 6 begin 6, 12, 18. Their **least positive common multiple** is 12, the first positive value in both lists. We abbreviate this operation as

lcm

. For RSA, the two lists come from

$p - 1$

and

$q - 1$

. We call their least positive common multiple

$\lambda(n)$

, reading the Greek letter as lambda. With distinct primes

p

and

q

, we write

The value

$$\lambda(n)$$

supplies a common period for the exponent calculation modulo the two primes. We choose a public exponent

$$e$$

with

$$3 \leq e < n$$

that is **coprime** to

$$\lambda(n)$$

: their greatest common divisor must be 1. We then find its **modular inverse**

$$d$$

, a positive integer whose product with

$$e$$

has remainder 1 modulo

$$\lambda(n)$$

:

$$(11.3)$$

Read the congruence as: multiply

$$e$$

by

$$d$$

and keep the remainder on division by

$$\lambda(n)$$

; that remainder must be 1. Equivalently,

$$ed - 1$$

must be an integer multiple of

$$\lambda(n)$$

. This is why

$$d$$

is a modular inverse rather than the ordinary fraction

$$1/e$$

. We will calculate it from the division steps in the example. Coprimality is the condition that ensures this inverse exists.

Building one small key

Take

$$p = 61$$

and

$$q = 53$$

. Their product is

$$n = pq = 61 \times 53 = 3233$$

. To find the common period, calculate the greatest common divisor of 60 and 52:

$$60 = 52 + 8$$

,

$$52 = 6 \times 8 + 4$$

, and

$$8 = 2 \times 4$$

. The last nonzero remainder is 4, so

$$\gcd(60, 52) = 4$$

. Therefore

$$(11.4)$$

We choose

$$e = 17$$

, which is coprime to 780. The extended Euclidean calculation from Chapter [4](#) gives the inverse. First divide with remainders:

The last nonzero remainder is 1, confirming coprimality. Substituting backwards expresses that 1 as an integer combination:

The first substitution replaces 2 by

$$17 - 15$$

. The next replaces 15 by

$$780 - 45 \times 17$$

. Collecting the coefficients leaves

$$1 \times 780 - 45 \times 17 = 1$$

. These are the same division relationships rewritten to identify which multiple of 17 leaves remainder 1 modulo 780.

Thus

$$-367$$

is an inverse of 17 modulo 780. Adding 780 gives the positive representative

$$d = 413$$

. A direct check is

$$17 \times 413 = 7021 = 9 \times 780 + 1$$

.

Our public key is

; our private calculation uses 413. A real private-key representation commonly also retains the prime factors and values that accelerate the operation. Every one of these particular teaching values is public and unsuitable for protecting anything.

Encrypting an integer

Let

$$m = 65$$

. We calculate

$$e = 907 \text{ mod } 221$$

by repeated squaring, reducing after every multiplication. Table [11.1](#) records the five powers needed. Its first nontrivial step is

$$65^2 = 4225 = 1 \times 3233 + 992$$

Squaring that remainder gives

$$992^2 = 984064$$
$$= 304 \times 3233 + 1232$$

Continuing with remainders avoids constructing an enormous decimal integer. It gives the same modular result because replacing a number by its remainder preserves later sums and products modulo the same

Table 11.1. Repeated squares for the public RSA calculation.

Exponent	Remainder of a^{exponent} raised to that exponent, modulo

Since

$$17 = 16 + 1$$

, only the last and first entries need to be multiplied:

$$(11.5)$$

The reduction subtracts

$$15 \times 223 = 3345$$

, leaving 2790. Anyone with the public key can repeat this calculation. The fact that 2790 looks different from 65 is not evidence that a small set of possible inputs has been hidden. Indeed, that public repeatability will expose a weakness of textbook encryption. For the moment, we are checking the forward arithmetic, not claiming that this tiny example or this raw encoding is secure.

Why the private operation recovers the input

There are two levels to the explanation of recovery. The example shows what happens to one message number; a correctness argument explains why every allowed message number returns correctly. We use one standard result about prime numbers and one about remainders. **Fermat's little theorem** says that, for a prime

$$p$$

and an integer

$$m$$

not divisible by

$$p$$

, the remainder of

$$m^{p-1}$$

on division by

$$p$$

is 1. The next paragraph applies exactly this statement. The special case in which

p

divides

m

is handled separately, so the argument does not silently exclude such messages.

Equation [11.3](#) implies

$$ed - 1 = k\lambda(n)$$

for some nonnegative integer

t

. Because

$$p - 1$$

divides

$$\lambda(n)$$

, the difference

$$ed - 1$$

is also a multiple of

$$p - 1$$

. For an input not divisible by

p

, Fermat's little theorem gives

$$a^{p-1} \equiv 1 \pmod{p}$$

. Raising

m

to

ed

consequently leaves the same residue as

m

modulo

p

. If the input *is* divisible by

p

, both

m

and

$$m^{ed}$$

have residue zero modulo

$$p$$

, so equality still holds. The same two cases apply modulo

$$q$$

; we have not assumed that every input is invertible modulo

$$n$$

.

The second result is the **Chinese remainder theorem**. For two coprime moduli, knowing a number's remainder under each determines one value in the combined range. For example, 8 has remainder 2 modulo 3 and remainder 3 modulo 5. Among the integers from 0 through 14, it is the only value with both remainders. Adding 15 preserves both remainders, which is why the uniqueness is stated modulo

$$3 \times 5$$

. In RSA the two moduli are the distinct primes

$$p$$

and

$$q$$

, and the combined modulus is their product

$$n$$

.

The Chinese remainder theorem now connects those two equalities. Since

$$p$$

and

$$q$$

are distinct primes, they are coprime. A residue modulo

$$pq$$

is uniquely determined by its residues modulo

p

and modulo

q

. We have just shown that

m^{ed}

and

m

have the same residue under both, so

(11.6)

This proves recovery for every representative

$0 \leq m < n$

, including inputs divisible by a prime factor and the input zero. It is a statement of correctness for valid keys and the prescribed arithmetic. It does not establish that a person without

d

cannot learn a message, or that an implementation's timing and error behavior reveal nothing.

For the example, direct modular exponentiation gives

$100^{27} \text{ mod } 1021 = 94$

. We can also see the private calculation through its two smaller prime moduli. Reduce the private exponent to

$d_1 = 67 \text{ mod } 61 = 6$

and

$d_2 = 67 \text{ mod } 53 = 14$

. The ciphertext has remainders

$2790 \text{ mod } 61 = 45$

and

$2790 \text{ mod } 53 = 34$

. Repeated squaring produces the values in Table [11.2](#); each column uses its own modulus. Reading the columns separately is essential: 61 and 53 cannot be interchanged in the middle of a calculation.

Table 11.2. The smaller powers used to recover the two private-operation residues.

Exponent	Power of	Power of
	, modulo	, modulo

Exponent	Power of	Power of
	, modulo	, modulo

Exponent	Power of	Power of
	, modulo	, modulo

Exponent	Power of	Power of
	, modulo	, modulo

Exponent	Power of	Power of
	, modulo	, modulo

Exponent	Power of	Power of
	, modulo	, modulo

For the first modulus,

$$59 \equiv 59 \pmod{61}$$

, so the product

$$12 \equiv 12 \pmod{61}$$

has remainder 4 modulo 61. For the second,

$$49 \equiv 49 \pmod{53}$$

, so

$$46 \equiv 46 \pmod{53}$$

has remainder 12 modulo 53. We need one result

$$m$$

satisfying both

$$m \equiv 4 \pmod{61}$$

and

$$m \equiv 12 \pmod{53}$$

. Write

$$m = 12 + 53h$$

, where the integer

$$h$$

counts how many multiples of 53 have been added to the remainder 12. Substitution into the first relation gives

$$59(12 + 53h) \equiv 4 \pmod{61}$$

. Since

$$59^{-1} \equiv 38 \pmod{61}$$

, we obtain

$$h \equiv 12 \cdot 38 \pmod{61}$$

. Taking

$$h = 1$$

gives

$$m = 12 + 53 = 65$$

, as required.

This method explains why practical RSA private keys often store extra prime-related values. Working modulo smaller primes and recombining the results can accelerate the private operation. It also creates implementation responsibilities: sensitive intermediates, fault handling and secret-dependent behavior need protection. An equality written on paper does not validate an optimized routine. The small public Python calculation below uses ordinary

integer operations for inspection, not a constant-time implementation for secret keys. The public exponent 65537, equal to

$$2^{16} + 1$$

, is a common practical choice; it permits a short public-exponent calculation without making the private exponent equal to it.

Another common presentation uses **Euler's totient**,

$$\varphi(n) = (p-1)(q-1)$$

, the number of invertible residues modulo

$$n$$

for these two distinct primes. The letter

$$\varphi$$

is read phi. For modulus 15, the invertible representatives are 1, 2, 4, 7, 8, 11, 13 and 14, giving

$$\varphi(15) = (3-1)(5-1) = 8$$

. Each has greatest common divisor 1 with 15. Choosing

$$d$$

as the inverse of

$$e$$

modulo

$$\varphi(n)$$

also works, since

$$\lambda(n)$$

divides

$$\varphi(n)$$

. The two presentations can give different positive representatives for

$$d$$

without changing the required reverse operation. RFC 8017 states the private-exponent condition using

$$\lambda(n)$$

. It also describes multiprime RSA; our explanation stays with two primes so that both the construction and its correctness can be followed directly. [\[59\]](#)

11.3 What factoring and RSA inversion mean

To **factor** a number is to find whole numbers that multiply to give it; for an RSA modulus, the prime factors are the values of interest. To **invert RSA** is to reverse its public operation for a given output. These tasks are related, but their definitions are different. RSA is often called a **trapdoor** construction: the intended reverse is efficient with certain private information, while reversing an appropriately generated instance without that information is assumed difficult. Here the trapdoor is the private information that makes the intended reverse calculation efficient.

An attacker who factors a properly generated two-prime modulus obtains

p

and

q

. From them, the attacker can calculate

$\lambda(n)$

and the inverse of

e

, then perform the private operation. Factoring is therefore a sufficient route to breaking RSA's trapdoor. The RSA inversion problem asks a different question: given the public values

(n, e)

and an appropriate challenge

C

, recover the representative

m

satisfying

$m^e \equiv C \pmod{n}$

. The RSA assumption describes difficulty on the specified generated keys and challenge distribution. It does not say that every possible input is difficult: encrypting 0 or 1 with the raw operation gives an immediately recognizable result.

For general RSA inversion, no proof is known that every efficient inversion method must factor

n

. We should therefore avoid replacing an assumption about RSA inversion with a claimed proof that it is as hard as factoring. The known route through factoring explains the relationship in one direction; the converse is a separate research question. Recovering one weakly encoded message can also be much easier than solving general inversion. A dictionary of likely messages, a special small input or an information leak from a receiver can defeat an application without revealing the factors. Boneh and Shoup distinguish these problems and examine the effects of poor RSA key generation.^[7]

The totient illustrates how apparently secondary private information can reveal the factors. If an attacker learns

$$\varphi(n)$$

for a modulus with two distinct prime factors, then

$$(11.7)$$

This follows by expanding

$$(p - 1)(q - 1) = pq - p - q + 1$$

For our example,

, so

$$p + q = 3233 - 3120 + 1 = 114$$

The two factors have product 3233 and sum 114. They are the roots of
, giving

$$(11.8)$$

Here

z

is a candidate for either unknown factor. A root is a value making the polynomial

$$z^2 - 114z + 3233$$

equal zero. The displayed expression uses the quadratic formula to find those values from the known sum and product. The symbol

$\pm$

tells us to use both addition and subtraction:

$$(114 \pm \sqrt{114^2 - 4 \cdot 3233}) / 2 = 61$$

and

$$(114 - \sqrt{114^2 - 4 \cdot 3233}) / 2 = 53$$

. The square root is of 64 because

$$114^2 - 4 \cdot 3233 = 64$$

. Multiplying the two recovered values gives 3233, and adding them gives 114, independently checking both conditions. This is not an attack using only the public modulus; it explains why leaking this extra private value changes the problem completely.

Key generation matters as much as nominal key length. If two supposedly independent moduli share a prime, a greatest-common-divisor calculation can expose it. For example,

$$3233 = 61 \times 53$$

and

$$3589 = 61 \times 59$$

satisfy

$$\gcd(3233, 3589) = 61$$

. The Euclidean algorithm is efficient even when its inputs are large; the weakness is the repeated factor, not a newly efficient general factoring algorithm. Secret generation, prime selection and private-key storage are

therefore parts of the security argument. A modulus with 2048 bits also does not provide 2048 bits of classical security. That number measures the modulus representation, while attack cost depends on the mathematical problem and available algorithms.

RSA additionally shares the quantum-computing limitation introduced in Chapter 4. Shor's algorithm gives an efficient factoring method in the appropriate quantum-computing model. That theoretical result does not certify a current machine's ability to factor a particular deployed modulus, but it does mean that ordinary RSA is not a post-quantum encryption method. Deciding how long information must remain confidential, and which approved mechanisms apply to a system, requires more than choosing a familiar large number. The small calculations here explain the structure; they are not permanent key-size or migration advice. [22]

11.4 Why textbook RSA is unsafe

Textbook RSA, also called raw or unpadded RSA, means using the integer transformation directly as if it were a complete encryption scheme. There is no OAEP encoding to prepare the message. This stripped-down form is useful for learning the powers and remainders, but its weaknesses explain why the rest of the construction matters. We will first let an observer compare guessed messages, then show how changing one ciphertext can change its recovered number.

Suppose Nora directly represents either YES or NO by a publicly known integer and applies Equation 11.1. An observer can encrypt both candidates under Sam's public key and compare each result with the intercepted ciphertext. No private key is needed. Because the raw transformation is deterministic, the same representative always produces the same ciphertext under the same key. Even a modulus that is infeasible to factor does not stop this public comparison. This violates the message-hiding goal developed in Chapter 5: an attacker choosing two different valid representatives can distinguish their raw RSA encryptions perfectly by calculating the public operation.

The raw operation is also multiplicative. Given a ciphertext

C

for

m

, choose an invertible residue

s

and form

$$(11.9)$$

Its private-operation result is

$$m' = ms \bmod n$$

, because

$$c' = (cm') \bmod n$$

. For our key, let

$$s = 2$$

. The public calculation turns

$$c = 2790$$

into

$$c' = 3017$$

, and the private operation returns 130, twice the original 65. An attacker need not learn 65 to cause that algebraic change. If an application acts directly on the recovered number, correctness of decryption gives no protection against the altered meaning.

This relation also explains why access to a decryption service can be powerful. If that service refuses the exact target ciphertext but returns the raw private-operation result for a related ciphertext, the attacker can obtain

$$ms$$

and multiply by

$$s^{-1}$$

modulo

n

to recover

m

. In the example, the inverse of 2 modulo 3233 is 1617, and

$$130 \times 1617 \bmod 3233 = 65$$

A complete chosen-ciphertext security definition therefore considers related queries, not merely requests to decrypt the identical challenge. Real attacks may obtain less information per query, but repeated small observations can still matter. An API that exposes the raw private operation has provided a different capability from a carefully specified complete encryption interface.

Special inputs offer another warning. In a *different* tiny key, take

$$n = 31 \times 5 \times 11$$

and

$$e = 3$$

. Encrypting the integer

$$m = 2$$

gives

$$c = 2^3 = 8$$

, with no modular wraparound. Anyone can recover 2 by taking the ordinary integer cube root of 8. The problem is not that every use of exponent 3 reveals every message this way; the example requires a raw representative whose cube is smaller than

$$n$$

. It shows why a small raw message cannot rely on the modulus to hide it. Changing the public exponent does not, by itself, solve the deterministic-comparison or multiplicative weaknesses.

Adding an invented random prefix is not an adequate general repair. Randomized encryption needs a defined encoding, suitable randomness and a security argument matching the attacker's capabilities. The decoder must also implement the corresponding checks without becoming a source of extra information. There is a useful lesson here beyond RSA: padding is not merely whitespace added until a message fits a block. In a public-key

encryption scheme, the preparation of a message is part of what the scheme means and part of what its analysis must justify.

11.5 Padding and the RSA-OAEP guarantee

OAEP stands for **Optimal Asymmetric Encryption Padding**. Bellare and Rogaway introduced the construction in work presented at EUROCRYPT 1994, whose proceedings appeared in 1995. RSA-OAEP combines an OAEP encoding with the RSA operation. The encoding uses a hash function, a mask-generation function and a fresh random seed. A seed is the fresh byte string that supplies randomness to this encoding; a mask-generation function expands an input into a byte pattern of the required length. The specified XOR steps combine those patterns with the data; the receiver reverses the masking and checks the specified structure before returning the message. It is a complete construction, rather than a request that an application designer improvise extra bytes. We will follow the byte layout in RFC 8017 and then distinguish its security statement from broader claims about a communicating system. [\[61\]](#)[\[59\]](#)

The space available for a message

Let

k

be the length of the RSA modulus in bytes, rounded up if needed, and let

h

be the hash output length in bytes. If the application message has

ℓ

bytes, the permitted length satisfies

$$(11.10)$$

For a 2048-bit modulus,

$$k = 256$$

. With SHA-256,

$$h = 32$$

, so the maximum is

$$2^{32} - 84 - 2 = 190$$

message bytes. A 32-byte random symmetric key fits; a large document does not. The output ciphertext still has

$$k = 256$$

bytes regardless of which permitted message length was used. Its fixed length hides differences within this one encoded block, not the total length or traffic pattern of an enclosing application. Splitting a file into independently encrypted RSA chunks also creates ordering and substitution questions; a specified hybrid design is the more appropriate architectural model introduced in Chapter [10](#).

OAEP has an associated label

L

, whose hash is checked on recovery. The label is not secret, and it is not automatically transmitted with the ciphertext. Both sides must use the intended value and the same parameter choices. The generic encoding accepts a label, while RFC 8017's stated PKCS #1 usage fixes it to the empty string. Our library example uses that empty-label case. If a different protocol specifies a nonempty label, its definition and handling belong to that protocol; merely adding a label does not authenticate a sender. Hash choice and the hash used within the mask-generation function must also match. A decoder cannot safely guess among unspecified combinations.

Building and masking the encoded block

Write

$$H(L)$$

for the

h

-byte hash of the label. Construct a data block

Here
 M
 is the message byte string;

$\parallel$

means concatenation, not arithmetic addition. The field
 PS
 consists of

$k - \ell - 2h - 2$

zero bytes, and 01 is one byte of hexadecimal value 01 marking where the
 message begins after those zeros. Thus

DB

has

$k - h - 1$

bytes. A maximum-length message makes
 PS

empty, which is allowed. This separator rule also permits a message that
 begins with zero bytes: the decoder finds the separator before interpreting
 the remaining bytes as the message.

Choose a fresh unpredictable seed

R

of

h

bytes. Let

$MGF1(X, a)$

mean the specified mask-generation function that expands input
 X

into exactly

a

bytes. Using XOR, form

(11.12), (11.13), (11.14)

The name D denotes the masked data block;

S denotes the masked seed. They are not additional secret keys. The first mask is calculated from the seed and has the same length as DB .

. The second is calculated from the already masked block D and has the same length as the seed. The leading 00 is one byte, so EM has

bytes. The letter R here denotes OAEP randomness, rather than the fresh exponent used in an ElGamal example later in the chapter.

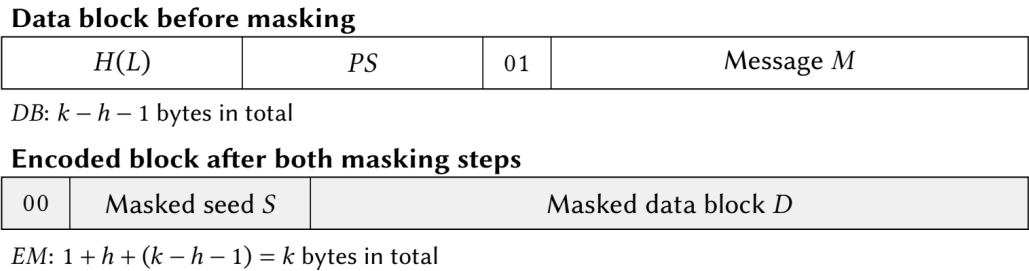


Figure 11.1. The two OAEP byte layouts. The data block is masked before it enters the encoded block; the seed is masked as well. The lower row contains

and

, not readable copies of

and

. Field widths in the drawing are schematic.

MGF1 generates its requested output by hashing its input followed by a four-byte, big-endian counter: counter values begin at zero, successive digest outputs are concatenated, and the result is truncated to the requested

byte length. Both sides use the same specified hash for these counter calculations. This is a deterministic expansion, not fresh entropy and not a password-stretching operation. The new randomness comes from

R

. In Figure [11.1](#), the two layouts also clarify an easy confusion: the message separator and label hash are in

DB

before masking, not exposed as clear fields in the RSA ciphertext.

The sender converts

EM

to a nonnegative integer

m

using big-endian byte order, applies the public RSA operation, and encodes the result

C

into exactly

k

big-endian bytes. The leading zero byte in

EM

ensures that this encoded representative is below the modulus for the prescribed modulus-byte length. Repeating encryption with independently generated seeds normally gives different ciphertexts for the same message. Reusing the exact seed with the same message, label and parameters repeats the entire encoded representative and ciphertext. There is no application-supplied OAEP nonce in the library interface below: the encryption implementation obtains its seed through its cryptographic random-generation mechanism.

On recovery, the receiver first applies the specified input-length and integer-range checks, then uses the private RSA operation to obtain a

k

-byte

EM

. It separates the first byte,

S
 and
 D
 . Since
 D
 is now available, it can recover
 S
 , then recover

$$DB = D \oplus \text{MGF1}(R, k - h - 1)$$

XOR cancels the same masks, just as in Chapter 3. The decoder checks the leading zero, the expected label hash, the zero-byte padding and the 01 separator. If the required structure is absent, it returns failure rather than an application message. The order of these conceptual explanations is not a recommendation to expose separate failure reasons or timings to a caller.

What the proof does and does not establish

The security history needs care. The original generic OAEP argument did not establish the full adaptive chosen-ciphertext guarantee that was sometimes attributed to it. Later work exposed the gap, and Fujisaki, Okamoto, Pointcheval and Stern established RSA-OAEP's chosen-ciphertext security under the RSA assumption in the random-oracle model. The distinction matters: a result specific to RSA is not a result for OAEP combined with every possible trapdoor permutation. A random oracle is an idealized interface that returns consistent random outputs for new inputs; using a real hash function is the construction's implementation choice, not a proof that the real function literally is that ideal object. [\[62\]](#)

An **oracle** in a security model is a service that answers a specified type of query. A decryption oracle returns a message or the specified failure result for a submitted ciphertext. In the adaptive chosen-ciphertext experiment, an attacker can obtain decryption results for other ciphertexts, including queries made after receiving the challenge, but cannot ask for the challenge ciphertext itself. The goal is still to distinguish which of two allowed messages was encrypted. This is a stronger secrecy test than observing a single ciphertext passively. The RSA-OAEP proof relates a successful attacker in that model to RSA inversion. Its reduction is not tight: the transformed inverter can have substantially worse success and running time. Therefore we cannot read the theorem as an exact real-world attack-cost estimate for every modulus size, let alone a statement of unconditional security.

Chosen-ciphertext secrecy also does not identify the sender or make replay impossible. A second person can produce an entirely new valid OAEP ciphertext because encryption uses a public key. A receiver can accept the same original ciphertext twice unless application state prevents that. Later compromise of the long-term RSA private key can expose previously recorded ciphertexts encrypted under it; randomized encoding alone does not provide forward secrecy. These limits do not contradict the proof. They concern different goals from hiding the selected challenge message in the defined experiment.

Bleichenbacher's 1998 work provides a concrete reason to examine receiver behavior. The attack concerns the older PKCS #1 v1.5 encryption format, not the OAEP construction just described and not RSA-PSS signatures. A receiver that reveals whether a private-operation result has the required v1.5 structure can supply useful information even if it returns only a yes-or-no distinction. Adaptive queries gradually narrow the possibilities for the target. The result does not require first factoring the modulus. It demonstrates how an encoding check exposed through a protocol can matter as much as the mathematical inverse. [\[63\]](#)

OAEP implementations likewise need to avoid revealing forbidden distinctions through their decoding checks. A common error message alone does not prove this: timing, responses at later protocol stages and other observations can still differ. Selected OpenSSL code makes this implementation layer visible. Its encoder obtains a random seed, calculates

the two MGF1 masks and applies XOR; its decoder accumulates validity information and uses constant-time helpers around sensitive checks and output handling. Reading those passages connects the specification to concrete responsibilities, but is not a full audit of every backend, caller or deployment. [\[65\]](#)

An optional RSA-OAEP library example

Listing [11.1](#) uses the documented interface of `cryptography` 50.0.2. It generates one teaching key with public exponent 65537 and a 2048-bit modulus, encrypts a 32-byte generated secret, and checks recovery. SHA-256 is selected explicitly both for OAEP's hash and for MGF1. The empty label is represented by `None`. The key size demonstrates this interface and the 190-byte capacity calculation; it does not decide the lifetime, approval requirements or migration plan of a real application. The program then rejects a changed ciphertext and a ciphertext presented under the wrong label, while accepting a fresh encryption made by another caller with the same public key. [\[64\]](#)

Listing 11.1. Using a defined OAEP interface, rather than the raw RSA operation.

```
from secrets import token_bytes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa

def oaep(label=None):
    return padding.OAEP(
        mgf=padding.MGF1(hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=label,
    )

private_key = rsa.generate_private_key(
    public_exponent=65537, key_size=2048
)
public_key = private_key.public_key()
secret = token_bytes(32)
sealed = public_key.encrypt(secret, oaep())
assert len(sealed) == 256
assert private_key.decrypt(sealed, oaep()) == secret
print("32 bytes recovered from 256 ciphertext bytes")

changed = sealed[:-1] + bytes([sealed[-1] ^ 1])
for packet, label in ((changed, None), (sealed, b"other")):
    try:
        private_key.decrypt(packet, oaep(label))
    except ValueError:
        pass
```

```

else:
    raise AssertionError("expected decryption failure")
print("changed ciphertext and wrong label rejected")

other = public_key.encrypt(b"another caller", oaep())
assert private_key.decrypt(other, oaep()) == b"another caller"
print("another caller's encryption also accepted")

```

The three printed lines describe the observed interface behavior, not a complete security proof. In particular, the modified-packet check is one negative example, not a claim that every malicious input has been exhaustively tested. The last line is an intended result: the public encryption key permits another caller to create an acceptable ciphertext. The nonempty label in the negative test deliberately disagrees with the empty label used for encryption; it demonstrates parameter binding, not sender identity. The program neither stores the private key nor defines a network service, a secure transport protocol or a format for encrypting whole files.

11.6 ElGamal encryption and its fresh secret

ElGamal turns a Diffie–Hellman-style shared value into a multiplying mask for a message. The sender makes a fresh private choice for this encryption and sends two public components. The first lets the recipient calculate the same mask; the second combines that mask with the message. The recipient removes the mask with a group inverse. This description gives us the purpose of each component before we name the generator, exponents and message element in the formulas.

ElGamal encryption uses a different structure. Sam has a private exponent

x

, selected uniformly modulo the group order by the key-generation process, and publishes

$$y = g^x$$

in a chosen cyclic group

G

generated by

g

. For the modern security explanation here, take

G

to have a large prime order

q

, and require the plaintext

M

to be an element of

G

. We write the group operation multiplicatively. In the small numerical example, elements are represented as residues modulo a prime

p

; the group order

q

and modulus

p

have different roles. The original 1985 paper presented its construction using the nonzero residues modulo a prime. We identify that historical source while using the explicit prime-order-group and message-space conditions needed for the security statement below.[\[60\]](#)

To encrypt a message element

M

, the sender chooses a fresh secret exponent

r

uniformly from the residues modulo

q

and forms

(11.15)

The ciphertext is the pair

(C_1, C_2)

. The term

$$y^r$$

is a shared group value, related to Diffie–Hellman, that multiplies the message. The sender can compute it from the public value

$$y$$

and the fresh exponent

$$r$$

. The receiver can compute the same value from

$$C_1$$

and the private exponent

$$x$$

, since

$$y^{rx} = (y^x)^r = C_1^r$$

. Recovery is

$$(11.16)$$

The exponent

$$-1$$

denotes a group inverse, not subtraction of 1 or ordinary real-number division. Multiplying by the inverse cancels the shared factor. All ciphertext components and messages in this formulation belong to the selected group. Tiny groups make this computation easy for an observer too; difficulty requires an appropriate group and parameters.

A complete small-number encryption

Reuse the subgroup generated by

$$g = 2$$

modulo

$$p = 23$$

, with order

$$q = 11$$

, from Chapter [10](#). Its elements are

Let Sam's private exponent be

$$x = 6$$

, so his public value is

$$y = 8 \text{ and } 23 = 23$$

. Nora selects message element

$$M = 9$$

and, for this one teaching encryption, fresh exponent

$$r = 3$$

. The first component is

$$C_1 = 9^6 \text{ and } 23 = 23$$

. To obtain the shared factor, she calculates

The second component is therefore

$$\begin{aligned} C_2 &= 9 \times 13 \text{ mod } 23 \\ &= 117 \text{ mod } 23 = 2 \end{aligned}$$

The pair sent to Sam is

. The message element is 9; the ciphertext's second component is 2. Confusing these two values would hide the very multiplication that the example is intended to explain.

Sam calculates

$$8^6 \text{ mod } 23$$

using his private exponent. First

$$8^6 = 262144 \text{ and } 23 = 23$$

; then

$$8^6 = 262144 \text{ and } 23 = 23$$

. Consequently

$$8^6 = 8^4 \cdot 8^2 \equiv 2 \times 18 \\ \equiv 13 \pmod{23}$$

,
matching Nora's shared factor. Since

, its inverse is 16. Multiplication recovers

$$M = 2 \times 16 \pmod{23} \\ = 32 \pmod{23} = 9$$

.
Nora has not sent

, and Sam has not sent

; the transmitted first component lets him reach the matching shared factor.

For a second independent illustration of the same message, choose

$$r = 4$$

instead. The first component becomes

$$2^4 \pmod{23} = 16$$

, and the shared factor is

$$16^4 \pmod{23} = 1$$

. The second component becomes

$$9 \times 1 \pmod{23} = 9$$

. Thus

(8, 2)

and

(16, 13)

both decrypt to 9. In real use, the exponent comes from the specified secure random-generation process, rather than choosing the next small integer for convenience. Fresh sampling also does not mean that repeats are mathematically impossible. In this tiny group, repeated exponents are common enough to show why it is unsuitable for secrecy.

The security conditions and the message space

For multiplicative ElGamal in the specified group, the decision Diffie–Hellman assumption supports a chosen-plaintext secrecy argument.

Informally, the mask

$$y^r$$

should be computationally indistinguishable from an independent uniform group element when the observer sees

$$g$$

,

$$y$$

and

$$g^r$$

. Multiplying a fixed message element by an independent uniform element gives a uniform element of the same group. This explains the hiding intuition. The actual shared factor is algebraically related to the public values, so the DDH assumption is what lets the argument replace that factor with the ideal independent one. Discrete-logarithm hardness alone is not a sufficient substitute for this decision assumption. [\[7\]](#)[\[19\]](#)

The phrase *of the same group* is essential. If an application places arbitrary residues or arbitrary byte strings into a message space that the argument does not cover, the multiplying mask need not hide their structure. Zero provides an immediate counterexample: multiplying 0 by any nonzero factor still gives 0, whereas no product of nonzero group elements is 0. For a proper subgroup, an outside nonzero element can also reveal which

subgroup coset it belongs to. A coset here is the set obtained by multiplying every subgroup element by that fixed outside value. Fresh randomness cannot erase a distinction that all the allowed masks preserve.

Encoding a general file as a suitable group element is therefore not a trivial conversion of bytes to an integer. It requires a defined construction with the intended message space and security properties. Another route derives a symmetric key from appropriately processed DH material and protects the bytes with a specified authenticated-encryption construction, as in the previous chapter. That design has its own validation, derivation, context and security requirements; it is not equivalent to arbitrarily hashing one number and appending an invented check. Our ElGamal example explains the multiplicative construction, not a ready-to-deploy file format.

What happens when the fresh exponent is reused

Suppose two encryptions under the same public key reuse the same exponent

r

. Both use the same shared factor

$$Z = y^r$$

. Their second components satisfy

$$(11.17)$$

The subscripts

a

and

b

distinguish the two messages. The final equality follows because

Z

and its inverse cancel. An observer learns a relation between the plaintexts without finding Sam's private exponent. If one plaintext is known, that

relation recovers the other. The inverse operations are legitimate here because the messages and masks belong to the multiplicative group.

In our example,

$$M_a = 9$$

and reused exponent 3 give second component

$$C_{2,a} = 2$$

. If the second message is

$$M_b = 12$$

, the same mask 13 gives

$$\begin{aligned} C_{2,b} &= 12 \times 13 \bmod 23 \\ &= 18 \end{aligned}$$

The inverse of 2 modulo 23 is 12, so their ciphertext ratio is

. Knowing

, an observer recovers

. The repeated first component

makes the reuse visible in this execution. This failure is distinct from nonce reuse in an ElGamal *signature* scheme, which has different equations and consequences. We have shown plaintext-relation leakage from encryption reuse, not a general recovery of

from these two encryption pairs.

11.7 Changing a ciphertext without learning the message

Even when its randomness is correct, multiplicative ElGamal does not supply the chosen-ciphertext protection discussed for RSA-OAEP. Anyone

can take a valid pair

$$(C_1, C_2)$$

and multiply its second component by a chosen group element

$$T$$

. The result

$$(C_1, TC_2)$$

decrypts to

$$TM$$

, since the receiver cancels the same shared factor. In our example, choose

$$T = 2$$

. The ciphertext

$$(8, 2)$$

becomes

$$(8, 4)$$

, and Sam recovers

$$4 \cdot 10 \text{ and } 20 \cdot 10$$

, twice the original group element 9. The attacker has changed the plaintext in a controlled algebraic way without first learning it. This is malleability, the property introduced for other raw encryption operations earlier in the book.

ElGamal permits another transformation called **rerandomization**. Choose an exponent

$$u$$

and multiply the first component by

$$g^u$$

and the second by

$$y^u$$

:

$$(11, 18)$$

The new pair is an encryption of the same
 M
with exponent

$r + u$
 modulo
 q

. In the example, take

$u = 1$
 :
 $(8, 2)$

becomes

$(16, 13)$

, the second encryption of 9 already calculated. The receiver sees a changed ciphertext but obtains the same message. No private key was used to create this change. Some specialized protocols deliberately use such algebraic structure, but they need additional definitions and proofs suited to their goals.

Rerandomization also gives a direct reason that the bare scheme fails adaptive chosen-ciphertext secrecy. In the ideal experiment, an attacker cannot request decryption of the exact challenge. However, a distinct rerandomized pair decrypts to the same challenge message. Querying that other pair would reveal it. This reasoning does not claim that every application exposes an unrestricted decryption service; it explains why the basic construction lacks the stronger property and why simply forbidding an identical packet is insufficient. Controlled changes to the second component give related attacks when the resulting plaintext is observable.

It would therefore be misleading to describe RSA-OAEP and multiplicative ElGamal as interchangeable merely because both encrypt with a public key. We have examined a complete RSA encryption encoding with a conditional chosen-ciphertext argument, and a basic ElGamal construction with DDH-based chosen-plaintext secrecy and useful but malleable algebra. Neither name alone establishes sender identity, application authorization, freshness of a delivered instruction or safe private-key storage. A modern system selects a defined scheme or protocol with the properties it needs, instead of assuming that any reversible public-key formula provides them all.

Checking the public arithmetic in Python

Listing [11.2](#) repeats the small calculations and verifies two modifications. All its keys, messages and exponents are disclosed teaching data. Python's three-argument `pow` computes a modular power; an exponent of

-1

requests a modular inverse when one exists. The program is intentionally short because its purpose is to connect the earlier equations, not to implement OAEP, generate safe RSA primes or validate a production ElGamal group. Ordinary big-integer operations in such a script should not be treated as constant-time private-key software. [\[21\]](#)

Listing 11.2. Checking the public teaching values and their algebraic changes.

```
from math import lcm

p, q, e = 61, 53, 17
n = p * q
d = pow(e, -1, lcm(p - 1, q - 1))
c = pow(65, e, n)
assert (n, d, c, pow(c, d, n)) == (3233, 413, 2790, 65)
changed = c * pow(2, e, n) % n
assert (changed, pow(changed, d, n)) == (3017, 130)
print("RSA: 65 -> 2790 -> 65; changed result 130")

p, g, x, r, message = 23, 2, 6, 3, 9
y = pow(g, x, p)
c1 = pow(g, r, p)
c2 = message * pow(y, r, p) % p
inverse = pow(pow(c1, x, p), -1, p)
assert (y, c1, c2, inverse) == (18, 8, 2, 16)
assert c2 * inverse % p == message
assert (2 * c2) * inverse % p == 18
rerandomized = (c1 * g % p, c2 * y % p)
assert rerandomized == (16, 13)
a, b = rerandomized
assert b * pow(pow(a, x, p), -1, p) % p == message
print("ElGamal: 9 -> (8, 2) -> 9; changed result 18")
print("Rerandomized pair (16, 13) also recovers 9")
```

The chapter's central lesson is the separation between arithmetic and a security guarantee. RSA gives a public operation with a private reverse, but direct use exposes deterministic comparisons and algebraic changes. OAEP specifies how to prepare and check an RSA message, under clearly stated security assumptions. ElGamal connects encryption to Diffie–Hellman and fresh randomness, while demonstrating that hiding a message need not prevent controlled modifications. The next chapter changes the mathematical representation to elliptic curves; these distinctions between

correctness, secrecy, authentication and the surrounding protocol will remain just as necessary.

Questions to check your understanding

1. What does the name RSA stand for, and what does RSA encryption let a sender and recipient do? For

$$p = 61$$

and

$$q = 53$$

, calculate

$$n$$

and

$$\lambda(n)$$

. Why does

$$d = 413$$

reverse

$$e = 17$$

? Is

$$d$$

an ordinary real-number reciprocal?

2. Use the public powers to encrypt 65. Explain how private-operation residues 4 modulo 61 and 12 modulo 53 recombine into 65. Does the correctness argument exclude messages divisible by

$$p$$

?

3. Why does factoring a two-prime RSA modulus give a way to invert RSA? Is the converse proved? If

$$\varphi(130) = 110$$

is leaked, what are the sum and product of the factors?

4. Explain the public comparison attack against two guessed raw RSA messages. Why can multiplying a raw ciphertext by

$$2^e$$

change 65 into 130 without revealing the original message first?

5. With a 2048-bit RSA modulus and SHA-256, how many message bytes fit in OAEP? Distinguish its random seed, label, masked block

and final ciphertext.

6. Which assumptions and model qualify RSA-OAEP's chosen-ciphertext argument? Does that result mean that an accepted ciphertext identifies its sender, cannot be replayed or has forward secrecy?

7. In the prime-23 subgroup, calculate the ElGamal encryption of 9 using

$$x = 6$$

and

$$r = 3$$

. Why does recovery multiply by 16 rather than divide by 13 as a real number?

8. State the message-space and decision assumption used for the multiplicative ElGamal secrecy explanation. Why is encrypting zero outside that explanation? What relation is exposed when one exponent is reused for messages 9 and 12?

9. Starting from

$$(8, 2)$$

, explain the difference between changing the second component to 4 and rerandomizing the pair to

$$(16, 13)$$

. Why does rerandomization conflict with the chosen-ciphertext experiment?

12. Elliptic Curves

Part IV: Public Keys

When Nora and Sam agree on a secret, their public values need a mathematical setting in which the forward calculation is practical and a relevant reverse problem appears difficult. Chapter [10](#) used powers and remainders to show this arrangement. An elliptic curve provides another setting: its elements are points, and its operation combines points according to a precise rule. Nora can publish a point calculated from a private number, while Sam publishes a point calculated from his own private number. Each can then combine the other person's point with their own number to obtain matching secret material. Understanding this process begins with what a point is and what it means to add two points.

An **elliptic curve**, in the form we first study, is a collection of coordinate pairs satisfying a particular cubic equation, together with a special identity element. A coordinate pair

(x, y)

identifies a horizontal position

x

and a vertical position

y

. The word elliptic does not mean that the graph is an ellipse. More importantly, a curve used in cryptography is not a smooth drawing stored inside a computer: its coordinates belong to a finite mathematical system. We will use a real-number drawing to explain the operation, then move to a small finite example where every result can be checked by hand.

Neal Koblitz and Victor Miller independently proposed elliptic-curve approaches to public-key cryptography in the 1980s. Their contribution was a choice of mathematical groups with a promising relationship between efficient calculation and difficult reverse problems. It was not a claim that every curve or every way of using one is secure. We will separate the group operation, the computational assumptions and the software interface. This

lets us explain both a complete small calculation and the actual bytes used by X25519, the key-agreement function introduced in Chapter [10](#).^{[18][66]}

12.1 Points and the group operation

Begin with ordinary real numbers, which include whole numbers, fractions and values such as square roots. Consider the equation

$$(12.1)$$

The equation is a test for membership: substitute two coordinates and check whether the two sides agree. The point

$$P=(0,1)$$

belongs because

$$P^2=P^2=0+1=1$$

. The point

$$Q=(1,1)$$

also belongs because

$$Q^2=Q^2=1+1=1$$

. A pair does not belong merely because we can draw it on the same axes. For example,

$$(0,2)$$

fails because its left side is 4 while the right side is 1. Coordinates identify a candidate; the equation decides whether that candidate is on this curve.

For the curves in this first form, point addition has a geometric description. Draw the straight line through two different points

P

and

Q

. In the ordinary case illustrated in Figure [12.1](#), the line meets the curve at a third point

R

. Reflect

R

across the horizontal axis, changing its vertical coordinate from

$$y$$

to

$$-y$$

. The reflected point is the sum

$$P + Q$$

. Here the line through

$$P = (0, 1)$$

and

$$Q = (1, 1)$$

is

$$y = 1$$

. Substituting it in the curve equation gives

$$x^3 - x = 0$$

, whose three roots are

$$-1$$

, 0 and 1. Thus the third intersection is

$$R = (-1, 1)$$

, and reflection gives

$$P + Q = (-1, -1)$$

.

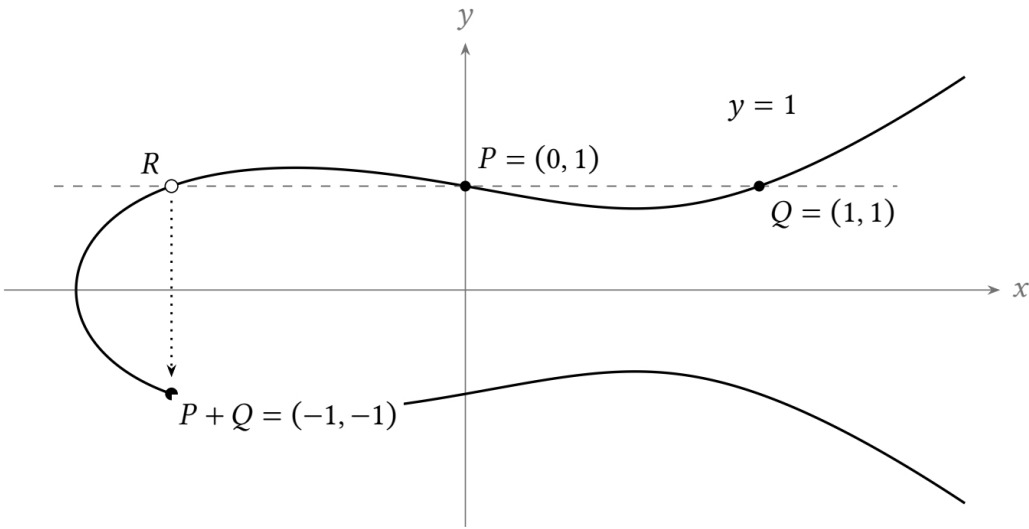


Figure 12.1. Point addition on the real curve

. The dashed chord meets

,

and

; reflecting

gives the sum. This picture explains the rule. The finite-field example later uses a different curve and has discrete points.

This operation is not coordinate-by-coordinate addition. Adding the coordinates of

P

and

Q

would give

$(1, 2)$

, which does not satisfy the equation: 4 is not equal to 1. The plus sign names the curve's specified operation, just as multiplication modulo a prime named an operation in Chapter 4. Its meaning comes from the mathematical setting. The geometric construction makes the operation visible over the real numbers; algebraic formulas will let us perform it over a finite field without drawing any lines.

To complete the operation, we need the **point at infinity**, written

$\mathcal{O}$

. It is the identity: adding it leaves another point unchanged, so

$$P + \mathcal{O} = P$$

. It is not the coordinate pair

$(0, 0)$

and does not represent an exceptionally large ordinary coordinate. For the equations considered here, the inverse of

$$P = (x, y)$$

is

$$-P = (x, -y)$$

. Adding a point to its inverse gives

$\mathcal{O}$

. In the real picture, the vertical line through these two points does not have a third finite intersection; the completed rule assigns their sum to the identity. This is how the operation remains defined in a case where the simple three-intersection picture would otherwise leave a gap.

We also need to add a point to itself. This is called **doubling**, written

$$2P = P + P$$

. Instead of a line through two distinct points, the geometric rule uses the tangent at

P

: the line with the curve's local slope there. We will supply its slope formula, so no knowledge of calculus is needed to follow the numerical example. If a point has vertical coordinate zero, it equals its own inverse and doubles to

$\mathcal{O}$

. These special cases are part of the operation, not failures that a program may ignore.

With the required smoothness condition, the curve points and

$\mathcal{O}$

form an abelian group. In beginner terms, the operation stays within the allowed set; it has an identity and inverses; regrouping additions does not change the result; and exchanging their order does not change the result. The last property is commutativity, expressed as

$$P + Q = Q + P$$

. Associativity,

$$(P + Q) + R = P + (Q + R)$$

, is particularly important when we repeatedly double and add. It is a mathematical property of the curve law, not something established by a single drawing. These properties give repeated point addition the consistent meaning needed for public-key calculations. [\[18\]](#)

12.2 From a real curve to a finite field

A digital key exchange uses a finite set of values rather than arbitrary real coordinates. We will work in the prime field

F_{17}

, whose elements are represented by the integers from 0 to 16. Its addition, subtraction and multiplication keep remainders modulo 17. Every nonzero element has a multiplicative inverse, so division by a nonzero value also has a defined meaning. This is the prime-field arithmetic developed in Chapter 4. It differs from the polynomial-based byte field used inside AES, even though both systems are finite fields.

Our teaching curve is

(12.2)

The letter

E

names the curve. The symbol

$\equiv$

says that the two sides have the same remainder on division by 17. Both coordinates are field elements, so we choose their representatives from 0 through 16. For

$P=(5,1)$

, the left side is 1 and the right side is

$$125 + 10 + 2 = 137$$

$$= 8 \times 17 + 1$$

The equal remainders put

on the curve. For

, the two sides are 9 and

$$216 + 12 + 2 = 230$$

$$= 13 \times 17 + 9$$

,

so

also belongs.

This curve uses the **short Weierstrass form**, a standard name for equations

$$y^2 = x^3 + ax + b$$

. The letters

a

and

b

are fixed coefficients specifying the curve; they are not private keys. For a prime field with

$$p > 3$$

, the required nonsingularity condition in this form is

$$(12.3)$$

The condition excludes degenerate equations for which the usual elliptic-curve group law does not apply as stated. Here

$$p = 17$$

and

$$a = b = 2$$

, so

$$4 \times 8 \times 17 \times 2 = 1088$$

has remainder 4, not zero. The condition is satisfied. It establishes that we have the intended kind of curve; it does not establish cryptographic security. Our tiny field can be searched completely, as the next figure makes clear.[\[68\]](#)

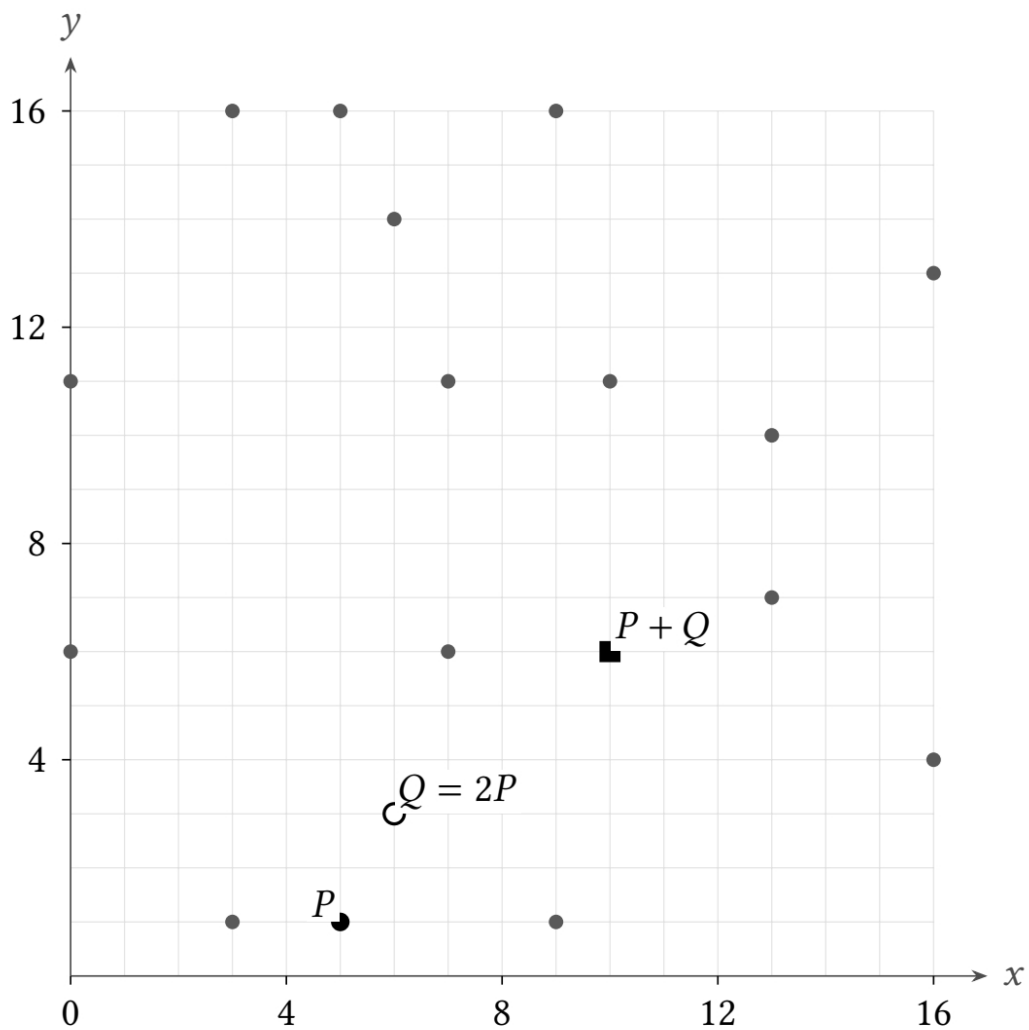


Figure 12.2. All 18 coordinate pairs on

. Including

, which has no coordinate position on the grid, gives 19 group elements. The emphasized points will satisfy

and

. No continuous line connects the finite-field points.

The grid in Figure [12.2](#) contains all possible horizontal and vertical representatives. Testing the equation finds 18 permitted pairs. Adding

$\mathcal{O}$

gives 19 group elements. The number 17 counts the values available to each coordinate; 19 counts the curve's group elements. These are different quantities. A field with 17 elements does not require a curve group with 17 elements, just as a grid with 289 candidate positions does not require every position to satisfy the equation.

The inverse of

$$P = (5, 1)$$

is now

$$(5, 16)$$

because

$$-1$$

has remainder 16 modulo 17. Their vertical coordinates add to zero in the field. More generally, the negative of

$$(x, y)$$

is

$$(x, -y \bmod 17)$$

. The plot therefore has paired vertical coordinates that sum to 17 whenever they are nonzero. This is a modular symmetry, not ordinary reflection about the horizontal axis at zero in the displayed grid. Keeping the arithmetic and the picture separate prevents a misleading attempt to draw real-number chords between these finite points and read off their sums.

Division also changes its practical appearance. In this field, dividing by 5 means multiplying by its inverse 7, since

$$5 \times 7 = 35 = 2 \times 17 + 1$$

. The ordinary fraction

$$1/5 = 0.2$$

is not one of our field representatives. Similarly, the inverse of 2 is 9 because

$$2 \times 9 = 18$$

has remainder 1. We will use this second inverse in doubling. A denominator of zero has no inverse, which is why the special identity and inverse cases must be handled before applying a slope formula.

12.3 Adding and doubling points

Let

$$P = (x_1, y_1)$$

and

$$Q = (x_2, y_2)$$

be points on a short Weierstrass curve over our prime field. The subscripts distinguish the first point's coordinates from the second point's coordinates; they are not powers. When

$$x_1 \neq x_2$$

, define the slope value

$$S$$

, then calculate the sum

$$(x_3, y_3)$$

:

$$(12.4), (12.5), (12.6)$$

First find the change in vertical coordinate and multiply it by the inverse of the change in horizontal coordinate. This gives the field version of a slope. Next square that result and subtract the two original horizontal coordinates. Finally use the new horizontal coordinate in the third formula to obtain the vertical coordinate of the reflected intersection. Every subtraction and multiplication uses the same field. The exponent

$$-1$$

means multiplicative inverse, not a negative coordinate or an ordinary decimal reciprocal.

Apply the formulas to

$$P = (5, 1)$$

and

$$Q = (6, 3)$$

on our prime-17 curve. Their horizontal difference is 1, whose inverse is 1, so

$$s = (3 - 1) \cdot 1 = 2$$

. The coordinates are then

For the first remainder, adding 17 to
 -7
 gives 10; for the second, adding 17 to
 -11
 gives 6. Thus

$$P + Q = (38, 6)$$

. We can check membership independently:

$$6^2 = 36$$

has remainder 2, while

$$38^2 = 1444$$

also has remainder 2. The calculation has produced another point on the same curve.

For doubling, the two points coincide and the preceding denominator would be zero. When

$$P = (x_1, y_1)$$

has

$$y_1 \neq 0$$

, use the tangent slope instead:

$$(12.7)$$

The coefficient

a

comes from the curve equation. Square the horizontal coordinate, multiply it by 3 and add

a

; then multiply by the inverse of twice the vertical coordinate. Having obtained

s

, use the same coordinate rules as before with

$$x_2 = x_1$$

. The tangent formula is different from the distinct-point slope formula, although the final coordinate calculation has the same pattern.

For

$$P = (5, 1)$$

, the numerator is

$$3 \times 25 + 2 = 77$$

, and the denominator is 2. Its inverse is 9, giving

$$\begin{aligned} s &= 77 \times 9 \bmod 17 \\ &= 693 \bmod 17 = 13 \end{aligned}$$

Therefore

We obtain

$$2P = (6, 3)$$

, the point previously called

Q

. The earlier addition therefore also shows

$$3P = P + 2P = (3, 6)$$

. Repeated addition is beginning to give us a sequence of points from one starting point, which is the operation a private number will control.

There are three exceptional cases to settle before these formulas: an input equal to

$\mathcal{O}$

, two inputs that are inverses, and doubling a point whose vertical coordinate is zero. Respectively, the result is the other input,

$\mathcal{O}$

, or

$\mathcal{O}$

. On an odd-prime curve, two valid points with the same horizontal coordinate are equal or inverses. Once those cases are distinguished, the relevant nonzero denominator has an inverse. A generic program that blindly divides without checking these conditions is not an implementation of the complete group operation. [\[68\]](#)

The coordinate formulas shown here are called **affine** formulas: they use the ordinary pair

(x, y)

. Efficient implementations often use additional coordinates that represent the same point and postpone expensive inversions. Those internal representations do not change the group result. They can, however, change the sequence of machine operations, exceptional cases and resistance to information leaks. A short teaching implementation is valuable because its arithmetic is visible; that visibility does not make it a suitable implementation for secret production keys.

12.4 Scalar multiplication and its reverse problem

A **scalar** here is an integer used to count repeated point addition. The expression

dP

, called scalar multiplication, means add

P

to itself

d

times for a nonnegative integer

d

. It does not mean multiply both coordinates by

d

. We define

$0P = \mathcal{O}$

, and negative scalars can be understood using inverse points. For example,

$$2P$$

is the doubling already calculated, not

$$(10, 2)$$

. The name multiplication refers to the integer's repeated action on a group element.

Successive additions eventually return to

$$\mathcal{O}$$

in a finite group. The **order of a point** is the smallest positive integer

$$n$$

for which

$$nP = \mathcal{O}$$

. For our

$$P = (5, 1)$$

, that integer is 19. Its multiples reach every group element, so it generates the entire 19-element group in this particular example. In general, a chosen base point can generate only a subgroup. We call the full group size

$$N$$

, the chosen point's order

$$n$$

, and the ratio

$$h = N/n$$

the **cofactor**. Here

$$N = n = 19$$

and

$$h = 1$$

. Other curves can have a larger cofactor or a group that is not cyclic.

Table 12.1. Selected multiples of

on the teaching curve. Doubling builds powers of two; the last two rows show a combined scalar and the return to the identity.

Scala	Result	How to obtain it
r	t	The chosen base point.
1		
Scala	Result	How to obtain it
r	t	Double
2		

Scalar	Result	How to obtain it
4	16	Double 4
8	64	Double 16
13	169	Add 128 and 41
19	361	The first positive return to the identity.

Calculating a large multiple does not require one addition for every unit of the scalar. Write the scalar in binary and use doubling to build powers of two, as repeated squaring did for exponents. For

, Table 12.1 supplies

P

,

$4P$

and

$8P$

. Adding

$8P + 4P$

gives

$12P = (10, 11)$

; adding

P

gives

$13P = (10, 4)$

. Associativity ensures that the grouping does not change the result. The number of doubling stages grows with the scalar's bit length, making the forward operation practical even when the possible scalars form an enormous set.

The order

n

also tells us how scalar actions repeat:

$$(d + n)P = dP$$

. Thus scalars acting on this subgroup can be reduced modulo 19, while each coordinate calculation still uses modulus 17. Confusing these two moduli would give incorrect results. For instance,

$$35P = 16P$$

because

$$35 = 19 + 16$$

, not because we reduced 35 modulo 17. A curve description, its field modulus and its base point's order jointly define the calculation. A phrase such as a 256-bit curve does not supply all these details or establish a security level by itself.

To see elliptic-curve Diffie–Hellman, let Nora choose private scalar 5 and Sam choose private scalar 7. Their public points are

$$A = 5P = (9, 10)$$

and

$$B = 7P = (8, 6)$$

. Nora calculates

$$5B$$

and Sam calculates

$$7A$$

. Their results agree because

$$(12.8)$$

Multiplying the private scalars gives 35; reducing that scalar modulo the point order gives 16; and

$$16P = (16, 11)$$

. Each participant can obtain the result using their own scalar and the received public point. An observer sees

P

,

A

and

B

, but our tiny example offers no secrecy: listing at most 19 multiples reveals both private scalars. The calculation explains agreement, not a secure parameter choice.

The **elliptic-curve discrete logarithm problem** asks for

d

when

P

and

$$Q = dP$$

are known, with the subgroup specified. The word logarithm describes this reverse relationship; it does not require taking a decimal calculator logarithm of either coordinate. For suitable large prime-order subgroups, no efficient classical algorithm is known for this task. Generic square-root attacks require roughly

$\sqrt{n}$

group operations, up to algorithmic and model qualifications. Here

$\sqrt{n}$

means the positive number whose square is

n

. This is a benchmark, not a proof that all curves resist every attack. Curves with special weaknesses, incorrect subgroup handling or a leaking implementation can invalidate a simple estimate. [\[7\]](#)

The **computational Diffie–Hellman problem** instead asks for

$$abP$$

given

P

,

aP

and

bP

. Recovering either scalar would let an attacker solve it, but a general equivalence between these two problems is not being assumed. The confidentiality claim of a complete protocol must identify the problem and model it needs. Authentication is also separate: an active intermediary can substitute curve public values just as in Chapter [10](#). Finally, the familiar classical assumptions are not post-quantum guarantees. Shor's algorithm addresses these discrete logarithms in its quantum-computing model; that theoretical result is different from a claim about what an available machine can do today. [\[18\]\[66\]\[22\]](#)

Optional calculation: repeat the small example

The following Python program implements only our public teaching curve. The value `None` represents

$\mathcal{O}$

. The addition function checks the identity and inverse cases before finding a modular inverse. The multiplication function follows the scalar's binary digits by repeated halving, doubling the current point at each stage. Its branches depend on the scalar; this is not constant-time code for a secret key. Keeping that limitation explicit lets us use the program to check arithmetic without presenting it as a cryptographic library.

Listing 12.1. Public arithmetic on the prime-17 teaching curve.

```
p, a = 17, 2
O = None

def add(P, Q):
    if P is O:
        return Q
    if Q is O:
        return P
    x1, y1 = P
    x2, y2 = Q
    if x1 == x2 and (y1 + y2) % p == 0:
        return O
    if P == Q:
        s = (3*x1*x1 + a) * pow(2*y1, -1, p) % p
    else:
```

```

        s = (y2-y1) * pow((x2-x1) % p, -1, p) % p
        x3 = (s*s - x1 - x2) % p
        y3 = (s*(x1-x3) - y1) % p
        return x3, y3

def mul(d, P):
    if d < 0:
        raise ValueError("use a nonnegative teaching scalar")
    result = O
    while d:
        if d & 1:
            result = add(result, P)
        P = add(P, P)
        d >>= 1
    return result

P = (5, 1)
assert mul(2, P) == (6, 3)
assert add(P, mul(2, P)) == (10, 6)
assert mul(13, P) == (16, 4)
assert mul(19, P) is O
A, B = mul(5, P), mul(7, P)
assert (A, B) == ((9, 16), (0, 6))
assert mul(5, B) == mul(7, A) == (10, 11)
print("2P = (6, 3); 13P = (16, 4); 19P = O")
print("public points (9, 16), (0, 6); shared (10, 11)")

```

The function assumes that every supplied pair is already a valid point on this fixed curve, represented canonically. It is not a received-key parser or a validation routine. The assertions check the stated examples, while a separate exhaustive calculation can check all 19 group elements. Even exhaustive checking on this tiny set cannot establish the security of a much larger curve, its optimized code or a protocol that uses it.

12.5 X25519 and modern key agreement

X25519 is a specified function for elliptic-curve Diffie–Hellman using Curve25519. The name of the function and the name of the curve are related but not interchangeable: the function also defines input bytes, scalar processing, coordinate decoding and output bytes. Curve25519 uses a different equation form from our small example, called a **Montgomery form**. With coordinates named

u

and

v

, its equation is

(12.9)

The letters

u

and

v

play the roles of horizontal and vertical coordinates in this form. The number 486662 is a public coefficient specifying the curve. The prime

p

specifies the coordinate field, and

2^{255}

means 2 multiplied by itself 255 times. Changing the equation form changes the calculation formulas; the short Weierstrass slopes above must not be copied directly into this function.

The full curve group has cofactor 8. Its standard base point has

u

-coordinate 9 and prime order

(12.10)

The letter

ℓ

, read as a lowercase L, names the order of the chosen prime-order subgroup. The full curve has

8ℓ

elements. Again,

p

is the coordinate modulus and

ℓ

is the subgroup order; they are not the same number. The large decimal term is part of the specified order, not a value a reader needs to memorize or a private setting chosen by a user. These fixed parameters let independent implementations agree on exactly which calculation they are performing. [\[58\]](#)

At the byte interface, X25519 takes a 32-byte scalar input and a 32-byte encoded

u

-coordinate. It produces a 32-byte encoded

u

-coordinate as output. It does not require a transmitted

v

-coordinate. To create a public value, use the private input with the encoded base coordinate 9. To perform the exchange, use the same private input with the peer's public value. The resulting raw shared bytes are then processed as specified by the surrounding protocol, normally through an appropriate key derivation step. A public value and a raw shared result happen to have the same length, but their roles are different.

The field coordinate is encoded in **little-endian** order: the first byte contributes the least significant eight bits. The base input therefore begins with byte 09, followed by 31 zero bytes. Before interpreting a received X25519 coordinate, the specified decoding masks the highest bit of its last byte. It also accepts field representatives at least

p

and reduces them modulo

p

. For example, 9 and

$$p + 9$$

represent the same field element when encoded within the permitted input range. A second representation of the same field element does not create a different mathematical coordinate. The consequences for identifiers and protocol context will be important in the next section.

Private scalar input also has specified processing. Number the bytes starting from zero. Clear the lowest three bits of byte 0; clear the highest bit

of byte 31; and set that byte's next-highest bit. This operation is often called **clamping**. The resulting integer has the form

$$(12.11)$$

The fixed bit contributes

$$2^{254}$$

, the three zero low bits make the number a multiple of 8, and the remaining 251 free bits specify the integer

t

. This is a description of X25519's scalar decoding, not a password-strengthening step. A maintained library performs it internally. Choosing a private input still requires suitable unpredictable generation; clamping a predictable input does not create an unpredictable secret. Nor should a private key be reused across unrelated curve functions merely because their byte lengths match.

X25519 can calculate with only the

u

-coordinate by using a **Montgomery ladder**. Conceptually, it maintains two neighboring multiples,

$$rP$$

and

$$(r+1)P$$

, while reading the scalar's bits. The next bit determines which neighboring pair is required after a doubling stage. The defined arithmetic updates both sides in a regular pattern and uses conditional swaps to place the required values in the appropriate positions. Internally, a coordinate can be represented as a ratio

$$u = X/Z$$

, letting the calculation postpone division until the end. These extra coordinates are a representation technique; they do not mean that a public key transmits two independent coordinates.

This regular pattern helps implementers avoid a simple leak in which one secret bit performs noticeably different work from another. It does not prove that every implementation is constant time. A conditional swap needs a suitable implementation, field arithmetic must avoid secret-dependent behavior, and the compiler and platform can matter. The variable-time Python loop above was chosen for visible arithmetic and has a different purpose. The ladder specified in RFC 7748 is a design for implementing the defined function, not permission to infer timing security from a few matching outputs. [\[58\]](#)

The 32-byte public representation also does not imply 256-bit attack resistance. Its size describes an encoding. RFC 7748 places Curve25519's classical security slightly below the conventional 128-bit level, with qualifications about comparing costs across primitive types. That estimate depends on the mathematical problem and attacker model, rather than the number of characters printed in hexadecimal. Likewise, good mathematical parameters do not establish that an application authenticates a peer, protects its private input or handles an invalid exchange safely. These remain separate requirements.

Optional program: a published exchange

The following program uses the RFC 7748 Section 6.1 private inputs, which are public test values and must never protect real data. The RFC names its participants Alice and Bob; their first and second roles correspond to Nora and Sam here. Each hexadecimal string is split across adjacent Python string literals for readability; the strings join before conversion to bytes. The expected public values and shared result come from the same published example. In addition to matching these bytes, the program derives equal purpose-specific keys using HKDF-SHA-256. Its context contains a fixed purpose label followed by both fixed-length public values in a specified order. [\[58\]](#)[\[69\]](#)

Listing 12.2. RFC 7748 test values through the documented X25519 interface.

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x25519 import (
    X25519PrivateKey,
)
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
```

```

first = X25519PrivateKey.from_private_bytes(bytes.fromhex(
    "77076d0a7318a57d3c16c17251b26645"
    "df4c2f87ebc0992ab177fba51db92c2a"
))
second = X25519PrivateKey.from_private_bytes(bytes.fromhex(
    "5dab087e624a8a4b79e17f8b83800ee66"
    "f3bb1292618b6fd1c2f8b27ff88e0eb"
))
first_public = first.public_key()
second_public = second.public_key()
A = first_public.public_bytes_raw()
B = second_public.public_bytes_raw()
assert A.hex() == (
    "8520f0098930a754748b7ddcb43ef75a0"
    "dbf3a0d26381af4eba4a98eaa9b4e6a"
)
assert B.hex() == (
    "de9edb7d7b7dc1b4d35b61c2ece43537"
    "3f8343c85b78674dadfc7e146f882b4f"
)
z_first = first.exchange(second_public)
z_second = second.exchange(first_public)
assert z_first == z_second == bytes.fromhex(
    "4a5d9d5ba4ce2de1728e3bf480350f25e"
    "07e21c947d19e3376f09b3c1e161742"
)
info = b"book-example: first-to-second key" + A + B

def derive(z):
    return HKDF(
        algorithm=hashes.SHA256(), length=32,
        salt=None, info=info,
    ).derive(z)

assert derive(z_first) == derive(z_second)
print("published public values and shared result matched")
print("matching purpose-specific keys derived")

```

The private test inputs are fixed only so that another reader can reproduce the example. A real ephemeral exchange generates fresh private material using the library's documented generation interface. A protocol must also define the required authentication, allowed algorithms, key derivation, directional keys and message protection. The purpose label above explains context binding; it is not a complete handshake specification. An attacker who substitutes their own public values can still establish different shared secrets with the participants unless the protocol checks the required identities or authenticated key relationships. Equal derived keys establish agreement within the demonstrated calculation, not the identity of the other party.

The key derivation step does not add a fresh private contribution or make a known shared result secret. Its job here is to prepare suitable existing

material and distinguish the intended key purpose. Both public encodings have fixed length, and their first/second order is explicit, so their concatenation is unambiguous in this example. Variable-length fields in a larger protocol need specified framing. If a protocol uses different raw encodings of equivalent public values, it must also specify how those encodings enter its transcript and derivation; mathematical equivalence alone does not make every byte-level protocol behavior equivalent.

12.6 Encodings, small-order inputs and validation

A network transports bytes, not abstract points. **Encoding** specifies how a mathematical value becomes bytes; **decoding** interprets received bytes under that format. For a full short Weierstrass point, a common SEC 1 uncompressed format contains a prefix and both coordinates. Its compressed format contains a different prefix, the horizontal coordinate and a choice between the two possible vertical coordinates. On our odd-prime teaching curve,

$(5, 1)$

has compressed hexadecimal encoding `03 05`; its inverse

$(5, 16)$

has `02 05`. The prefix records odd or even vertical parity. The uncompressed encoding of

$(5, 1)$

is `04 05 01`. These examples use one byte per coordinate because the field is tiny; real formats use their specified fixed coordinate length and byte order.

X25519 uses another format entirely: 32 little-endian bytes for a

u

-coordinate, with no SEC 1 prefix or vertical-parity bit. It is therefore misleading to call every 32-byte curve value a compressed SEC 1 public key. Compression of a full point retains enough information to select its vertical coordinate. X25519 intentionally works through its specified coordinate interface instead. Prefix rules, byte order, lengths and acceptable inputs must come from the selected construction. A successful conversion from bytes to an object in a programming interface does not establish that every subsequent security requirement has been satisfied.

For a full-point scheme using validated short Weierstrass domain parameters, SEC 1's full public-key validation checks that the point is not

$\mathcal{O}$

, that its coordinates are in the specified range, that it satisfies the equation, and that multiplying it by the specified prime subgroup order gives

$\mathcal{O}$

. The domain parameters specify the field, curve, base point and its order; they also need their own appropriate validation. Checking a received point does not repair an unsuitable parameter set. When the full group has prime order and cofactor 1, a nonidentity curve point already belongs to that group, so the subgroup reasoning simplifies. [\[68\]](#)

These checks have different purposes. An equation check distinguishes points on the selected curve from arbitrary coordinate pairs. An order check addresses membership in the intended subgroup. Rejecting the identity prevents a trivial public contribution. Range and encoding checks establish the required representation. None proves that a named person owns the key. Conversely, replacing all these mathematical checks with a length check would leave important conditions untested in a scheme that requires full-point validation. The appropriate procedure depends on the construction, rather than on a universal rule to accept or reject everything described as a curve point.

X25519's rules deliberately differ from that full-point checklist. The specified function masks its input's highest bit and processes noncanonical field encodings as reduced values. Its coordinate arithmetic also handles inputs associated with the curve's **quadratic twist**. This is a related curve with a different set of full points, for which the same

$\mathcal{U}$

-coordinate ladder formulas can also be used. A reader need not calculate a twist to understand the interface distinction: a received

$\mathcal{U}$

-coordinate need not decode to a full point on the original equation before X25519's specified calculation is applied. Generic full-point code may impose different acceptance rules and therefore is not automatically a conforming replacement for the X25519 interface. [\[58\]](#)

The cofactor introduces a further issue. A **small-order point** returns to the identity after very few additions. Because the decoded X25519 scalar is divisible by 8, small-order inputs can eliminate the private scalar's contribution and produce an all-zero raw result. This is a failure of the contribution expected by many key-agreement uses. Passing that predictable result through HKDF would not recreate a secret. RFC 7748 describes an optional all-zero-output check; a surrounding protocol can require rejection, and the library used for the examples rejects such an exchange. We must distinguish the RFC's function and key-exchange discussion from a particular protocol's mandatory acceptance rule.

For example, the installed library can construct a public-key object from 32 zero bytes because the length is permitted. Calling `exchange` with that object raises `ValueError` rather than returning an accepted all-zero shared result. An input of 31 bytes instead fails the public-key constructor's length requirement. These failures occur at different stages. An application must handle the appropriate failure without treating missing shared material as a successful agreement. The all-zero check's implementation should also avoid revealing extra secret-dependent information through its method of comparison.[\[69\]](#)[\[70\]](#)

Different received encodings can also yield the same raw X25519 result. The masked highest bit and reduction of a noncanonical coordinate provide simple examples. Other equivalences arise from the mathematical setting. This is why a raw public value cannot automatically serve as a unique identity with proof of ownership inferred solely from knowledge of a shared result. A defined protocol binds the appropriate public values, roles, identities and context in its authenticated exchange and key schedule. Binding bytes in a derivation is useful, but it does not itself authenticate the party that supplied those bytes.

Encoding a point is different from hashing to a curve

There is one more operation whose name can cause confusion. Serializing a point begins with an existing mathematical point and represents it as bytes. **Hashing to a curve** begins with an arbitrary byte string and produces a point using a specified construction. The input might be an identifier or message required by a larger protocol. This is neither

ordinary point compression nor a reversible way to encrypt a file. Producing a point from a message does not conceal the message from an observer who can run the same public procedure.

RFC 9380 specifies hash-to-curve constructions with named suites, domain separation and subgroup handling. Its distinction between `hash_to_curve` and `encode_to_curve` concerns their required output-distribution properties; neither should be replaced by an improvised recipe that hashes a message and guesses successive coordinates. The required distribution, constant-time behavior and cofactor processing depend on the specified construction. We will return to specialized uses when the surrounding protocols need them. For now, the practical lesson is to identify the operation before borrowing a byte format or a mathematical check from a different one. [\[67\]](#)

Elliptic curves therefore give us more than a compact drawing or a shorter key. They supply a precise group operation, efficient scalar multiplication and computational questions that support carefully specified schemes. The small example made every point operation visible; X25519 showed how a real function defines its inputs and handles them differently from a generic full-point interface. Neither layer removes the need for key derivation, authentication or sound implementation. Those boundaries will help us understand signatures next, where a public point has a different role: checking a statement made using the corresponding private key.

Questions to check your understanding

1. What does a coordinate pair mean, and how do we check whether it lies on a curve? On the real curve in Figure [12.1](#), why is

$$(0,0), (0,1), (1,0)$$

rather than

$$(1,2)$$

?

2. In the prime-17 example, distinguish the field size, the number of affine coordinate pairs and the full group order. What is

$$-(5,1)$$

, and why is

$$\mathcal{O}$$

not

$$(0,0)$$

?

3. Calculate

$$(5,1) + (6,3)$$

using the distinct-point formulas. Then double

$$(5,1)$$

, explaining why the inverse of 2 is 9. Which cases must be handled before taking an inverse?

4. Use

$$13 = 8 + 4 + 1$$

to explain scalar multiplication. Why are coordinates reduced modulo 17 while multiples of the base point repeat modulo 19? What is the cofactor here?

5. With private scalars 5 and 7, identify the two public points and the shared point. Distinguish recovering a scalar from computing a shared point given the two public points.
6. What does each of X25519's two inputs represent? Explain its 32-byte output, base input, little-endian order and scalar clamping. Does the output length imply 256-bit security?

7. Why are SEC 1 compressed points and X25519 public values different formats? Does a full-point validation checklist apply unchanged to X25519?
8. Why can an all-zero raw shared result be unacceptable? Distinguish the RFC's optional check, the chosen library's rejection and a protocol's requirements. Can HKDF turn a publicly predictable result into a secret?
9. Explain why matching X25519 results do not authenticate a participant. How do point serialization and hashing to a curve differ?

13. Signatures, Certificates and Identity

Part IV: Public Keys

Nora publishes a software update that anyone may download. Sam wants to check that the file matches a release approved under Nora's signing key, even if a different website supplies the download. Encrypting the file would answer a different question: it would limit who can read it. A shared-key authentication code would let Sam check a tag, but anyone given the checking key could also create tags. Nora needs a method that separates the ability to approve a message from the ability to check that approval. A digital signature provides this separation through a private signing key and a corresponding public verification key.

A **digital signature** is verification data calculated from a message and a private signing key under a specified scheme. The message may be a document, a program or a protocol record. The signature is a sequence of bytes, not a picture of handwriting. A verifier checks the message and signature together using the corresponding public key. Passing that check connects the exact signed data to the key under the scheme's security assumptions. Connecting the key to Nora, deciding what her approval means and checking whether the release is current require additional information.

We will first explain signing, verification and forgery without assuming earlier experience with signatures. A small Schnorr calculation will make the central algebra visible. We will then compare RSA-PSS, ECDSA and EdDSA, examine the consequences of reused signing secrets, and explain how certificates connect public keys to named subjects. These layers are related, but each answers a different question. Keeping them clear is essential when a signature becomes part of a payment system, software update or secure connection.

13.1 Signing and verification

A signature scheme has three algorithms: key generation creates related signing and verification keys; signing calculates a signature using the private key; and verification checks a proposed signature using the public key. Let

m

denote the exact message bytes,

sk

the secret signing key,

pk

the public verification key and

σ

the signature. The Greek letter

σ

, read as sigma, simply names those signature bytes. We describe the intended operations as

(13.1)

Read the first expression as: the signing operation takes the private key and message and produces a signature. Read the second as: the verification operation takes the public key, message and proposed signature and gives an acceptance or rejection result. The notation hides scheme-specific parameters and, where required, fresh randomness. Those inputs remain part of the scheme even when this introductory interface does not display them.

Correctness requires signatures produced by the prescribed signing algorithm on allowed inputs to pass the matching verification algorithm. Security asks a separate question: can an attacker create an accepted signature for a message that the legitimate signing process did not sign? The attacker should not need to recover the private key to count as successful. If a weakness lets them create just one accepted new message, the signature's purpose can already fail. This is why describing a signature only as a difficult private-key calculation is incomplete.

The standard **chosen-message** setting gives the attacker access to genuine signatures on messages they select. They may choose the next message after examining earlier signatures. A successful forgery is an accepted signature on a message that was not among their signing requests. The message need not be meaningful or valuable for this mathematical security goal. Preventing all such forgeries also protects against the attacker finding one application-specific message that matters. Goldwasser, Micali and Rivest's 1988 work made this adaptive attack setting central to the scientific treatment of signatures; the historical construction is distinct from the schemes we use below.^[71]

Ordinary unforgeability concerns a new message. A stronger definition can also prohibit producing a new accepted signature for a previously signed message. This distinction matters because some schemes permit more than one representation or valid signature for the same message. Replaying the exact old message and signature is a different issue again: signature verification normally accepts them again. An application that must process an instruction only once needs a signed identifier, sequence number or other defined freshness mechanism and state to enforce it.

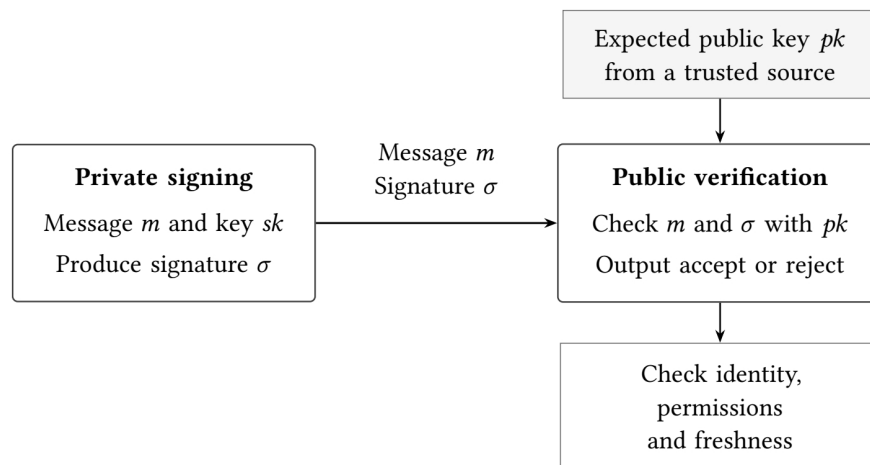


Figure 13.1. The private signing operation produces a signature. The public verification operation checks the received message and signature together. Trust in the public key and the application's acceptance decision are separate inputs and steps. The message is not concealed by this process.

Figure [13.1](#) also shows why a signature is not encryption with a private key. Verification generally needs the message as a separate input; it does not decrypt an arbitrary hidden document. Although an RSA primitive can use

the same modular-power machinery for different purposes, RSA encryption and signatures specify different encodings and checks. Other signature schemes do not even have a corresponding message-decryption operation. The correct description follows the scheme's interface and goal rather than an apparent symmetry in one algorithm.

The exact signed representation is important. A changed space, line ending or character encoding can change the message bytes even when a screen display looks similar. For a structured instruction, the signed data should identify the operation, recipient, amount, units and relevant context under a specified encoding. If two components interpret those bytes differently, a valid signature cannot decide which interpretation is intended. **Canonical encoding** means choosing one specified representation for a value when that is required; alternatively, a protocol can define exactly which received bytes are signed. Either approach needs clear parsing rules.

Hash functions commonly help process a long message, but a public digest is not itself a signature. Anyone can calculate a digest of changed data. In a hash-based signature calculation, the signing relation must protect the appropriate digest and its context. If an attacker can prepare two different messages with the same relevant digest, a signature obtained on one may also fit the other. Collision resistance therefore matters in constructions that rely on it, alongside the separate public-key assumption. A signature format must also specify the hash and algorithm parameters rather than allowing an attacker to reinterpret the same bytes under a different operation.

13.2 A small Schnorr example

Schnorr signatures, named after Claus-Peter Schnorr, connect a secret exponent to a short public verification equation. We will use a positive-public-key convention:

$y = g^x \bmod p$
 , where
 x

is the private scalar. Other descriptions can use inverse public keys or a different sign in the response; their formulas must be changed consistently. Our example illustrates the algebra with a deliberately tiny subgroup and a

defined teaching hash. It is not a standardized signature format or a secure parameter choice. [\[72\]](#)

Use the subgroup generated by

$$g = 2$$

modulo the prime

$$p = 23$$

. Its order is

$$q = 11$$

: powers of 2 first return to 1 after eleven positive steps. The private scalar

$$x = 6$$

gives public value

$$y = 2^6 \text{ mod } 23 = 16$$

. As before, there are two moduli. Products of represented group elements are reduced modulo 23; scalar calculations repeat modulo subgroup order 11. The verifier knows

$$p$$

,

$$q$$

,

$$g$$

,

$$y$$

and the chosen message representation. Only

$$x$$

and the per-signature secret introduced next must remain private in a real construction.

To sign a message, choose a fresh per-signature secret

$$k$$

and calculate a public commitment

$$R = g^k \text{ mod } p$$

. Here a **commitment** means the initial public group value used by this signing calculation; we are not claiming every property of a separate

cryptographic commitment scheme. Hash the public key, commitment and message under the prescribed encoding to obtain a challenge scalar

e

. Then calculate the response scalar

s

:

(13.2)

First, raising

g

to the secret

k

produces the public value

R

. Second, the hash challenge connects that value and the public key to the exact message. Third, multiply the challenge by the private scalar, add the per-signature secret and keep the remainder modulo

q

. The signature in this teaching presentation is the pair

(R, s)

. The challenge

e

is recomputed by the verifier rather than trusted as an independent statement from the signer.

For reproducibility, define

H_{toy}

as SHA-256 applied to a fixed sequence of bytes, interpreted as a big-endian integer and reduced modulo 11. The sequence starts with the ASCII label `book-schnorr-v1`, followed by one-byte

y

, one-byte

R

, a two-byte big-endian message length, and the message itself. The one-byte choices fit only our tiny field, and the length field limits this teaching interface to 65,535 message bytes. This framing removes ambiguity between inputs. Reducing a digest to eleven values makes the challenge easy to search; it provides no meaningful security.

Take

m

to be the six ASCII bytes `pay 10`, and choose

$k = 3$

. Then

$e = 9$

. The specified teaching hash gives

$e = 9$

, as the optional program below checks. The response is

(13.3)

The signature is therefore

(8, 2)

. The number 3 is not sent as part of it. Knowing

$R = 8$

should not reveal

k

in a suitable large group, but an observer can find it immediately in this tiny one by trying the eleven possible scalars.

Verification checks the required input ranges and subgroup membership, recomputes

e

from the received message and then tests

(13.4)

The left side uses the response scalar as an exponent of the fixed generator. The right side multiplies the commitment by the public key raised to the recomputed challenge. For our signature,

$$2^2 \bmod 23 = 4$$

, while

$$12^8 \bmod 23 = 12$$

and

$$8 \times 12 \bmod 23 \\ = 96 \bmod 23 = 4$$

.

The equal values make the signature pass the arithmetic check.

Why does the check work? Substitute the signing definitions:

(13.5)

The first replacement uses

$$s = k + ez$$

modulo the subgroup order, under which powers of

g

repeat. The next step splits a sum of exponents into a product, and the last substitutes

R

and

y

. This proves the intended algebraic agreement. It does not prove unforgeability. A security argument for a full Schnorr construction also needs the specified group, hash treatment, encoding, attacker model and assumptions. The fact that a discrete logarithm is believed difficult is not, by itself, a proof for any improvised signature recipe.

Optional calculation: signing and verification

The program below uses only public teaching values. It recomputes the challenge from the actual bytes and checks membership in the disclosed subgroup before comparing the equation. The signature on `pay 10` does not pass for `pay 20` in this example. Because the hash range and group are tiny, other changed messages can collide or be forged by search. That limitation is intentional: the program makes the relation visible without pretending to be a secure signing implementation.

Listing 13.1. A tiny Schnorr-style teaching calculation.

```
from hashlib import sha256

p, q, g = 23, 11, 2
x = 6
y = pow(g, x, p)

def challenge(y, R, message):
    if len(message) > 65535:
        raise ValueError("message exceeds teaching length field")
    encoded = (
        b"book-schnorr-v1" + bytes([y, R])
        + len(message).to_bytes(2, "big") + message
    )
    return int.from_bytes(sha256(encoded).digest(), "big") % q

def sign(message, k):
    R = pow(g, k, p)
    e = challenge(y, R, message)
    return R, (k + e*x) % q

def verify(message, signature):
    R, s = signature
    if not (1 < y < p and 1 < R < p and 0 <= s < q):
        return False
    if pow(y, q, p) != 1 or pow(R, q, p) != 1:
        return False
    e = challenge(y, R, message)
    return pow(g, s, p) == R * pow(y, e, p) % p
```

```
message = b"pay 10"
signature = sign(message, 3)
assert (y, challenge(y, 8, message), signature) == (18, 9, (8, 2))
assert verify(message, signature)
assert not verify(b"pay 20", signature)
print("public key 18; challenge 9; signature (8, 2)")
print("original accepted; selected changed message rejected")
```

An elliptic-curve version replaces multiplication of group elements by point addition. With base point

$$G$$

, public point

$$Y = xG$$

and commitment point

$$R = kG$$

, the same positive-key convention gives

$$xG = R + rY$$

. Each scalar acts by repeated point addition, as Chapter 12 explained. This connects the two group notations without making their byte encodings interchangeable. A specified elliptic-curve signature scheme must define its own point representation and validation rules.

13.3 RSA signatures, ECDSA and EdDSA

Table 13.1. Three signature approaches. Each row describes a complete scheme or family, rather than treating a bare group operation as a signature.

Method	Main calculation	Details the interface must specify
RSA-PSS	RSA private/public powers around the PSS message encoding.	Hash, mask function, salt length and signature width.
ECDSA	A curve-point calculation and an inverse of a per-message scalar.	Curve, hash conversion and encoding of the pair
Ed25519	An EdDSA instance with specified curve, hashing and deterministic signing.	Exact variant, point decoding, scalar ranges and message processing.

RSA-PSS: a signature encoding around RSA

RSA-PSS means the RSA Probabilistic Signature Scheme. It hashes the message and combines the digest with a salt in a defined encoding before applying the RSA signing primitive. The salt is additional input used by this encoding, normally generated randomly for a selected nonzero salt length. Its role is different from the secret per-signature exponent in Schnorr or ECDSA: disclosure of the PSS salt does not itself disclose the signing key. The hash, mask generation function and permitted salt length are part of the scheme parameters. PSS is different from OAEP, the RSA encryption encoding studied in Chapter [11](#).^[59]

Let

n

be the RSA modulus,

d

the private exponent and

e

the corresponding public exponent. After PSS has prepared the message, let

v

be the integer represented by its encoded block. The primitive calculations are

(13.6)

The signer calculates

t

and transmits its fixed-width byte encoding as the signature. The verifier obtains

v'

using the public exponent, then checks the prescribed PSS structure against the supplied message and parameters. Recovering a number with the public power is not enough: verification includes the format, range, mask and hash checks. The variable

v

is the encoded block's integer, not an arbitrary message number whose private power would already define a secure signature.

For example, a signature using a 2048-bit RSA modulus has 256 bytes. That length does not impose the 190-byte OAEP message capacity discussed earlier: PSS hashes the message rather than encrypting it inside the RSA block. Its internal encoded block still has strict size conditions, including room for the hash and selected salt. A verifier configured for SHA-256 with a 32-byte salt must not silently reinterpret a packet as a different scheme when the check fails. The RFC's RSA-PSS security discussion is qualified by the RSA assumption, hash modeling and specified construction; it is not a proof for every private-power operation or every application using RSA.

ECDSA: a curve calculation with a secret inverse

ECDSA stands for Elliptic Curve Digital Signature Algorithm. Its public parameters include a curve and a base point

G

of prime order

n

. A private scalar

d

has public point

$$Q = dG$$

. The message's hash is converted to an integer

z

using the specified bit-selection rule; if the hash output is too long, the relevant leftmost bits are used before interpretation. Choosing a per-message secret

k

between 1 and

$$n - 1$$

, the signer calculates

$$(13.7), (13.8)$$

Here

$$x_R$$

is the horizontal coordinate of the calculated point

$$R$$

, interpreted as an integer before reduction modulo

$$n$$

. The inverse

$$k^{-1}$$

is taken modulo the subgroup order, not the coordinate-field modulus. The signature is the pair

$$(r, s)$$

. Both values must be nonzero and within the specified range; the generation procedure defines what happens if either is zero. A deterministic procedure must not be turned into an unchanging retry loop that produces the same failing value forever. [\[73\]](#)

The verifier checks the key, parameters and signature format, then computes

$$(13.9)$$

First invert the response scalar. Then use the two resulting coefficients to multiply the base point and the public point, adding those point multiples to obtain

$$V$$

. Reject the identity point and compare the integer horizontal coordinate of

$$V$$

, reduced modulo

$$n$$

, with

r

. For an honestly generated signature, substituting

$$Q = dG$$

makes

$$r = (u_1 + u_2)G \pmod{O}$$

, which explains the matching coordinate. This relation does not replace the required range, encoding or public-key checks.

We can follow the calculation on Chapter [12](#)'s teaching curve

$$d^2 = d^2 \pmod{O}$$

. Choose

$$G = (5, 1)$$

, whose order is

$$n = 19$$

, and private scalar

$$d = 5$$

, giving

$$Q = (9, 16)$$

. For this separate arithmetic example, take

$$z = 13$$

as the already-converted hash integer; it is public test data, not a claim about a particular full-size hash of a sentence. With

$$k = 3$$

, the point is

$$R = 3G = (18, 6)$$

, so

$$r = 10$$

. Since

$$z^2 = 13^2 \pmod{O}$$

,

(13.10)

For checking,

$$u_1 = 1^2 \pmod{O} = 1$$

. The two coefficients are

and

. Therefore

$$V = 16G + 5Q = 16G + 25G = 41G = 3G = (10, 6)$$

whose horizontal coordinate matches

. The pair

passes. Modulus 17 still governs point coordinates, while order 19 governs the scalar calculations. The example establishes correctness on a searchable toy group, not security.

Encoding remains a separate matter. Some interfaces serialize ECDSA's pair as a structured ASN.1 DER sequence; others specify two fixed-width integers placed consecutively. **DER** is a set of precise encoding rules for structured values. A receiver expecting DER cannot simply split every signature into two equal halves. Nor should it accept out-of-range integers because a later modular calculation happens to produce a convenient remainder. The selected protocol and library determine the required representation and failure behavior.

EdDSA and the specific Ed25519 instance

EdDSA stands for Edwards-Curve Digital Signature Algorithm. An Edwards form is another equation form for elliptic curves, with its own point operations and encodings. **Ed25519** is a particular EdDSA instance, with specified curve parameters and SHA-512 processing. It is different from both ECDSA and X25519. The relationship among some underlying curves does not make their public keys, signatures or private-byte processing interchangeable. Ed25519 has a 32-byte public key and a 64-

byte signature; its public point encoding stores a vertical coordinate together with one bit distinguishing the horizontal coordinate, rather than X25519's encoded Montgomery coordinate. [\[74\]](#)

For plain Ed25519, signing begins with a 32-byte private seed. SHA-512 expands that seed; specified bit processing on the first half creates the secret scalar, while the second half supplies a secret prefix for the per-message calculation. A seed is the private input from which this signing material is derived. It still needs suitable unpredictable generation. The seed, expanded scalar and secret prefix are related but different values; treating an arbitrary public hash as a private seed would not create secrecy.

To explain the main relation, call the private scalar

a

, the base point

B

, its prime order

ℓ

and the public point

$$A = aB$$

. Plain Ed25519 derives the per-message scalar

r

from SHA-512 of the secret prefix followed by the message, with the specified little-endian interpretation and reduction modulo

ℓ

. It calculates

$$R = rB$$

, then derives a challenge

h

from SHA-512 of the encoded

R

, encoded public point

A

and exact message, in that order. The response is

(13.11)

Multiply the public challenge by the private scalar, add the secret per-message scalar and keep the remainder modulo the base point's order. This resembles the earlier Schnorr relation in additive group notation. The transmitted signature contains the 32-byte encoded point

R

followed by the 32-byte little-endian encoding of

S

. The public challenge and per-message scalar are not separately transmitted fields.

For honestly generated values,

$$SB = R + kA$$

explains agreement. Verification must also decode the points and enforce the scalar range

$$0 \leq S < \ell$$

. RFC 8032 specifies a check using the cofactor 8,

$$8SB = 8R + 8kA$$

, and permits the sufficient, stricter unmultiplied equation. These acceptance rules can differ on unusual inputs, so software that must reach the same decision needs a specified validation policy. Reducing an out-of-range received

S

modulo

ℓ

instead of checking its range is not an equivalent parser. An attractive group equality is only one part of the verifier.

Plain Ed25519, Ed25519ctx and Ed25519ph are different variants. The latter variants have specified context and prehash processing. Taking a SHA-512 digest of a file and feeding those digest bytes to a plain Ed25519

interface does not implement Ed25519ph. A **prehash** is a digest calculated before the remainder of the defined signing procedure; the scheme must specify how it is identified and incorporated. A program and its verifier must agree on the exact variant, rather than guessing it from the identical signature length.

Optional program: checking a published Ed25519 signature

The next program reproduces RFC 8032 Section 7.1, Test 2, through a maintained library. All private bytes shown here are published test material and must never protect real data. The message is the single byte with hexadecimal value 72; in ASCII it represents the letter r. The signature is compared with the complete published value, then verified. A changed message, a changed signature byte and a shortened signature are rejected. Repeating signing on the same message produces the same signature for this deterministic instance. [\[74\]](#)[\[77\]](#)

Listing 13.2. A published Ed25519 known-answer example.

```
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.asymmetric.ed25519 import (
    Ed25519PrivateKey,
)

seed = bytes.fromhex(
    "4ccd089b28ff96da9db6c346ec114e0f"
    "5b8a319f35aba624da8cf6ed4fb8a6fb"
)
expected_public = bytes.fromhex(
    "3d4017c3e843895a92b70aa74d1b7ebc"
    "9c982ccf2ec4968cc0cd55f12af4660c"
)
expected_signature = bytes.fromhex(
    "92a009a9f0d4cab8720e820b5f642540"
    "a2b27b5416503f8fb3762223ebdb69da"
    "085ac1e43e15996e458f3613d0f11d8c"
    "387b2eae4302aeeb00d291612bb0c00"
)
message = bytes.fromhex("72")
private = Ed25519PrivateKey.from_private_bytes(seed)
public = private.public_key()
signature = private.sign(message)
assert public.public_bytes_raw() == expected_public
assert signature == expected_signature
assert private.sign(message) == signature
public.verify(signature, message)

changed = bytes([signature[0] ^ 1]) + signature[1:]
for candidate, data in [
    (signature, bytes.fromhex("73")),
```

```

    (changed, message),
    (signature[:-1], message),
]:
    try:
        public.verify(candidate, data)
    except InvalidSignature:
        pass
    else:
        raise AssertionError("invalid case was accepted")

public.verify(signature, message) # replay still verifies
print("published public key and 64-byte signature matched")
print("three invalid cases rejected; exact replay still verifies")

```

The library reports failed verification through an exception. An application must handle failure and stop the operation that required a valid signature; catching the exception and continuing would discard the protection. Successful verification likewise does not identify the public key's owner by itself. In this example, we know the expected public key because the test specification supplies it. A real application needs its own trustworthy way to select the appropriate verification key.

13.4 Nonces, deterministic signing and key reuse

The word **nonce** occurs in several cryptographic settings, but its requirements depend on the construction. The AEAD nonce in Chapter [7](#) could normally be public and had to obey the selected uniqueness rule. Schnorr and ECDSA use a per-signature scalar that must remain secret and meet the generation procedure's requirements. A predictable or reused value can expose the long-term private signing scalar. The PSS salt serves another role again: treating every value called a nonce or salt as if it had the same confidentiality and reuse conditions would be a serious mistake.

We can calculate the Schnorr reuse failure directly. Keep the same private scalar

$$x = 6$$

and deliberately reuse

$$k = 3$$

, so the commitment remains

$$R = 8$$

. Our defined hash gives

$$e_1 = 9$$

for pay 10, with response

$$s_1 = 2$$

. For $\text{pay} = 20$, it gives

$$e_2 = 4$$

, with response

$$s_2 = (3 + 4 \times 6) \bmod 11 \\ = 5$$

.

The two signing equations contain the same secret

. Subtracting them cancels that term:

$$(13.12)$$

Every quantity except

x

is now known from the messages, public key and signatures. When the challenges differ modulo prime

q

, their difference is nonzero and has an inverse. Multiply both sides by that inverse to obtain

$$(13.13)$$

For our values, the response difference is

$$2 - 5 = -3$$

and the challenge difference is

$$9 - 4 = 5$$

. Since

$9^{-1} \bmod 11 = 9$

, the recovered scalar is

$$\begin{aligned} & -3 \times 9 \bmod 11 \\ & = -27 \bmod 11 = 6 \end{aligned}$$

The attacker has recovered the signing key through arithmetic on public data. They can also recover the reused

as

$$\begin{aligned} s_1 - e_1 x \bmod 11 &= 2 \\ -54 \bmod 11 &= 3 \end{aligned}$$

This is not a discrete-logarithm attack and does not become safe merely by enlarging the group.

Disclosure of one signing secret can also be enough. If

k

is known and the challenge

e

is nonzero, rearranging

$s = k + ex$

gives

$x = (s - k)e^{-1} \bmod q$

. Biased or partially leaked scalar values can create other attack opportunities whose analysis depends on the scheme and leak. We do not need to claim that every small bias immediately reveals a key to understand the requirement: the generation and protection of per-signature secrets are part of the signature system's security.

ECDSA has a related reuse problem because its response contains

$$k^{-1}(z + rd)$$

. Reusing the same

k

gives the same point and normally the same

r

, allowing equations from different message hashes to reveal

k

and then the private scalar under the relevant nonzero conditions. The exact algebra differs from Schnorr, so its recovery formula must not be copied unchanged. Both examples show why choosing a sound curve is insufficient when the per-message secret is mishandled.

Deterministic signing calculates the required per-message value from secret key material and the message under a specified construction. RFC 6979 defines such a procedure for ECDSA using a keyed internal process, with the usual verification equation unchanged. Plain Ed25519 also derives its per-message scalar deterministically, using its secret prefix and message. Determinism is not the dangerous reuse just demonstrated: with the same key and message it repeats the same signing calculation, while different messages are processed through the prescribed secret-dependent derivation. Manually fixing

$$k = 3$$

, or using a public message hash alone as

k

, is not a substitute. [\[75\]](#)

Deterministic signing still needs an unpredictably generated private key. It also needs protection against secret-dependent timing, power leakage and faults that can disturb the calculation. It removes a particular dependence on

fresh signing randomness; it does not solve every implementation problem. Nor does the concept transfer unchanged to protocols with several cooperating signers: those protocols have their own nonce coordination, commitment and restart rules. We will keep such constructions with the chapters that introduce them.

Keys should also have clear purposes. Reusing the same underlying private value for signing and key agreement, or across unrelated algorithms, creates relationships and interfaces that the individual schemes' arguments may not cover. Use separately generated key material and the protocol's specified derivation and storage arrangements. Merely placing different names on the same private bytes does not create that separation. Key lifecycle, compromise response and recovery will be developed in Chapter 14; here the essential point is that sound arithmetic must be used through the right interface with the right secret inputs.

13.5 Certificates and trust anchors

Sam can verify a signature using a public key supplied next to a downloaded file. But an attacker can supply their own file, signature and key together. All three may be mathematically consistent. Sam therefore needs a trustworthy basis for deciding which public key represents the publisher or service he intended. A **certificate** is a signed statement that connects a subject and a public key under the issuer's specified rules. A **certification authority**, abbreviated CA, is an issuer authorized within a particular trust arrangement to make such statements. The certificate adds a signed relationship; it does not create trust without a starting point.

In the Internet X.509 system, a certificate contains the subject's public-key information, issuer and subject information, a serial number, a validity period and extensions describing purposes or constraints. The issuer signs an exact encoded certificate body. That body includes the relevant public key and metadata; the issuer is not signing only an informal name printed on the certificate. **X.509** names the certificate framework, while the Internet profile in RFC 5280 specifies formats and rules used in this setting. A **public-key infrastructure**, or PKI, includes the authorities, policies, certificates and validation processes that maintain these relationships. [\[76\]](#)

The issuer's key can itself appear in a certificate signed by another authority. Figure [13.2](#) shows a simple arrangement with a trust anchor, an intermediate CA and a service certificate. A **trust anchor** is trusted information supplied to the validation process, commonly a configured authority name and public key with applicable constraints. The device or application obtains that trust through an established installation or administration process. A server sending a root certificate does not, by that act, make its root trusted. Likewise, a self-signature shows a relationship within the certificate; it cannot make an unknown key an acceptable trust anchor.

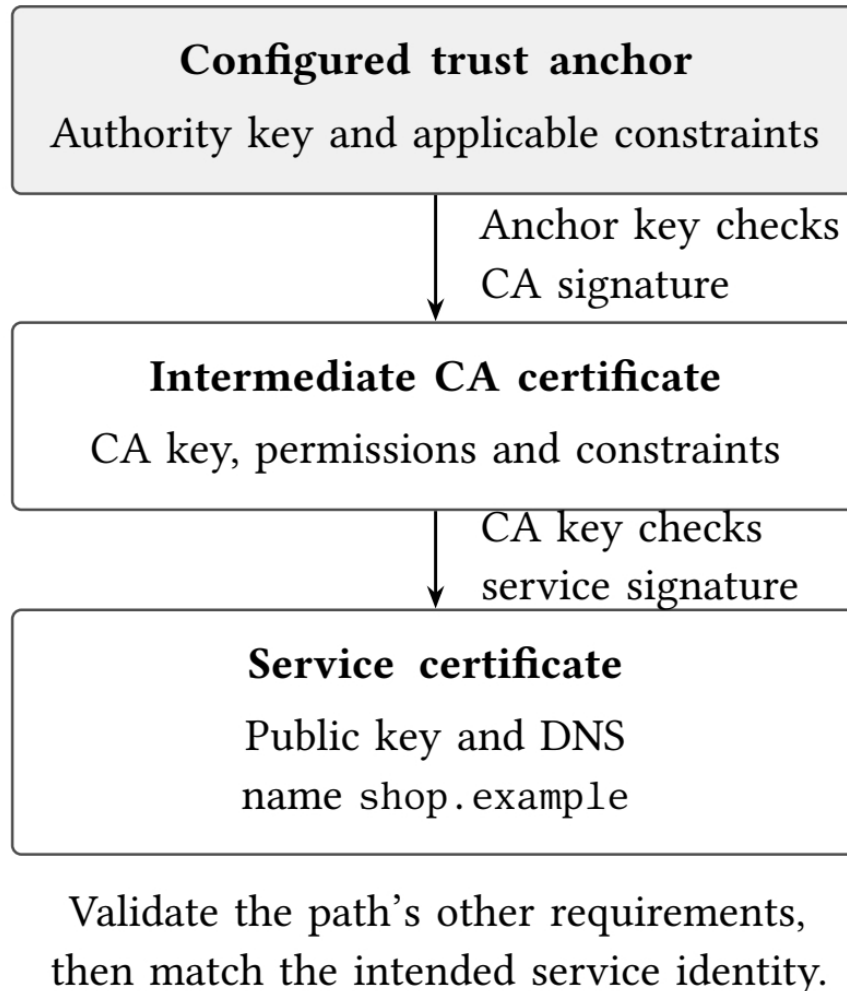


Figure 13.2. A simplified certificate relationship. A configured trust anchor provides the starting authority; its key checks the intermediate CA certificate, and the intermediate's key checks the service certificate. The self-signed certificate often used to represent an anchor is not an extra certificate in RFC 5280's prospective validation path. Signature checks alone do not complete validation.

Path building finds candidate certificates connecting the target to an acceptable anchor. **Path validation** checks a candidate against the required rules and inputs. These tasks are related but distinct. A certificate's issuer name alone does not prove that it was signed by the expected issuer, and successful signature checks alone do not establish a valid path. Validation also considers the relevant time, CA status and signing permissions, path-length and name constraints, permitted policies and other applicable requirements. A critical extension tells the verifier that its rule matters: an unrecognized or unprocessable critical extension cannot simply be ignored.

The service certificate and the certificates above it need not use the same public-key algorithm. A CA might use one permitted signing method to certify a subject key for another permitted method. The subject's public-key algorithm describes the certified key; the certificate's signature algorithm describes the issuer's signing operation. These are different fields with different roles. An algorithm name appearing somewhere in a certificate is therefore not enough to decide which operation or parameters a verifier should apply.

A certificate's validity interval limits when it is intended to be accepted, but it does not directly answer whether the private key has since been compromised or the certificate revoked. A **certificate revocation list**, or CRL, is a signed list used to publish revoked certificates under the relevant issuer and scope. Its date, issuer, coverage and freshness matter. Applications have defined status-checking policies; they do not all obtain the same online information automatically. Expiration and revocation are different reasons why a certificate may no longer be acceptable.

Consider a separate TLS example: a client wants the service `shop.example`, while the server presents a certificate for `files.example`. The certificate chain could be valid under a configured trust anchor, yet the service identity still fails to match the requested name. RFC 9525 separates this matching task from certification-path validation. For a DNS service name, its rules use the appropriate subject alternative name entry, rather than treating the subject's Common Name as a fallback DNS identity. Checking the chain and checking the intended service are both necessary parts of the relevant authentication process; passing one does not make the other unnecessary. [\[10\]](#)

The level of identity established also depends on the certificate's purpose and issuance policy. A certificate binding a DNS name to a public key does not automatically establish who is legally responsible for every claim on a website, and it does not certify the safety of a downloaded program. Even a correctly identified service can deliver harmful content. Returning to Nora's update, Sam needs the appropriate publisher-key trust arrangement and release policy for that application. A service certificate is not automatically interchangeable with a code-signing credential or a previously trusted publisher key.

13.6 What a valid signature does not prove

A verification result should be interpreted as evidence about the specified message, key and scheme. Under the relevant assumptions, a valid signature supports the conclusion that the signing capability associated with that key was used, rather than an outsider successfully forging it. It does not reveal whether Nora personally pressed an approval button. Malware, a stolen private key or an automated signing service can use that capability too. A strong signature algorithm preserves its mathematical behavior when the wrong person controls the key; the operational failure is that the wrong person now has the signing power.

Nor does a valid signature establish that the message is true. A signature on a measurement identifies a signed representation of that measurement, not whether the sensor was accurate. A signature on a payment instruction does not prove that the account has sufficient funds or that the signer was permitted to approve that amount. **Authorization** is the application's decision about allowed actions. An authenticated key relationship can inform that decision, but it cannot replace the account rules, role checks or other evidence the application requires.

Freshness also requires its own evidence. If Sam downloads a correctly signed but outdated release, an attacker may have succeeded in a rollback without altering any signed byte. A signed version number helps only if the updater checks it against a trusted policy or retained state. A date written into a signed message is a statement made under that key, not independent proof of when the signature was created. Establishing a reliable signing time needs an appropriate additional mechanism. The mathematical verifier alone does not supply a trusted clock or remember the most recent approved version.

The same separation helps explain signature changes. In ordinary ECDSA verification,

$$(r, s)$$

and

$$(r, n - s)$$

can both verify the same message under the same key. On our order-19 teaching curve,

$$(10, 2)$$

and

(10, 17)

both pass. This is a form of **signature malleability**: an accepted signature can be transformed while retaining an accepted relationship to the same message. It is different from forging a signature on a new message. Some protocols require a particular representative, such as a low-

S

rule, to remove this ambiguity; such a rule belongs to that protocol's specified acceptance policy. Applications should not assume that signature bytes alone provide a unique identifier for a signed action.

For Ed25519, the specified scalar-range check prevents a simple change that adds the group order to the encoded response. That requirement is one reason to test invalid inputs as well as valid examples. Different libraries or protocols can also disagree on unusual point encodings or cofactor-related cases if their acceptance policies differ. A system whose participants must agree on validation needs those rules fixed explicitly. Matching a few ordinary signatures does not establish identical behavior for every malformed or exceptional input. [\[74\]](#)[\[78\]](#)

Non-repudiation describes using signature evidence in a dispute about an action or statement. Digital signatures can support that purpose, especially when identity binding, key custody, records and timing are well managed. A verification equation does not by itself prove human intent, exclude a prior key compromise or settle a legal question. Those conclusions depend on the surrounding evidence and applicable framework. This is why the capability to show signature evidence to a third party is valuable, while a blanket claim that a signer can never deny any signed data is too strong.

Finally, conventional RSA, ECDSA and Ed25519 do not provide post-quantum security against the sufficiently capable quantum attacker considered in Shor's algorithm. Long-lived verification needs an algorithm and migration policy appropriate to its time horizon. We will examine post-quantum constructions in their own chapters rather than treating a classical signature's present use as a permanent guarantee. The lesson throughout this chapter is to state exactly which claim is being checked: signature

correctness, resistance to forgery, trust in a key, authority to act, freshness and the meaning of the signed data each require their own reasoning.

Questions to check your understanding

1. Identify the inputs and output of signing and verification. Why is a signature different from both encryption and a shared-key MAC?
2. What counts as a forgery in the ordinary chosen-message setting? Distinguish a new-message forgery, a changed signature for the same message and replay of the exact signed record.

3. In the Schnorr example, calculate

y

,

R

and

S

for

$$x = 6$$

,

$$k = 3$$

and challenge

$$e = 9$$

. Check both sides of the verification equation. Which calculations use modulus 23, and which use modulus 11?

4. Why must the verifier recompute the challenge from the exact message and specified public inputs? Does the correctness equality prove that the teaching construction is unforgeable?
5. Explain the role of PSS before and after the RSA primitive. Does the 190-byte OAEP capacity limit the messages that an RSA-PSS scheme can sign?
6. On the order-19 teaching curve, verify the ECDSA pair

$$(10, 2)$$

for

$$d = 5$$

,

$$z = 13$$

and

$$k = 3$$

. Why is the inverse of

S

taken modulo 19 rather than 17?

7. Distinguish Ed25519's private seed, private scalar, public point and signature. Is signing a file's SHA-512 digest with plain Ed25519 equivalent to Ed25519ph?
8. Recover the Schnorr private scalar from responses 2 and 5 with challenges 9 and 4 under order 11. Why is specified deterministic signing different from fixing a constant secret
k
?
9. What is a trust anchor, and why does a server-supplied self-signed certificate not automatically establish one? Distinguish certificate-path validation from matching an intended DNS service name.
10. Give three claims a valid signature does not establish by itself. How can an outdated but correctly signed software release still be dangerous?

V. Building Secure Systems

14. Managing Keys

Part V: Building Secure Systems

Nora and Sam have chosen sound algorithms for their document service. Files are encrypted, release signatures verify correctly, and connections use authenticated key agreement. Yet a practical question remains: who controls the keys, and what happens when a device fails or a key is copied? A correct cipher cannot recover a lost decryption key on demand. A correct signature algorithm cannot distinguish Nora from an attacker who has obtained her signing capability. The protection depends on the life of the key around the mathematical operation.

Key management is the set of arrangements for creating, establishing, storing, using, replacing, recovering and eventually destroying cryptographic key material. It also protects the information that explains each key's purpose, owner and allowed use. Some keys must remain secret; public keys normally require a trustworthy association with the intended party rather than confidentiality. Availability matters too: if nobody can obtain an authorized decryption key when needed, protected data can become permanently unreadable. Keeping keys hidden is therefore necessary for many tasks, but it is not the whole subject.

This chapter follows one service from key generation to incident recovery. We will connect the randomness, key-agreement and signature ideas already developed to practical decisions about access, storage and replacement. A small encrypted-file example will distinguish changing a wrapping key from changing the key that protects the file itself. The aim is to understand the reasoning behind a key policy, rather than adopt an unexplained rule such as replace every key after a fixed number of days. Different key roles have different dependencies, and a change that is safe for one role can destroy information needed by another.^[8]

14.1 Generating keys with the right source

Key generation begins with the selected algorithm and its required inputs. A secret AES key is a fixed-length byte string drawn through a suitable generation process. An elliptic-curve private key also needs the algorithm's prescribed scalar generation or seed processing. RSA key generation creates suitable primes and related parameters, rather than choosing an arbitrary string and calling it a key. A maintained cryptographic library's key-generation interface performs the algorithm-specific work. Using the same number of bytes does not make keys for different algorithms interchangeable.

For a key that must be unpredictable, the generation process needs an appropriate source of uncertainty. Chapter [5](#) distinguished an entropy source from a deterministic random bit generator. The entropy source supplies unpredictability under a stated model; the generator maintains secret state and expands suitable input into output for applications. Its state must be protected and initialized correctly, with reseeding as required by its design. Producing a long output does not create an equally long amount of independent uncertainty. The SP 800-90 documents treat these roles separately, so a program that merely calls a random-looking function has not demonstrated every condition of a sound generation system. [\[23\]\[24\]\[25\]](#)

For ordinary application development, this means using the documented cryptographic generation interface backed by the operating system or selected cryptographic module. It does not mean constructing a generator from the current time, a username, a process identifier or a general-purpose simulation generator. Such inputs can be guessed or repeated. A timestamp that changes quickly is still drawn from a narrow, predictable range when an attacker knows approximately when a key was created. Likewise, passing a predictable value through SHA-256 does not make the value secret.

An application must also respond correctly when generation fails. If the required randomness or module service is unavailable, substituting zero bytes, a fixed demonstration key or yesterday's saved output defeats the intended design. The appropriate failure behavior is part of the application: it may refuse the operation, wait for a documented initialization condition or report that the service is unavailable. It should not continue under an undisclosed weaker rule. A successful call only establishes that the selected

interface returned; confidence in the generation system also depends on its implementation and environment.

Copying machine state creates another concern. A virtual-machine image or process snapshot can copy a generator's secret state as well as application secrets. Two restored instances could then follow related deterministic histories if the underlying system does not handle the situation appropriately. The response depends on the operating system, module and deployment design; adding the current time is not a general repair. The key-management lesson is to treat cloning, rollback and restoration as events that can affect secret state, counters and key identity, rather than assuming that a restored application is automatically starting with fresh cryptographic material.

A **password-derived key** has a different starting point. It is derived from a human-chosen secret using the specified password-based procedure, with its salt and resource parameters, as Chapter 9 explained. Its resistance to guessing depends on that password distribution and derivation cost. It is not automatically equivalent to an independently generated random key of the same length. In an encrypted-file system, a password may protect access to a separately generated file key. Changing the password can then change that access protection without changing the file key, a distinction we will examine shortly.

Before a key becomes active, give it a defined purpose and record its identity. A **key identifier** is a name or reference used to select a key; it is not necessarily secret and is not proof of ownership. A **key inventory** records which keys exist, which algorithms and purposes they belong to, who or what may use them, their states and the information they protect. The inventory normally points to protected key storage instead of collecting all private values in an ordinary spreadsheet. An accurate inventory makes both planned replacement and emergency response possible.^[8]

14.2 Distribution and trust establishment

Key establishment is the process by which the required parties obtain the key material and associations needed for an operation. In key agreement, parties contribute inputs to calculate matching secret material. In key transport, one party provides key material to another through an

appropriate protected construction. In public-key distribution, the public value need not be hidden, but its association with the intended party and purpose must be protected. These tasks have different security requirements. Sending a public verification key and sending a secret data-encryption key are not the same distribution problem.

Chapter [10](#) showed why agreement needs authentication when an active attacker can substitute public values. Chapter [13](#) showed how a trusted certificate path or another trusted key arrangement can support the association between a public key and a subject. A new key must enter through that established trust process. If an untrusted message says that Nora's verification key has changed, accepting its replacement key solely because the message names Nora would allow an attacker to choose her apparent identity.

A **fingerprint** is a digest calculated from a specified public-key representation or related object. It can make comparison easier when a trusted independent channel is available. But displaying a fingerprint next to an untrusted download of the same key does not create an independent check: an attacker can replace both. The parties must agree on what is hashed and how the result is represented. Comparing only a short fragment also gives less evidence than comparing the intended complete fingerprint under the relevant procedure.

An authenticated confidential channel can protect a transferred secret key, but endpoint authorization still matters. The channel must reach the intended recipient, the receiving process must be permitted to obtain the key, and the stored result must retain its purpose and algorithm association. Protecting a network packet does not stop a debug log at either endpoint from recording the plaintext key. A key copied through a deployment script, support message or temporary file creates additional locations that the management policy must account for.

Metadata can be security-critical even when it is public. Suppose a record names a key as belonging to customer A, while an altered record assigns the same key to customer B. The bytes of the key may be unchanged, yet the application could now apply it under the wrong authority. Algorithm identifiers, permitted operations, customer or document associations and version information require appropriate

protection. An untrusted key identifier should guide a controlled lookup, not give the caller unrestricted authority to select any secret in the system.

This also explains why **key confirmation** and identity checking are separate. Confirmation supplies evidence, under the selected protocol, that another participant has obtained the expected keying material. It does not automatically prove that this participant is the person or service the application intended. Likewise, a proof of possession can demonstrate control of a private key without establishing the real-world identity or authorization of the controller. The trust arrangement must connect these pieces without treating one as a replacement for all the others.

For a document shared by several people, distribution needs an explicit membership policy. New members may require access to current files; removed members may need to lose access to future versions. Once a member has copied a plaintext file or an authorized decryption key, later deletion of their account cannot erase that copy. A new group key can protect later material from a removed member who does not obtain it. It cannot reverse past disclosure. This limitation is a consequence of giving someone the information, not a weakness that a stronger cipher can repair.

14.3 Storage, access and hardware protection

Stored key protection has two related questions: can an attacker extract the key, and can an attacker cause it to be used? Keeping a signing key inside a restricted module can reduce the chance of copying its private bytes. But if a compromised application can ask that module to sign any message, it may still abuse the signing power. A useful policy therefore restricts operations and callers as well as exports. The protected object is both the key material and the authority associated with its use.

Software storage can include an operating-system key store or a service that protects secrets and controls access to them. A **key-management service**, often called a KMS, manages keys and cryptographic operations through defined interfaces. The phrase describes a service role, not a guarantee that every deployment has the same protections. Its administrators, authentication credentials, access rules, availability and recovery mechanisms are part of the design. Placing a raw key in an environment variable may avoid hard-coding it in source, but does not

establish that it cannot appear in diagnostics, process inspection or another unintended copy.

Least privilege means granting only the operations and scope needed for a task. A file-reading worker might be permitted to decrypt selected documents without being allowed to export the master key or approve software releases. An administrator who changes access policy need not automatically be able to read every protected file. Separating duties, requiring more than one authorized approval for particularly sensitive actions and recording the resulting decisions can reduce the power of a single mistake or compromised account. These controls need to be enforced by the system, rather than written only in a policy document.

A **hardware security module**, or HSM, is a device designed to perform cryptographic operations within a protected boundary. A key can be generated and used there without ordinary applications receiving its raw private value. A device may also enforce operation permissions, authentication and controls against specified physical attacks. These capabilities depend on the particular module and configuration. Hardware-backed storage does not make the host's instructions trustworthy or make data returned to an authorized caller inaccessible to that caller.

FIPS 140-3 defines security requirements for cryptographic modules across several assurance levels. Validation concerns the identified module, version, boundary and required operating conditions. It does not automatically validate an entire application that calls the module. Nor does the word HSM alone identify which requirements or evaluation results a device meets. When a system depends on a validated configuration, that configuration and its applicable security policy are part of the evidence, not an interchangeable marketing label. [\[81\]](#)

Separating file keys from wrapping keys

An **envelope-encryption** design separates a key that protects data from a key that protects that data key. Call the first a data-encryption key,

K_D

, and the second a key-encryption key,

K_W

, also called a wrapping key. A file's AEAD operation produces protected data

C

, while a specified key-wrapping operation produces wrapped key material

W

:

(14.1)

Here

M

is the file's plaintext,

N

its nonce and

A

the associated data. The output

C

includes the ciphertext and authentication tag under the selected interface.

The output

W

protects the data key, not the file directly. The application stores the nonce, protected file, wrapped key and required identifiers in their specified formats. The wrapping key is maintained under separate protection.

To read the file, an authorized operation unwraps

W

using

K_W

to recover

K_D

, then supplies that key to the matching AEAD decryption operation. Failure at either stage is a failure to obtain accepted plaintext. This hierarchy can limit where a higher-level key is exposed and allow different access controls at different layers. It also creates dependencies: losing the only usable wrapping key can make many wrapped data keys unavailable,

while exposing that wrapping key can endanger all data keys an attacker can unwrap with it.

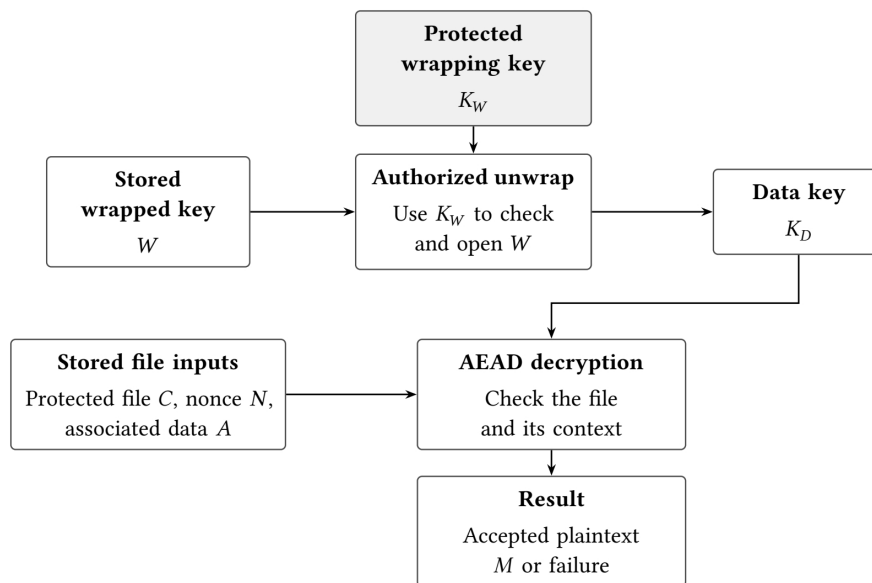


Figure 14.1. Two layers in a simplified envelope design. The protected key operation recovers the data key from its wrapped form; the data key then opens the protected file. Metadata and permissions govern both stages. A wrapping operation does not by itself authenticate every surrounding identifier or authorize the caller.

AES Key Wrap, abbreviated KW, is one specified method for protecting key material. Its padded variant, KWP, supports different input-length requirements. These are defined constructions, not a recommendation to invent a way of encrypting a key with a block cipher. The ordinary KW interface used later is deterministic for a fixed wrapping key and key input, has an eight-byte expansion, and accepts the specified input lengths. Wrapping a 32-byte data key therefore produces

32 + 8 = 40

bytes; this is the wrapper size, not the file's ciphertext length. KW does not have the same nonce-and-associated-data interface as AES-GCM. Its integrity check also has its own bounds; it must not be described as a general GCM replacement with an identical authentication tag. [\[79\]](#)[\[80\]](#)

Bindings outside the wrapped key need their own specified protection. For example, a wrapping-key identifier used to locate a service key must be checked against the caller's permitted scope. A document identifier and stable data-key version can be bound through the file's AEAD associated

data. If changing a wrapping-key version is meant to leave the file ciphertext untouched, that mutable wrapper version cannot simply be changed inside the file's authenticated inputs without updating its protection. A complete format must define which metadata belongs to each layer, how it is authenticated and how updates are recorded.

14.4 Usage periods, rotation and revocation

A **cryptoperiod** is the period during which a key is authorized for its specified use. Its length depends on the key's role, the amount and sensitivity of protected information, the algorithm and usage limits, the operating environment and the consequences of compromise. There is no single replacement interval that is correct for every key. A short-lived session key, an archival decryption key and a software-release signing key serve different purposes. Calendar dates are only one possible limit; a construction can also impose limits on messages, invocations or protected data volume.

It helps to distinguish **applying protection** from **processing information already protected**. An encryption key may stop being used for new files while remaining available to decrypt older files. A private signing key may reach the end of its signing period and be destroyed, while the public verification key and supporting evidence remain available for checking earlier signatures. NIST calls these the originator-usage and recipient-usage periods. The distinction concerns the operation being authorized, rather than a physical sender or receiver who must always hold the same role. [\[8\]](#)

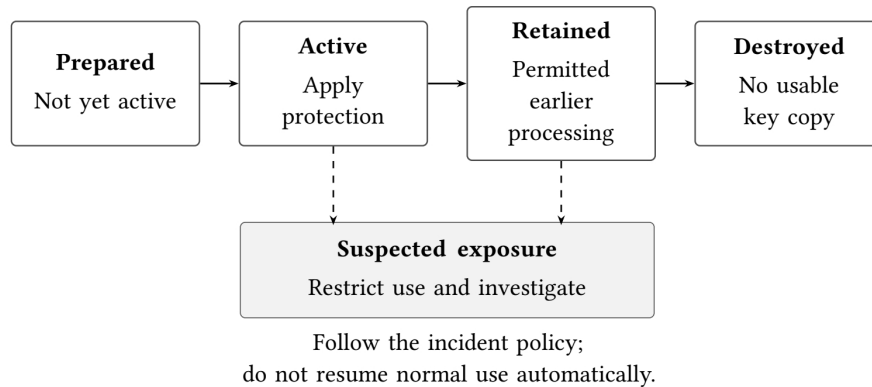


Figure 14.2. A simplified management route, not a complete normative state machine. Retention for earlier processing is permitted only for the relevant key role and policy; an expired private signing key does not need to be kept for public verification. Suspected exposure requires a separate response rather than automatic return to normal use.

Rotation replaces a key or active key version under a defined transition plan. The plan must explain how new protection selects the new key, how recipients recognize its identifier, what happens to work already in progress and how earlier data remains readable where required. Publishing a new identifier does not mean every running worker has stopped using the old key. Cached values, offline devices, queued tasks and restored backups can all preserve earlier state. A transition is complete only when its required operational conditions are met.

Table 14.1. Retirement depends on the key’s purpose. These are management distinctions, not universal retention periods.

Key role	When new use ends	What may still be needed
Data-encryption key	Stop encrypting new data with that version.	Controlled decryption of retained older data.
Private signing key	Stop generating signatures; destroy it when required by its policy.	The public key and evidence for earlier verification.
Wrapping key	Stop wrapping new keys with that version.	Controlled unwrapping, or verified migration of retained wrappers.
Ephemeral agreement secret	End the specific protocol use.	Usually destruction, rather than archival of a secret meant to expire.

In an envelope design, **rewrapping** changes the protection around a data key while leaving that data key unchanged. If

W_1

wraps

K_D

under the earlier wrapping key

$K_{W,1}$

, a transition can calculate

(14.2)

First recover the existing data key through the authorized unwrapping operation. Then protect that same data key with the new wrapping key. The protected file

C

can remain unchanged if its authenticated inputs remain unchanged. A system can perform this operation inside an appropriate protected service, but the management policy must still authorize it and record the updated association.

Rewrapping can be useful for a planned transition when the earlier key remains trusted. It does not heal a data key exposed through a compromised earlier wrapper and wrapping key. An attacker holding those earlier values can still recover the same

K_D

. If protection of future data requires a new data key, generate and establish that new key, use fresh inputs as required by the selected encryption construction, and migrate data according to the plan. Re-encrypting does not remove plaintext or old ciphertext-and-key copies already obtained by an attacker.

Revocation withdraws a key's authorization or the trust placed in its associated credential. It can be needed before an intended expiration date, including after suspected compromise or an administrative change. A certificate revocation mechanism informs relying parties under its specified rules; a service's access policy can also stop particular callers from using a key. These mechanisms affect compliant participants. They do not change

the mathematical result of decrypting a copied ciphertext with a copied secret key, or force an offline verifier to receive a notice it never checks.

Replacing a key also differs from changing an algorithm. A new key for the same construction does not repair an attack that breaks that construction itself. Moving to a different algorithm can change key formats, signature or ciphertext sizes, interfaces and trust dependencies. A useful inventory records these associations so that an algorithm transition can identify affected records and users. The policy should permit planned migration without allowing an attacker to force an unapproved weaker alternative when a preferred operation fails.

Optional program: what rewrapping changes

The program below generates transient demonstration keys, encrypts one message with AES-GCM and wraps its data key using AES Key Wrap. It then rewraps that data key under a new wrapping key. The associated data binds a stable document and data-key identity; the mutable wrapping-key version is deliberately outside it. A real format must also protect and authorize the wrapper's surrounding metadata. The program has no persistent key store, distributed transition or access-control system, and it deliberately keeps the old key in memory to show the limit of rewrapping.

[\[83\]](#)

Listing 14.1. Two layers of protection and a wrapping-key transition.

```
import os
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.keywrap import (
    aes_key_wrap, aes_key_unwrap, InvalidUnwrap,
)

data_key = AESGCM.generate_key(bit_length=256)
old_wrapper_key = AESGCM.generate_key(bit_length=256)
new_wrapper_key = AESGCM.generate_key(bit_length=256)
nonce = os.urandom(12)
message = b"Project file for Nora and Sam"
aad = b"book-file:v1|document:17|data-key:42"
sealed = AESGCM(data_key).encrypt(nonce, message, aad)
old_wrap = aes_key_wrap(old_wrapper_key, data_key)

recovered = aes_key_unwrap(old_wrapper_key, old_wrap)
new_wrap = aes_key_wrap(new_wrapper_key, recovered)
assert len(old_wrap) == len(new_wrap) == 40
opened_key = aes_key_unwrap(new_wrapper_key, new_wrap)
```

```

assert AESGCM(opened_key).decrypt(nonce, sealed, aad) == message
print("rewrapped 32-byte data key; file ciphertext unchanged")

changed = bytes([new_wrap[0] ^ 1]) + new_wrap[1:]
try:
    aes_key_unwrap(new_wrapper_key, changed)
except InvalidUnwrap:
    pass
else:
    raise AssertionError("changed wrapper was accepted")

try:
    AESGCM(opened_key).decrypt(nonce, sealed, aad + b"changed")
except InvalidTag:
    pass
else:
    raise AssertionError("changed file context was accepted")
print("selected changed wrapper and file context rejected")

# Deliberately retained old material still opens the old file.
copied_key = aes_key_unwrap(old_wrapper_key, old_wrap)
assert AESGCM(copied_key).decrypt(nonce, sealed, aad) == message
print("copied old wrapping material still recovers the file")

```

The two failure types belong to different layers: a key-wrapping check rejects the selected modified wrapper, while the AEAD check rejects the selected changed file context. These checks do not prove that every corruption can be detected with certainty or that the whole application is secure. The final lines show a deterministic consequence of retained old material. A new active key version is a management decision, not a mathematical instruction that destroys an attacker's copies.

14.5 Backup, recovery and destruction

A **backup** preserves information needed to restore service after a failure. A **key archive** retains key information for permitted later processing beyond its normal operational period. A **recovery process** defines how an authorized party retrieves or reconstructs what is needed. These arrangements can overlap, but they are not interchangeable. Copying an encrypted database without its recoverable key dependencies may preserve unreadable bytes. Copying every key without controlling the recovery path may create a second, weaker route to all the protected data.

The decision to preserve a key depends on its role. An archival data key may be needed as long as retained documents must remain readable. A public verification key and evidence about its earlier association can be needed for old signatures. A private signing key is not needed merely to

verify those signatures, and making extra private signing copies increases the number of possible signers. SP 800-57 treats backup of private signing keys cautiously, while allowing justified cases such as some CA continuity arrangements under the required control. It does not recommend backing up every private key indiscriminately.^[8]

Ephemeral secrets introduce another trade-off. Their intended deletion can limit what an attacker learns from a later compromise. Archiving those secrets for convenient recovery may remove that benefit. If a service promises that old session material is no longer available, its logs, diagnostic dumps and device backups must not silently preserve a route to the same secrets. The recovery requirement and the desired exposure limit must be decided together. Neither unlimited recoverability nor deletion of everything is the correct policy for all applications.

Recovery requires the dependencies, not just one labeled key file. In the envelope example, restoring the file can require the ciphertext and tag, nonce, exact associated data, wrapped data key, permitted algorithm parameters and access to the appropriate wrapping-key version. It may also require service credentials and enough inventory information to select the right objects. Protecting the wrapper while losing the mapping from documents to wrapper versions can create an availability failure even if every cryptographic primitive remains correct.

A recovery test should therefore demonstrate the intended operation through the full authorized route. Restoring a readable sample document is stronger evidence than checking that a backup file exists. The test should also confirm that an unauthorized caller cannot use the same route, that required approvals are enforced and that a restored service does not reactivate keys marked compromised. A restore can bring back old policy records or counters as well as old keys. Those values must be reconciled with the system's trusted current state before normal processing resumes.

Some arrangements use **split knowledge**, where no one participant holds the complete secret in an independently usable form, or **dual control**, where a sensitive operation requires more than one authorized participant. These are different properties. Dividing a key into two arbitrary files does not automatically provide a sound threshold-recovery construction, and requiring two people to click a button does not imply that neither can access

the complete key. The selected mechanism must define what a participant can learn or do, how recovery is authorized and what happens when a participant becomes unavailable.

Recovery authority is itself sensitive. A helpdesk path that can bypass normal encryption access becomes part of the effective security boundary. Resetting an account password does not automatically recover a separately encrypted file, as Chapter 2 explained. If an application offers recovery, it must identify which additional secret, service or authorized party makes that recovery possible. If no such route exists and the only decryption material is lost, an honestly implemented recovery screen cannot reconstruct it from the public ciphertext.

Destruction means removing every usable route

When secret material is no longer required, **key destruction** aims to make it unavailable for further use or recovery under the intended assurance level. Deleting a filename or assigning a new value to one program variable does not establish this result. Copies can exist in backups, caches, logs, crash dumps or replicated storage. A parent secret may also be able to derive the key again. The inventory and dependency record help identify which copies and reconstruction paths must be addressed.

Cryptographic erase uses the removal of required key material to make encrypted target data infeasible to recover. It depends on sound encryption, suitable key generation and control of the relevant key copies and derivation paths. It does not erase data that escaped the encrypted boundary in plaintext. A retained copy of a usable data key, or of an earlier wrapper and the key that opens it, can defeat the intended result. The sanitization method must fit the storage system and sensitivity of the information. ^[82]

NIST's current media-sanitization guidance separates checking that a sanitization operation completed from assessing whether its outcome is acceptable for the data and risks involved. An application should retain suitable evidence of the process without preserving the secrets it intended to remove. Repeatedly overwriting an ordinary file is not a universal answer for flash storage, replicated services or storage that remaps writes internally.

Our Python example makes no claim to demonstrate zeroization: its high-level runtime can retain copies beyond the lifetime of a particular variable.

Finally, destruction of a managed copy cannot revoke knowledge already obtained elsewhere. If an attacker has copied the old wrapping key and an old wrapped data key, destroying the service's current copy does not make those attacker-held bytes disappear. The distinction is essential when describing deletion guarantees. An organization can enforce and assess its own defined storage boundary; a claim about all copies everywhere requires evidence that the system may not possess.

14.6 Responding to a key compromise

A **key compromise** occurs when the key or its trusted use can no longer be relied on for the required protection. Disclosure is one possibility. Unauthorized substitution, a broken association with an owner, or access that permits abusive signing or decryption can also undermine the intended security. Suspicion may arise from an exposed backup, a device theft, an unexpected key-management operation or an access-control failure. Waiting for a visible forged signature before acting would ignore other evidence of lost control. [\[8\]](#)

Begin by establishing what is known and which capabilities may have been exposed. A data key affects its protected data. A wrapping key can affect every accessible wrapper under it. A private signing key can enable new accepted signatures wherever its public key is still trusted. A copied service credential may permit key operations even when no raw key has been exported. The response must follow these dependencies rather than replacing only the object mentioned in the first alert.

A lost device and an exposed recovery package

Consider Nora's document service. An employee loses a laptop containing a local copy of the document database. The encrypted database includes wrapped file keys, but the wrapping keys are held by a separate service. A file being encrypted is useful protection, yet it is not enough information to finish the assessment. The team checks whether the laptop also held cached plaintext, unwrapped file keys or a credential authorized to

request unwrapping. It disables the lost device's service access and reviews relevant operation records through an appropriate incident process.

Now suppose the team discovers that a troubleshooting package on that laptop also contained an exported old wrapping key. The inventory links that key to several thousand wrapped document keys. An attacker with the package and copied database may already be able to recover those document keys. The absence of a later unwrap call in the central service's logs cannot exclude this offline use. The compromise scope is therefore larger than the set of centrally recorded operations after the laptop disappeared.

The team stops using the exposed key for new protection and distributes the required revocation or authorization changes. It also protects the response process: creating new keys through the same compromised administrator account would not establish a trustworthy new state. Replacement material and credentials must be generated, approved and installed through a route whose integrity is reasonably established. Restoring the affected laptop from an old image without resolving the cause could simply reintroduce the exposure.

If new protection requires independence from the exposed data keys, the migration creates fresh data keys and uses the selected encryption construction correctly for the new records. Merely rewrapping the exposed keys would preserve their earlier values. Migration also needs to handle backups and replicas, verify successful recovery of the permitted data, and prevent unapproved rollback to an old active version. A failed or partial migration should not cause silent deletion of the only readable copy.

For already copied documents, the team must separate what it can change from what it cannot establish. New access rules can stop the service from honoring future requests. New keys can protect future versions that the attacker does not obtain. Re-encryption cannot prove that earlier plaintext was never read, or remove earlier plaintext from an attacker's storage. The assessment should describe this uncertainty accurately rather than treating a completed rotation as evidence that no disclosure occurred.

An unrelated release-signing key need not automatically be compromised solely because a data-wrapping key was exported. Its own storage and access path must be examined. Conversely, if the same recovery package

also contains the signing key or a credential that can authorize signatures, the signing trust arrangement requires its own response. This is the practical value of separating key purposes and keeping their dependencies visible. One common administrator or backup path can reconnect capabilities that appear separate in an architecture drawing.

The response record should identify affected key versions and operations, the basis for the exposure window, actions taken and what evidence remains unavailable. Logs should support accountability without copying plaintext keys, passwords or sensitive message contents into the record. Appropriate parties need the information required to stop relying on revoked material. Removing an old identifier from one interface is not sufficient if other relying systems still accept the old key.

After containment and migration, test the restored service's security decisions as well as its ability to read a sample file. An old credential should no longer authorize the forbidden operation. A new key should be selected for new protection, while controlled historical processing follows the stated policy. Monitoring should cover unexpected uses and incomplete dependencies. A clean test run supplies evidence about the tested paths; it does not retroactively prove that no attacker used the earlier secret.

This chapter completes the move from choosing primitives to managing their authority over time. Good key management connects generation, identity, access, recovery and retirement through explicit relationships. It treats ordinary failure and malicious access as different events while preparing for both. The next chapter applies these ideas to a secure connection, where long-term identity keys, short-lived agreement secrets and traffic keys have separate jobs within one protocol.

Questions to check your understanding

1. Why is an operating-system-backed cryptographic generation interface different from hashing the current time? Does a long key string by itself establish enough unpredictability?
2. What should a key inventory record, and why should an ordinary inventory normally contain references rather than all the raw secret keys?
3. Explain the different protection needs of a public verification key and a secret data key. Does a matching fingerprint supplied next to an untrusted key establish its owner?
4. How can an attacker misuse a key that cannot be exported from a module? Distinguish control of key bytes from permission to request operations.
5. In the envelope equations, identify
$$K_D$$
,
$$K_W$$
,
$$C$$
,
$$W$$
,
$$N$$
and
$$A$$
. What does rewrapping change, and what remains unchanged?
6. Distinguish a key's originator-usage period from its recipient-usage period. Why might an old decryption key be retained while an old private signing key is destroyed?
7. Why does revoking a key not make an attacker's copied ciphertext and secret key stop working mathematically? What other dependency can make a replacement ineffective?

8. Which information must be available to restore the encrypted file in the program? Why is a private signing key not required merely to verify earlier signatures?
9. Distinguish split knowledge, dual control and cryptographic erase. Why does deleting one variable or file not demonstrate complete key destruction?
10. In Nora's lost-laptop case, why does an exported old wrapping key enlarge the exposure? Why would merely rewrapping the same data keys be an incomplete response?

15. Securing a Connection with TLS 1.3

Part V: Building Secure Systems

Nora opens her document service from a hotel network. Her browser must send a password and receive private files, but the network between browser and service is outside her control. Someone along that route could read packets, change them or direct the browser to a different machine. Encrypting a file before sending it addresses only part of this problem. The browser also needs evidence about the service it has reached, agreement on fresh protection keys, and a way to reject changes to the continuing conversation.

Transport Layer Security, or TLS, is a protocol for establishing and using a protected communication channel between applications. A **client** starts the exchange, and a **server** answers it; here these are Nora's browser and the document service. TLS combines several earlier ideas in a defined order: key agreement, authentication, hashing, key derivation and authenticated encryption. Its security comes from the relationship between those operations, including the messages and identities they protect. Merely finding all those algorithms in a program would not establish that the program implements TLS correctly.

We will follow a certificate-authenticated TLS 1.3 connection, then examine resumption and its different guarantees. The current base specification is RFC 9846, a July 2026 update that preserves the TLS 1.3 version number and compatibility while clarifying and strengthening requirements. Older analyses and examples often refer to its predecessor, RFC 8446. We will identify that boundary when using a published trace, rather than treating an old experiment as a complete test of today's requirements. The main route needs no programming; two optional listings reproduce selected calculations with public test data. [\[85\]](#)

15.1 The problem a secure channel solves

A **secure channel** protects a sequence of communications under specified assumptions about the endpoints, credentials and cryptographic operations. Confidentiality limits what an observer can learn about protected contents. Integrity and data-origin authentication allow a receiver to reject records that do not pass the required checks. Peer authentication connects the exchange to an intended participant through a trust arrangement. These goals overlap, but they answer different questions: hiding a password from an observer does not establish that the machine receiving it is the correct service.

TLS has two main parts. The **handshake** establishes the connection's cryptographic parameters, performs the selected authentication and derives keys. The **record protocol** uses those keys to protect successive pieces of communication. Some handshake messages themselves travel inside protected records. This creates an important distinction between possessing enough material to encrypt the next message and having finished authenticating the other party. The first protected handshake record can arrive before the receiver has validated the server's identity.

For the ordinary server-authenticated connection described here, the browser checks the server, while the server does not receive a client certificate. Nora may subsequently log in with an application password or another credential inside the channel. That login is separate from TLS server authentication. TLS can also authenticate a client through a certificate when the application and configuration require it. The direction of authentication must therefore be stated: saying that the connection is authenticated does not by itself tell us which participant has been identified to whom.

The channel's boundary is equally important. If a service uses a front-end machine to terminate TLS and then forwards a request to another machine, the original TLS connection ends at that front end. A second protected connection may secure the next hop, but the first connection is not one uninterrupted cryptographic channel to the final process. Likewise, TLS normally allows each endpoint to read the plaintext it receives. It does not make an untrusted service unable to read a file that Nora deliberately uploads.

Nor does TLS make a remote machine honest. An attacker can operate a site with a valid certificate for the attacker's own domain. The certificate can support that domain association without establishing that the offered investment is sensible, that a download is safe or that a message is truthful. A correctly authenticated channel to the wrong intended destination still gives the wrong practical result. The application must select the intended service and make appropriate authorization decisions after the channel is established.

Visible information also remains. Network addresses, timing, connection duration and record lengths can support traffic analysis. Padding can obscure some lengths, but does not remove every observable pattern. In the base handshake we describe, the initial hello messages are not encrypted by the newly established connection keys, and the usual server-name indication can reveal a requested name. Additional mechanisms can change parts of that exposure; ordinary TLS 1.3 alone should not be described as making all connection metadata private.

Finally, a channel does not guarantee availability. A network attacker can discard traffic or break a connection without solving the key-agreement problem. TLS can report many detected failures, but cannot force delivery through a hostile network. The useful objective is more precise: a suitably authenticated connection can protect accepted communication against the specified network attacks, while failures remain failures. Continuing after a certificate error or accepting unauthenticated plaintext would replace that objective with a weaker one.

15.2 The handshake in messages

A **cipher suite** in TLS 1.3 selects an authenticated-encryption algorithm and a hash used in key derivation and handshake authentication. For example, `TLS_AES_128_GCM_SHA256` selects AES-128-GCM for records and SHA-256 for those hash-dependent operations. It does not also specify the server's certificate type or the key-agreement group. Those are negotiated through separate fields and extensions. An RSA certificate can therefore authenticate a connection whose agreement uses X25519 and whose records use AES-GCM; RSA is not encrypting the traffic keys in that exchange.

The client begins with **ClientHello**. It offers supported versions, cipher suites and other capabilities. For a common ephemeral agreement, it also sends a **key share**: a public value generated for that exchange. Here **ephemeral** means that the agreement's private material is temporary, generated for this exchange and erased when no longer needed. With X25519, the private input remains at the client while the public output is sent. The client's list of supported groups describes acceptable choices; the actual key-share list provides values already prepared. These lists have related purposes, but they are not identical.

The server returns **ServerHello**, identifying its selections and supplying its corresponding public share. Each side can now calculate the shared agreement value using its own private input and the other's public value. Chapter [12](#) explained why the values match for an allowed exchange. TLS then feeds that shared value into its key schedule; it does not use the raw output directly as an AES key. Fresh private inputs and the required peer-input checks are essential. RFC 9846 expressly forbids reusing key-share values across connections.

If the server supports an offered group but needs a share that the client did not send, it can issue a **HelloRetryRequest**. The client supplies an appropriate second ClientHello under the protocol's rules. This adds an exchange and changes transcript processing in a precisely defined way. It is not permission for the server to invent any new group or for the client to silently weaken its policy. Our main diagram omits this branch so that the normal path remains readable.

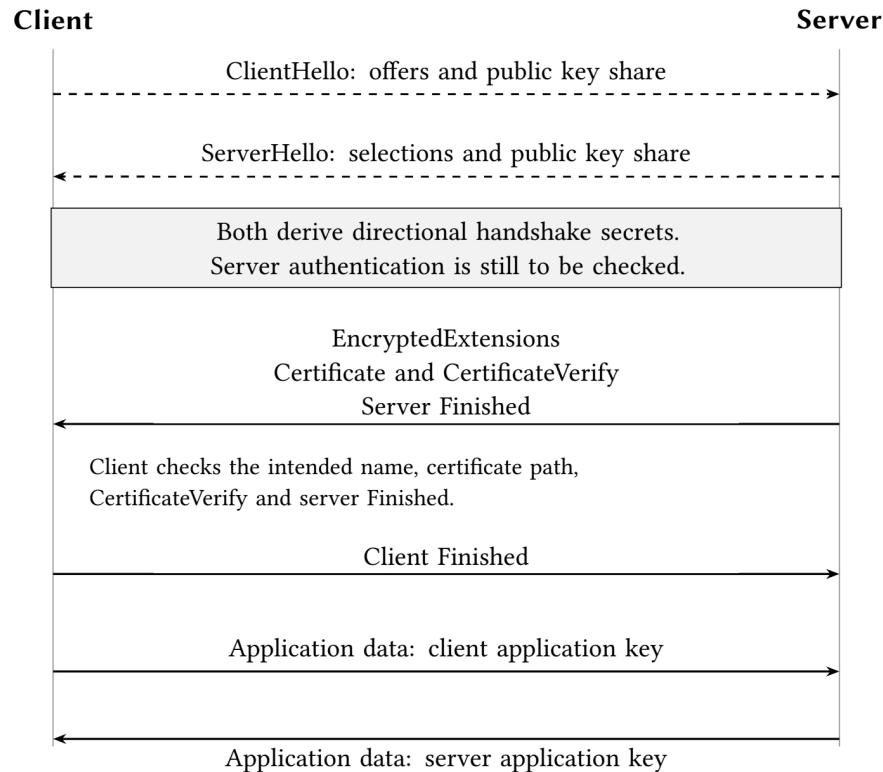


Figure 15.1. An ordinary TLS 1.3 certificate-authenticated exchange with ephemeral key agreement. Dashed message arrows are cleartext in this base exchange; solid message arrows are protected. Each named group of messages can span several records. Retry, early data and optional client-certificate messages are omitted. The server identity is checked during the protected handshake, not proved merely by successful decryption.

After **ServerHello**, the parties can derive separate handshake traffic secrets. The server sends **EncryptedExtensions** for selected parameters that do not have to appear in **ServerHello**. In the certificate-based path it then sends its **Certificate** message, **CertificateVerify** and **Finished**. **Certificate** carries the certificate information; **CertificateVerify** demonstrates possession of the associated private signing capability and binds the exchange to the transcript. **Finished** checks a keyed value calculated from the conversation so far. The next section distinguishes these checks in detail.

The client validates the server's authentication and **Finished**, then sends its own **Finished** under its client handshake keys. If the server requested certificate-based client authentication, the client's **Certificate** and, when applicable, **CertificateVerify** precede that **Finished**. Once the required

checks and sending steps are complete, ordinary application traffic uses application keys rather than handshake keys. The change of keys is part of the protocol state, not a guess made by inspecting whether the next plaintext looks like a web page.

This normal path is called a **one-round-trip**, or 1-RTT, handshake because the client can send ordinary application data with handshake completion after receiving the server's response flight. A **flight** is a group of messages sent before waiting for the other side. The count describes TLS message dependencies; it excludes such work as resolving a name and establishing an underlying TCP connection. It is not a promise that every real connection takes exactly one network round trip in total.

The server may send application data after its own Finished without yet receiving the client's Finished. At that point it has not obtained the corresponding evidence of the client's liveness or, where requested, completed client authentication. Applications must respect that stage distinction. On an ordinary server-authenticated connection, the browser still needs all required server checks before treating received application bytes as trusted service data. An early start by one side does not move every security decision earlier for the other.

Handshake messages, records and network packets also have different boundaries. A record can contain more than one handshake message, and a handshake message can cross record boundaries. The transport can split record bytes across several packets. Reading only a convenient packet's contents and calling it the whole transcript would therefore be incorrect. A TLS implementation reconstructs and processes the specified structures in the allowed order, even when network delivery fragments them differently.

[\[85\]](#)

15.3 Authentication and the transcript

A **handshake transcript** is the ordered sequence of handshake messages covered by a particular calculation. TLS hashes their encoded bytes, including handshake type and length fields, but not the enclosing record headers. This is not a summary written by the application. It is a precisely delimited byte history. For a path without HelloRetryRequest, a compact description is

(15.1)

Here
 M_1
through
 M_j
are the handshake messages included at that stage, in order;

j

tells us where that stage ends. The sign
||

means concatenate the bytes.

H

is the cipher suite's hash, and

T_j

is its output. With SHA-256, that output is 32 bytes regardless of the transcript's length. Increasing

j

includes more messages, so two checks at different handshake stages normally use different transcript hashes.

There is a special rule when HelloRetryRequest occurs. The first ClientHello is replaced in the ongoing transcript by a synthetic handshake message containing its hash, before the retry and later messages are included. Thus the simple formula above must be read with its stated no-retry condition. Real code follows the specified encoding and retry transformation. Neither dropping the first ClientHello nor hashing a textual description of it would implement that rule.

First consider the certificate check. The browser validates a certification path against its configured trust anchors, with the applicable validity, constraints and usage checks. It also checks whether the end-entity certificate represents the service it intended to reach. For a DNS service name, this uses the relevant subject alternative name and matching rules; an

arbitrary certificate subject's common name is not a substitute under RFC 9525. A valid path for `files.example` does not authenticate `shop.example`. The reference identity must come from the application's intended destination, not be chosen solely from whatever certificate the peer supplied.^[10]

Server Name Indication, or SNI, is a ClientHello extension commonly used to tell a hosting server which service the client wants. It helps a server choose parameters and certificates for a name. Sending SNI does not itself perform service-identity verification. Similarly, Application-Layer Protocol Negotiation, or ALPN, allows selection of an application protocol, such as a particular HTTP version. The TLS library and application must handle those selected parameters correctly; an authenticated byte channel still needs an agreed interpretation of its bytes.

CertificateVerify adds a different kind of evidence. The server signs a structured input containing 64 space bytes, a fixed context naming the server's TLS 1.3 CertificateVerify role, a zero separator byte, and the transcript hash through its Certificate message. The client checks that signature with the certificate's public key and the selected signature scheme. The context separates this signature from client signatures and other uses. It is inaccurate to describe CertificateVerify as merely signing the server's key share: it binds the preceding handshake history under a particular role and encoding.

The signature scheme and the transcript hash have separate jobs. A suite ending in SHA256 chooses the transcript and key-schedule hash; it does not mean every certificate in a path must have been signed with SHA-256, or that every permitted CertificateVerify scheme has the same internal hashing rule. Algorithm compatibility and the negotiated signature rules still apply. This is one reason TLS carries explicit algorithm identifiers rather than assuming that the word *secure* selects one universal set of operations.

Finished then provides a keyed check of the handshake and its derived secret. Let

S

be the sender's handshake traffic secret and let

h

be the selected hash output length in bytes. Define

as TLS's labeled HKDF expansion with secret

S

, label

l

, context bytes

C

and requested output length

n

. Its encoding is explained in the next section. For the main handshake:

(15.3)

K_F

is a separate Finished key.

ϵ

denotes an empty byte string, not a string of zero bytes.

T_{before}

is the transcript hash immediately before this sender's Finished, and

V

is the verify-data value carried in it. First derive

K_F

from the directional secret; then calculate HMAC over that stage's transcript hash. For SHA-256 the key and verify-data value here are each 32 bytes.

The receiver calculates the expected value from its own corresponding secret and transcript, and checks the received one through the implementation's specified verification path. The Finished message being checked is excluded from its own input. The client's later Finished covers the server's Finished as part of the preceding transcript. Otherwise the endpoints could disagree about what had been exchanged while informally

claiming to have authenticated the same conversation. Failure of CertificateVerify or Finished is a handshake failure, not a warning that an application should bypass to keep a connection working.

These checks support different conclusions. A certificate path and matching name establish a trusted key association under policy. CertificateVerify connects possession of the signing key to this handshake. Finished connects the handshake history to the computed handshake secret. None alone replaces the other two in the certificate-based path. Successful decryption of the earlier protected Certificate message is especially insufficient: an active intermediary could establish a separate unauthenticated agreement with the client. The later authentication checks are what prevent that intermediary's exchange from being accepted as the intended server's exchange. [\[85\]](#)

Nor is a TLS-protected application message automatically a transferable digital signature by the server. Traffic protection uses shared secret material and the protocol's direction-specific state. The client also derives the corresponding keys needed for its side of the channel. TLS authenticates communication to the participants under its assumptions; it does not turn every received web page into independently verifiable evidence that only the server could have created it.

15.4 Deriving separate traffic keys

A **key schedule** is the defined sequence of derivations producing secrets and keys for different purposes. TLS 1.3 uses HKDF, introduced in Chapter [2](#), with its Extract and Expand operations. Extract combines input keying material with a salt into a pseudorandom key. Expand derives output of a specified length with context information. This is not password stretching: the usual inputs here are an agreement secret and, where applicable, an already established pre-shared key, rather than a human password.

For the main example, let

Z

denote the byte string produced by the selected ephemeral agreement. Let

P

denote a **pre-shared key**, or PSK, if the mode uses one. A PSK is secret material that both parties already possess before this exchange; it is not a public session identifier. Set

$$P = 0^h$$

when no PSK is selected, where

$$0^h$$

means

$$h$$

zero bytes. We use

$$X(s, x)$$

for HKDF-Extract with salt

$$s$$

and input material

$$x$$

. A reduced schedule is

$$(15.8)$$

Read downward.

$$E$$

is the early secret.

$$B_E$$

is an intermediate value used as the next salt. Extract then combines it with

$$Z$$

to obtain the handshake secret

$$G$$

. A second labeled derivation produces

$$B_G$$

, and another extraction with a defined zero input produces the main secret

$$M$$

. These names describe stages, not directly usable record keys. The current specification calls

$$M$$

the main secret; older traces call it the master secret. The defined derivation labels retain their earlier byte values for compatibility.

The distinction between

$H(\varepsilon)$

and

ε

matters. The first is the hash of an empty message: 32 specific bytes for SHA-256. The second contains no bytes. The intermediate `derived` steps use the first; traffic-key and Finished-key expansions use an empty context. Replacing either by the other changes the encoded input and result. Likewise, absent PSK input means the defined zero-byte string, not skipping the early-secret stage.

The TLS expansion

L

constructs its HKDF information from the requested output length, a length-prefixed label beginning with the six ASCII bytes `tls13`, and length-prefixed context bytes. The output length has a two-byte encoding; the label length and context length each have a one-byte encoding. The fixed prefix includes a trailing space, and the label has no terminating zero byte. These boundaries make the inputs exact. Choosing another label or transcript does not merely rename a key: it changes the derivation.

For handshake protection, let

T_{CHSH}

be the transcript hash through ClientHello and ServerHello in our no-retry path. Directional handshake secrets are

(15.10)

The subscript

C

identifies client-sent traffic and

S

identifies server-sent traffic;

h

in these subscripts identifies the handshake stage. Both endpoints calculate both values. The client writes with material derived from

$S_{c,h}$

, and the server reads that direction with matching material. The server's outgoing direction uses

$S_{s,h}$

. They are separate secrets even though both derive from the same handshake secret and transcript.

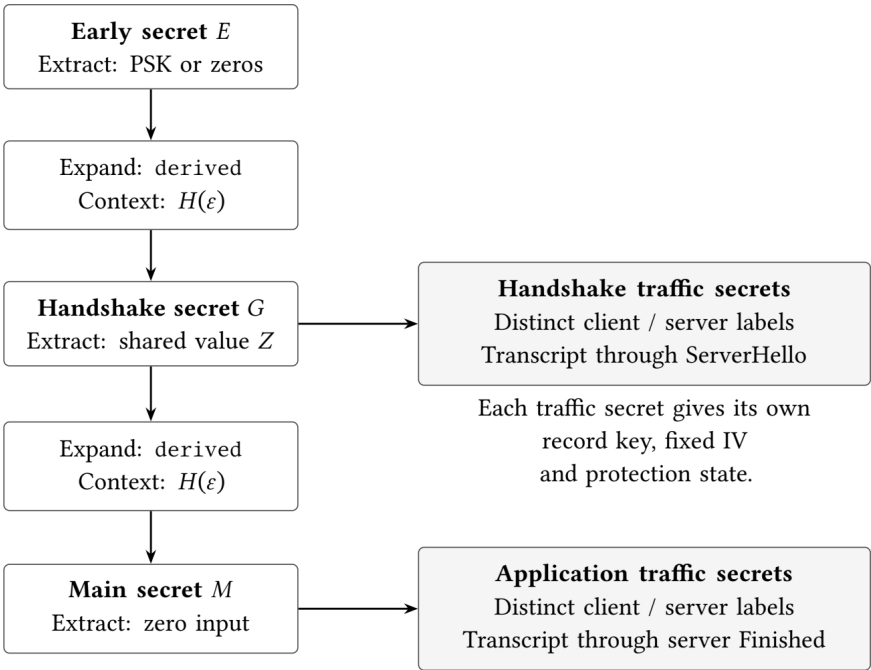


Figure 15.2. A reduced TLS 1.3 schedule for the ephemeral-agreement route. The vertical stages combine inputs; side branches add purpose labels and stage-specific transcript hashes. A traffic secret is further expanded into a record key and IV. Binder, early-data and exporter branches are omitted. The diagram does not represent PSK-only agreement or the complete protocol state machine.

Application traffic uses another pair. Let

T_{SF}

cover the handshake through the server's Finished. Then

The zero identifies the first application-secret generation. Both directions use the transcript ending at the server's Finished for this initial derivation, even though the client sends its own Finished later. The resumption secret uses a later transcript ending at the client's Finished. Calculation timing, transcript coverage and permission to send a particular message are distinct rules; obtaining an application secret does not bypass unfinished authentication.

From any appropriate directional traffic secret

S

, the record layer derives a key

K

and a fixed IV

I

:

(15.13)

k

is the selected AEAD key length in bytes and

v

its TLS IV length. For AES-128-GCM here,

$k = 16$

and

$v = 12$

. A SHA-256 traffic secret is 32 bytes, yet the resulting AES key is 16 bytes because the requested role and length determine that output. Traffic secret, record key and IV are therefore different objects, with different dependencies and lifetimes.

Reproducing a published derivation

RFC 8448 publishes TLS 1.3 example traces from 2019, including intermediate secrets and test keys. The following optional Python calculation uses Section 3's shared value and hello-transcript digest to reproduce selected outputs. Its HMAC implementation needs only the Python standard library. We deliberately supply that digest rather than print hundreds of handshake bytes here. This reproduces the downstream derivation, not parsing, identity checking or the whole handshake. [\[86\]](#)

Listing 15.1. Selected derivations from the public RFC 8448 trace. All supplied secrets are public demonstration values.

```
import hashlib
import hmac

def extract(salt, material):
    return hmac.digest(salt, material, "sha256")

def expand_label(secret, label, context, length):
    # This example supports one SHA-256 expansion block.
    if not 1 <= length <= 32:
        raise ValueError("example length must be 1 to 32")
    name = b"tls13 " + label
    info = (length.to_bytes(2, "big")
            + bytes([len(name)]) + name
            + bytes([len(context)]) + context)
    return hmac.digest(secret, info + b"\x01",
                       "sha256")[:length]

zero = bytes(32)
empty_hash = hashlib.sha256(b"").digest()
shared = bytes.fromhex(
    "8bd4054fb55b9d63fdbacf9f04b9f0d"
    "35e6d63f537563efd46272900f89492d")
hello_hash = bytes.fromhex(
    "860c06edc07858ee8e78f0e7428c58ed"
    "d6b43f2ca3e6e95f02ed063cf0e1cad8")

early = extract(zero, zero)
base = expand_label(early, b"derived", empty_hash, 32)
handshake = extract(base, shared)
server_hs = expand_label(
    handshake, b"s hs traffic", hello_hash, 32)
key = expand_label(server_hs, b"key", b"", 16)
iv = expand_label(server_hs, b"iv", b"", 12)

assert server_hs.hex() == (
    "b67b7d690cc16c4e75e54213cb2d37b4"
    "e9c912bcded9105d42befd59d391ad38")
assert key.hex() == "3fce516009c21727d0f2e4e86ee403bc"
assert iv.hex() == "5d313eb2671276ee13000b30"
print("Published handshake derivation matched")
```

For these requested lengths, HKDF-Expand needs only its first SHA-256 block. That block is HMAC over the encoded information followed by the

counter byte 01; the listing then takes the requested number of leading bytes. It explicitly rejects longer outputs, for which a complete HKDF implementation would need the further chained blocks. The two short expected outputs distinguish the record key from the IV, while the longer assertion checks the preceding traffic secret. No random-looking output is accepted merely because its length seems right.

This is a historical test, not a proposed certificate or live key configuration. The original trace also contains public private keys and an old demonstration certificate. Those are not credentials to trust in a current connection. Matching the selected bytes supports the explanation of the key schedule; it does not establish service authentication or compliance with every updated requirement of RFC 9846.

15.5 Records, nonces and key updates

A **TLS record** is a framed unit handled by the record protocol. For protected records, TLS forms an inner plaintext containing content bytes, an inner content-type byte and any zero padding. AEAD protects that structure and authenticates the specified outer record header as associated data. The inner type distinguishes such uses as handshake messages, application data and alerts. The outside of an encrypted record normally uses an application-data type even when its protected content is a handshake message, so the visible outer type does not reveal that distinction.

The outer legacy version bytes also have a compatibility role. Seeing the value associated with an older TLS version in that header does not establish that the connection negotiated that older protocol. TLS 1.3 version negotiation uses the supported-versions extension. A reader must interpret a field according to its defined role, rather than infer the active protocol from one convenient byte pattern. The protected header length counts the encrypted fragment, including the authentication tag, and the receiver must enforce framing and size rules.

Each direction maintains its own 64-bit record sequence number. The number begins at zero for the first record under a particular traffic key and increases for successive records under it. It is not transmitted as a separate explicit sequence field in each TLS 1.3 record. Both sides track it in connection state. If an attacker inserts, deletes or reorders protected records

in this stream, the receiver's expected sequence-dependent protection no longer matches the transmitted ciphertext. A failed authentication check must not release the unverified plaintext to the application.

TLS constructs the per-record nonce from that number and the fixed IV:

$$(15.14)$$

q

is the sequence number,

I

the directional fixed IV from the key schedule and

v

its byte length.

$\text{Pad}_v(q)$

encodes the 64-bit number in big-endian order and adds zero bytes on the left to reach

v

bytes. The symbol

$\oplus$

means bitwise XOR. With a 12-byte IV, the padded zero counter is twelve zero bytes, so the first nonce equals

I

. The next padded counter ends in 01, changing the IV's last byte by XOR.

For the public server-handshake IV in Listing [15.1](#), the last byte is 30. At sequence zero it remains 30; at sequence one it becomes 31, because hexadecimal $30 \text{ XOR } 01 = 31$. At sequence two it becomes 32. This example explains the operation without printing a fresh nonce for every message. Distinct counters give distinct nonces under the same IV, but that statement depends on keeping the counter within range and maintaining the correct connection state.

The AEAD operation can be written as

(15.15)

P_q

is the complete inner plaintext,

A_q

is the encoded outer header used as associated data, and

C_q

includes the encrypted content and tag.

K

and

N_q

are the record key and that record's nonce. The receiver uses the matching decryption operation and accepts only if all required checks succeed.

Chapter [7](#) explains the AEAD security conditions; TLS's contribution here is to define their inputs and maintain the state consistently.

Opening one public record

Continue the earlier Python session with Listing [15.2](#). It needs the `cryptography` package for AES-GCM. The supplied digest is the same trace's transcript through the server Finished. The ciphertext is its first client application record, with a five-byte outer header and sequence number zero. No network connection is made. The code retains the inner type for inspection instead of pretending that all decrypted bytes are application content.

Listing 15.2. Continuing the RFC 8448 example to recover a published application record and reject an altered one.

```
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

base = expand_label(
    handshake, b"derived", empty_hash, 32)
main_secret = extract(base, zero)
server_finished_hash = bytes.fromhex(
    "9608102a0f1ccc6db6250b7b7e417b1a"
    "000eaada3daae4777a7686c9ff83df13")
client_app = expand_label(main_secret, b"c ap traffic",
                           server_finished_hash, 32)
```

```

app_key = expand_label(client_app, b"key", b"", 16)
app_iv = expand_label(client_app, b"iv", b"", 12)

record = bytes.fromhex(
    "1703030043a23f7054b62c94d0affa"
    "fe8228ba55cbefacea42f914aa66bc"
    "ab3f2b9819a8a5b46b395bd54a9a"
    "20441e2b62974e1f5a6292a2977014"
    "bd1e3deae63aeebb21694915e4")
header, sealed = record[:5], record[5:]
assert int.from_bytes(header[3:], "big") == len(sealed)
# Sequence number zero: nonce equals the fixed IV.
inner = AESGCM(app_key).decrypt(app_iv, sealed, header)
assert inner == bytes(range(50)) + b"\x17"
print("Recovered 50 bytes; inner type is application data")

altered = sealed[:-1] + bytes([sealed[-1] ^ 1])
try:
    AESGCM(app_key).decrypt(app_iv, altered, header)
except InvalidTag:
    print("Altered record rejected")
else:
    raise AssertionError("altered record was accepted")

```

The recovered content is the byte sequence from 0 through 49, followed by inner type 17, the hexadecimal application-data value. This particular record has no padding. A full receiver must correctly find and validate the inner type after removing permitted trailing padding; it cannot always remove one last byte without examining the structure. The changed final byte in the negative test damages the authenticated ciphertext/tag input and produces failure. That is useful evidence for this path, not a complete record-parser test.

A sequence counter must never wrap under the same key. Even before that boundary, AEAD-specific usage limits can require replacing keys or closing the connection. Those limits depend on the algorithm, record volume and the security analysis; they are not simply the number of distinct 64-bit counters available. The current base specification requires acting before the applicable sending limits are reached. A long-lived connection must therefore track cryptographic usage rather than assume that a large counter makes every amount of encrypted traffic safe.

For application traffic, **KeyUpdate** advances one sending direction to new secret material. If

$$S_{d,n}$$

is direction

d

's application traffic secret at generation

 n

, its successor is

(15.16)

Here

 d

identifies client or server sending, and

 $n + 1$

is the next generation number. The sender derives a new record key and IV from that successor using Equation [15.13](#). It sends the KeyUpdate message under the old keys before switching. The receiver processes that message before accepting the new generation, and the affected sequence counter starts again at zero under its new key. A request can also ask the peer to update its own sending direction.

This is a one-way derivation, with an important limitation. If an attacker learns the current traffic secret, that attacker can calculate its future updates. KeyUpdate introduces no fresh independent agreement secret and does not by itself restore security after such compromise. Erasing old secrets can help protect earlier generations from later exposure, but recovery from known current traffic material requires a trustworthy fresh establishment and resolution of the compromise. RFC 9846 also limits repeated update generations and update requests; an implementation must not treat KeyUpdate as an unlimited way to prolong one connection. [\[85\]](#)

15.6 Resumption, 0-RTT and replay

Session resumption lets a later connection use secret material established through an earlier one. After the required client Finished, a server can send NewSessionTicket. The client associates the ticket with a resumption PSK derived from the earlier connection's resumption secret

and the ticket nonce. The ticket is an opaque identifier or protected object used by the server's design; the PSK is the secret used in the later handshake. Possessing the visible ticket alone should not be confused with knowing that secret.

On a later visit, the client can offer a ticket identity and a **binder**, a keyed value calculated over the required portion of its new ClientHello using a key derived from the offered PSK. The server checks the applicable ticket and binder rules before accepting that PSK. The binder binds the offer to possession of the PSK under this structure. It is not a new certificate signature, and a valid binder alone does not prove that a ClientHello was freshly sent rather than replayed.

Resumption can combine the PSK with fresh ephemeral Diffie–Hellman agreement. This mode derives new handshake and ordinary application secrets using both inputs. A PSK-only mode exists too, with no new Diffie–Hellman contribution. Their security properties are different. **Forward secrecy** means that earlier session material remains protected against later disclosure of specified long-term secrets, under the construction's assumptions and appropriate erasure of temporary material. Here a fresh ephemeral contribution can protect recorded ordinary traffic against later PSK disclosure. A PSK-only exchange does not gain that property merely by calling the connection resumed. Exposure of the live agreement or traffic secret is a different event.

The connection's authentication also follows the PSK's trust history and allowed context. In certificate-based resumption, that secret was established through an earlier authenticated connection. The application and TLS stack must enforce the relevant service identity, lifetime and parameter constraints. Resumption is not authority to use an old ticket for an unrelated service or to ignore a changed trust policy. External PSKs provisioned outside TLS need their own secure distribution and identity associations, with care about which parties share the same secret.

Zero-round-trip data, or 0-RTT data, allows an eligible client to send protected application bytes with its initial flight, before receiving fresh server messages for that connection. The early traffic secret derives from the offered PSK and ClientHello, so the client can calculate it at that point. A fresh server Diffie–Hellman contribution cannot protect bytes already

sent before that contribution exists. Consequently, 0-RTT data does not inherit the same forward-secrecy guarantee as ordinary traffic from a fresh ephemeral exchange.

Table 15.1. Different TLS 1.3 paths answer different security questions. These entries assume correct processing and the relevant protected credentials. Forward secrecy here concerns later disclosure of the named long-term authentication material, rather than exposure of the live traffic or ephemeral secret.

Protected traffic Ordinary certificate-based traffic	Fresh agreement contribution Ephemeral agreement in the path described here	Main distinction Later signing-key disclosure alone does not recover erased earlier agreement secrets.
Protected traffic Ordinary PSK with ephemeral agreement	Fresh agreement contribution PSK and fresh agreement	Main distinction Can retain forward secrecy against later PSK disclosure under its conditions.
Protected traffic Ordinary PSK-only traffic	Fresh agreement contribution No new asymmetric contribution	Main distinction Does not gain that forward secrecy against later PSK disclosure.
Protected traffic 0-RTT application data	Fresh agreement contribution Sent before fresh server contribution	Main distinction Replayable across connections; no corresponding forward secrecy against later PSK disclosure.

Early data has another limitation: **replay**. An attacker can capture an initial flight and send it again without learning its plaintext or forging a new AEAD tag. Each accepting server instance can derive the corresponding early keys from the replayed offer. A valid authentication tag shows that bytes match the construction; it does not show that this is their first accepted appearance across the service. Protection against record reordering within one established connection therefore does not imply protection against replaying an early flight into another connection.

Imagine Nora sends a request to purchase a second copy of a document as early data. If replaying that request creates another charge, the operation is unsafe merely because it is encrypted. The application needs an appropriate duplicate-handling design, or must wait until the ordinary handshake stage. A request identifier helps only when it is associated with the authorized operation and checked against sufficiently reliable state. Accepting every fresh network connection as a fresh business request would defeat that intended protection.

Servers can reduce early-data replay by tracking accepted offers or using single-use tickets, with the required coordination across their deployment. RFC 9846 requires at least that one server instance not accept the same 0-RTT handshake more than once, while stronger deployment-wide protection is encouraged. Separate machines with disconnected acceptance state can still create a wider replay boundary. Ticket lifetime and timing checks alone do not turn the early exchange into proof that an application action happened exactly once.

Even an operation considered **idempotent**, meaning that repetition has the same intended effect, can incur resource costs or expose timing information when repeated. Its safety has to include those consequences. Moreover, ordinary application retries can duplicate requests after connection failures even without 0-RTT. Early-data rejection and automatic retry add another path to that problem. TLS cannot alone promise exactly-once business processing; the application must define and enforce what repeated actions mean.

The server may reject early data while continuing an otherwise acceptable handshake. The client must learn whether the early bytes were accepted before deciding how its application should proceed. Blindly resending an operation whose earlier outcome is uncertain can cause duplication. A practical reader should therefore ask two separate questions: does the library support resumption, and is this particular application operation suitable for early transmission under its replay and retry policy? Faster setup is useful only when the weaker early-stage guarantees fit the task. [\[85\]](#)

15.7 What the model and implementation assume

Cryptographic analysis makes the channel's reasoning more precise, but its conclusion has a scope. Dowling, Fischlin, Günther and Stebila's 2021 journal analysis studies the TLS 1.3 handshake as specified in RFC 8446. It uses a multi-stage model: different derived secrets can have different authentication, forward-secrecy, use and replay properties. That approach explains why treating the entire handshake as one event that produces one uniformly secure key would hide important distinctions. A secret can

become available before the protocol has established all the authentication evidence associated with later stages. [\[87\]](#)

The work analyzes full certificate-based, PSK-only and PSK with ephemeral agreement modes under stated cryptographic assumptions. Those assumptions concern the selected primitives and the allowed attacker capabilities. Its compromise model includes long-term and stage-key exposure but does not include every form of ephemeral randomness or full session-state exposure. This boundary matters: an attacker who steals the live ephemeral secret is not merely learning a public transcript. The result cannot be quoted as proving security after every possible loss of endpoint state.

The paper is also an analysis of a cryptographic abstraction of the handshake, not certification of an arbitrary TLS product. It analyzes the selected mode and algorithm combination rather than the complete negotiation among possible choices. Certificate validation, service-name selection, parsing, record handling, memory safety and application decisions need their own correct behavior. Nor does a 2021 proof automatically establish every clarified or strengthened requirement in a 2026 specification. Its explanations remain valuable when the construction and model match the claim being made. A careful account identifies the analyzed version and property instead of attaching the word proved to a whole deployment.

Implementation evidence answers another set of questions. In the collected OpenSSL source, the TLS 1.3 key-schedule code separates secret derivation, key and IV expansion, and traffic updates. The record implementation constructs a nonce by XORing the IV with sequence bytes and prepares the record header as associated data. The client and server state-machine files handle which handshake messages are expected next. These are related responsibilities in different code paths; correctness requires them to agree about the stage, direction and currently active keys. [\[88\]](#)

These source locations are useful reading routes, not instructions to assemble a private TLS replacement from snippets. Checking that a nonce calculation appears in one function does not establish that counters can never repeat elsewhere, or that every error path discards unauthenticated

bytes. Likewise, reproducing one RFC trace establishes matching computations for those inputs. Broader confidence needs appropriate conformance tests, malformed-input tests, cryptographic checks, code review and attention to the chosen library's maintained configuration.

For Nora's service, an appropriate application interface should perform the required certificate and reference-name checks rather than merely turn on encryption. Trust anchors, time handling, permitted protocols and algorithms, and the selected application protocol must be configured consistently. A connection failure should be visible to the application as a failure. Disabling verification to get past an error changes the security being offered, even if the next packet is still encrypted with AES-GCM.

The application must also respect stream and closure semantics. An authenticated sequence of received records does not prove that the sender intended a file to end at a particular network interruption. TLS has a `close_notify` alert for orderly closure of a sending direction, and applications need their own rules for complete objects and permitted termination. A truncated download should not become a complete, trusted update merely because its earlier records passed authentication.

Endpoint secrets and observability require care too. A debugging facility that records traffic secrets can enable decryption of captured traffic for troubleshooting; that same record is sensitive material. Exposing it changes the assumptions of the channel. Hardware protection for a certificate's private key does not automatically protect live traffic secrets in a separate process. Key management from Chapter [14](#) applies to each of these roles, including short-lived state and retained resumption material.

Finally, TLS 1.3 is a protocol framework, not a single permanent choice of group and signature scheme. The conventional X25519 example here is not a post-quantum key exchange, and AES-256 alone does not change that fact. Other specified establishment mechanisms require their own definitions, negotiation and analysis. This chapter explains the conventional route and its boundaries; later treatment of quantum threats must examine the actual asymmetric choices rather than infer them from the symmetric cipher's key length.

We can now follow the whole argument. The client selects an intended service, exchanges capabilities and fresh agreement values, authenticates

the resulting handshake history, and uses purpose-separated secrets to protect records with controlled nonces. Resumption changes the inputs, and early data changes what the receiver can conclude about freshness. A trustworthy channel depends on preserving those distinctions from the mathematical construction through the implementation and into the application's decisions.

Questions to check your understanding

1. What are the separate jobs of the handshake and record protocol?
Why does decrypting a protected handshake message not yet establish the intended server's identity?
2. What does `TLS_AES_128_GCM_SHA256` select? Does it imply RSA key transport, a particular certificate type or a particular agreement group?
3. In Equation [15.1](#), what are

$$M_j$$

$$,$$

$$||$$

$$,$$

$$H$$

 and

$$T_j$$

 ? Why are record headers excluded, and why does HelloRetryRequest need a special rule?
4. Distinguish certification-path validation, service-name matching, CertificateVerify and Finished. What role does SNI have?
5. In Equation [15.8](#), identify

$$P$$

$$,$$

$$Z$$

$$,$$

$$E$$

$$,$$

$$G$$

 and

$$M$$

 . Why are

$$\mathcal{E}$$

 and

$H(\varepsilon)$

different inputs?

6. Which transcript endpoint is used for the first client and server application traffic secrets? Does knowing a derived secret grant permission to skip unfinished authentication?
7. Explain the first two record nonces from a fixed IV. Why must a counter reset be accompanied by the appropriate key change, and why can usage limits require an earlier update?
8. What do the two printed programs establish? What happens if an authenticated record is altered, and which important TLS tasks are not implemented by these programs?
9. How do PSK-only, PSK with fresh ephemeral agreement and 0-RTT differ? Why does an accepted early-data tag not prove that an operation was processed only once?
10. Why does KeyUpdate not repair exposure of the current traffic secret? Why do a handshake security proof and successful test trace not establish that every TLS application is secure?

16. Secure Messages and Changing Keys

Part V: Building Secure Systems

Nora wants to send Sam a private message, but his phone is offline. A service can hold the message until he returns. The difficulty is deciding what that service must be trusted to know. If Nora encrypts only the connection to the service, the service can read the message before forwarding it over another protected connection. If she wants protection from the intermediary as well as the network, the encryption must connect the intended endpoints. She also needs a way to begin without waiting for Sam to take part in a live handshake.

End-to-end encryption protects message content between the participating endpoints, with the intended recipients obtaining the decryption capability rather than the delivery intermediary. It does not mean that every connected device, backup or person receiving a copy is outside the trust boundary. Once an endpoint displays the plaintext, its software and user can act on that plaintext. We will first examine the cryptographic path between one Nora device and one Sam device, then return to those wider boundaries.

An ongoing conversation creates another problem. A single shared secret used indefinitely would let a later disclosure expose a large part of the conversation. A **ratchet** is a mechanism that advances secret state as communication proceeds, aiming to limit what one exposure reveals. The term refers to the direction of the key evolution, rather than a physical guarantee that no software copy can remain. This chapter separates the initial agreement, per-message key changes and the fresh inputs needed for recovery after compromise. The classical Double Ratchet provides the main explanation; newer post-quantum constructions extend it under separately stated assumptions. [\[89\]](#)

16.1 Messages when the recipient is offline

Communication is **asynchronous** when the participants need not be online at the same time. Sending an email is a familiar example: the sender can finish while the recipient's device remains disconnected. For encrypted messaging, that useful feature removes the immediate opportunity to ask the recipient for a fresh public value. TLS's ordinary handshake from Chapter [15](#) expects the server to respond during connection establishment. An offline person's phone cannot follow that live exchange until it reconnects.

One solution is to prepare public information in advance. A **prekey** is a key published before a particular sender starts the initial exchange. Sam generates the required key pairs on his device and uploads their public portions. The private portions remain protected at his endpoint. Nora can later fetch the public bundle and calculate an initial shared secret. Her message includes the public inputs and identifiers that Sam will need to calculate the corresponding secret when he receives it.

The delivery server therefore performs at least two logical jobs. It distributes public setup information and holds encrypted messages for delivery. The server can be unable to decrypt ordinary message content while still having substantial control over delivery, timing and the setup information it presents. Refusing to hand out a bundle, withholding a message or presenting an unauthorized identity key are different actions. End-to-end encryption does not remove the need to examine those server capabilities.

The initial message must be self-contained enough for the recipient to identify the intended setup. It carries the relevant sender public information, prekey identifiers and encrypted content under the selected construction. Sam should not need to guess which private key was used or try unrestricted computations for an arbitrary request. An identifier selects an object within the protocol's rules; it is not itself evidence that the object is authentic or that the caller is authorized.

One-time prekeys limit the reuse of particular private setup values. A server distributes a public one-time prekey for one initial exchange and removes it from its available pool. Sam later removes the matching private value after successful processing under the protocol. These are distinct removals at different places and times. Removing a public listing does not

erase the private key, and deleting a private key before the delayed message can be processed could make the intended message unreadable.

A service must also accommodate depleted prekey pools and delayed messages. Some constructions define fallback prekeys for use when one-time material is unavailable. This improves availability but changes which retained secret values could affect old exchanges. It is not correct to say that every initial message has identical protection regardless of the setup path. The protocol version, selected keys, successful processing and actual deletion determine the relevant exposure window.

Transport protection remains useful around this design. Nora and Sam can each use TLS to connect to the delivery service, protecting those network hops while the message body remains separately encrypted for the recipient. The layers address different adversaries and endpoints. TLS ending at the service does not establish end-to-end message secrecy by itself; end-to-end content protection does not automatically conceal every fact visible on the transport connection.

The scope here is one-to-one device communication. A conversation displayed on several devices needs a defined way to authenticate those devices, establish their sessions and distribute copies. A group also needs membership and key-update rules beyond a two-party ratchet. We should understand the two-party mechanism first, while keeping those additional systems visible as separate work rather than assuming that one successful pairwise calculation solves them all.

16.2 Initial key agreement and authentication

An **identity key** is a longer-lived key used within the protocol's association between an endpoint and a participant. It is not automatically a certificate carrying a legal name. A **signed prekey** is a setup public key accompanied by a signature under the relevant identity key. That signature allows a sender who trusts the identity public key to check the prekey's association. A **one-time prekey** is intended for one initial exchange, and an **ephemeral key** is fresh private/public material generated for the exchange in progress.

The classical Extended Triple Diffie–Hellman protocol, called **X3DH**, gives a useful way to see how these roles fit together. We use it here to explain the classical foundation before introducing PQXDH in Section [16.5](#). It is a specified initial agreement, not the name for every modern messaging setup. Its public-key encodings, signatures, derivation and message processing belong to the selected specification; a few DH calculations alone do not implement the protocol.^[94]

Table 16.1. Key roles in the classical initial-agreement example. Lowercase names identify private inputs and uppercase names the corresponding public values. The table describes their jobs, not a universal replacement schedule.

Role	Private / public	Purpose in this exchange
Nora identity		Contribute Nora’s identity-key relationship.
Sam identity		Identify Sam’s key and contribute to agreement.
Sam signed prekey		Permit setup while Sam is offline, with a checked signature.
Nora ephemeral		Supply Nora’s fresh agreement contribution.
Sam one-time prekey		Supply an additional setup contribution when available.

Nora fetches Sam’s identity key, signed prekey and signature, plus a one-time prekey when one is available. She checks the signed prekey’s signature under the identity key she is using for Sam. If it fails, the exchange stops. She generates her ephemeral pair and calculates three DH outputs, adding a fourth when the optional one-time prekey is present. Using the notation from Table [16.1](#):

$$(16.4)$$

Each line states a matching pair of computations, not two operations that one participant must perform. Nora uses the expression on the left, with a private input she holds and a public value from Sam’s bundle. Sam uses the

expression on the right, with his corresponding private input and Nora's received public value. The DH correctness property makes their outputs agree for the prescribed inputs.

Z_4

is included only in the path with a one-time prekey.

The different pairs connect different roles.

Z_1

connects Nora's identity key with Sam's signed prekey.

Z_2

connects Nora's ephemeral key with Sam's identity key.

Z_3

combines the ephemeral and signed-prekey roles, and

Z_4

adds the one-time contribution. The construction combines them in a fixed order; it does not choose whichever output looks most random or publicly XOR arbitrary values until a key has the desired length.

For the path using all four outputs, the initial shared secret is

(16.5)

K_{init}

names the resulting secret, and

||

joins the output byte strings in the shown order.

KDF_x

denotes X3DH's specified HKDF construction, including its domain-separation prefix, salt and application information. For its X25519 setting, that prefix contains 32 bytes of hexadecimal `FF`; it is not a secret fifth DH output. Without a one-time prekey, the specified input omits

Z_4

. Neither case means applying a bare hash to a sentence containing the public keys.

Nora's initial message provides her identity and ephemeral public values, the required prekey identifiers and an authenticated ciphertext. Identity-key

encodings form part of the associated context. Sam uses the identifiers to load his private values, reconstructs the matching DH outputs and derivation, and checks the ciphertext. If the check fails, he must not accept the plaintext or retain an unauthenticated replacement session. Correct interpretation of the encodings and context is part of this agreement, not presentation detail.

Authentication still has a trust starting point. Suppose an attacker replaces Sam's entire bundle with an attacker-controlled identity key and prekeys carrying valid signatures under that substituted key. Verifying those signatures would establish internal consistency of the substituted bundle. It would not establish that the identity key belongs to the Sam whom Nora intended. A trusted comparison, an appropriate verified directory arrangement or another defined identity-verification mechanism must support that association.

An out-of-band comparison uses a suitably authenticated route separate from the untrusted key delivery. The participants might compare the intended public-key fingerprints or a specified encoded representation. Merely receiving a fingerprint beside an untrusted key repeats the problem from Chapter [14](#): an attacker can replace both. Identity changes also require an explicit policy. A fresh installation can explain a changed key, but an explanation is different from evidence that a particular new key is authorized.

Finally, an accepted initial message is not always a fresh request. Classical X3DH permits replay of an initial message on the path without a one-time prekey. Repeating the same accepted setup can reproduce the same initial secret. The following messaging protocol must therefore handle those conditions correctly, including introducing fresh recipient randomness before using a reply encryption key. Counting every received setup packet as a new human action would confuse a cryptographic exchange with application-level freshness. [\[94\]](#)

16.3 The symmetric-key and DH ratchets

A continuing conversation should not use

K_{init}

directly as one permanent message-encryption key. The **Double Ratchet** maintains separate sending and receiving chains and a root chain. A **chain key** is secret state from which the next message key and successor chain key are derived. A **message key** protects one message through the selected encryption construction. A **root key** is the evolving secret used when incorporating a new DH output to create a new chain. These names describe roles; they are not three different labels for the same stored value.

The **symmetric-key ratchet** advances a chain once per message. Write

C_j

for the current chain key and

K_j

for the message key at position

j

. One recommended instantiation uses HMAC with distinct constant-byte inputs:

(16.7)

The displayed 01 and 02 mean single bytes, not the two-character text strings. Both calculations use the same current chain key, but their different inputs separate the message-key and continuation roles. The next message repeats these steps with

C_{j+1}

. A suitable SHA-256 instantiation produces 32-byte outputs. The message key is not the input used to advance the chain.

For example, Nora starts a sending chain at position zero. The first step produces

K_0

and

C_1

. After applying the selected encryption operation, she removes the old

C_0

and the no-longer-needed message secret from the live protocol state. The second step uses

C_1

to produce

K_1

and

C_2

. Sam follows the matching receiving chain. His receiving state need not have advanced as far as Nora's sending state while messages remain in transit.

Under the construction's one-way and pseudorandomness assumptions, knowing a later chain key should not recover an earlier erased chain or message key. However, knowing

C_1

does allow computing

K_1

,

C_2

and the rest of that unchanged chain. This is the precise limit of advancing a deterministic secret state without adding unknown input. It protects one direction through time but does not make an exposed current chain become secret again merely because several messages have passed.

A reproducible chain calculation

The optional program below uses a public seed from a libsignal chain-derivation test and checks its published successor value. The message-key output here is the HMAC-derived value at the abstract chain interface; a concrete protocol may further expand or combine it before encryption. In particular, it is not the same object as every implementation's final field named cipher key. The example needs only Python's HMAC module.^[95]

Listing 16.1. Checking a public chain step and the limitation of a copied current chain key.

```
import hmac

def chain_step(chain):
    message_key = hmac.digest(chain, b"\x01", "sha256")
    next_chain = hmac.digest(chain, b"\x02", "sha256")
```

```

    return next_chain, message_key

seed = bytes.fromhex(
    "8ab72d6f4cc5ac0d387eaf463378ddb2"
    "8edd07385b1cb01250c715982e7ad48f")
chain1, key0 = chain_step(seed)
assert chain1.hex() == (
    "28e8f8fee54b801eef7c5cfb2f17f32c"
    "7b334485bbb70fac6ec10342a246d15d")

copied_chain = chain1
chain2, key1 = chain_step(chain1)
copy_next, copy_key = chain_step(copied_chain)
assert (copy_next, copy_key) == (chain2, key1)
assert key0 != key1
print("Published successor matched")
print("Copied current chain predicts the next step")

```

Matching the expected successor checks a specified input/output relation. The copied chain's matching result illustrates future exposure within that chain. The final inequality distinguishes two outputs in this example; it is not a proof that an attacker cannot recover old secrets. That argument depends on the cryptographic assumptions and proper state erasure, which this small Python session does not establish.

The **Diffie–Hellman ratchet** supplies the other mechanism. Each side maintains a current DH ratchet key pair and the peer's current ratchet public value. Receiving a new peer value triggers a root update using the existing local private value. The endpoint then generates a fresh local pair and performs another root update for its new sending chain. Its next outgoing message advertises the new local public value, allowing the peer to follow the corresponding calculation.

Let

R_t

be the root key before one such update, and let

Z_t

be the relevant DH output. A reduced description is

$$(16.8)$$

The pair on the left contains two different results: the next root key and the initial key for the new message chain. The root derivation

KDF_R

can use HKDF with the old root as salt, the DH output as input material and distinct application context. Deriving 64 bytes and splitting them into two 32-byte outputs is one appropriate interface for this role. This formula omits the protocol's state and encoding details; it does not authorize combining unrelated DH inputs under arbitrary labels.

An endpoint receiving a new peer ratchet value usually performs two root steps, with different purposes. The first establishes a receiving chain using its old local private value and the new peer public value. The second establishes a sending chain using a newly generated local private value and that peer public value. The peer reconstructs each step when it receives the relevant advertised public key. Sending several consecutive messages without a new peer ratchet value advances the symmetric chain without inventing a new DH exchange for every outgoing message.

Figure [16.1](#) begins with

R_0

initialized from the established session secret and Sam's initial ratchet public value

B_0

associated with that setup. Nora generates

a_1

for the ratchet; it is distinct from the setup ephemeral input

e_N

. In the specified PQXDH integration, Sam's signed prekey supplies the initial ratchet key role. Discarding its obsolete copy from one conversation's ratchet state does not establish that every private prekey copy has disappeared: another setup may still require permitted retained prekey material. The deletion labels in the figure refer to the ratchet state being replaced.

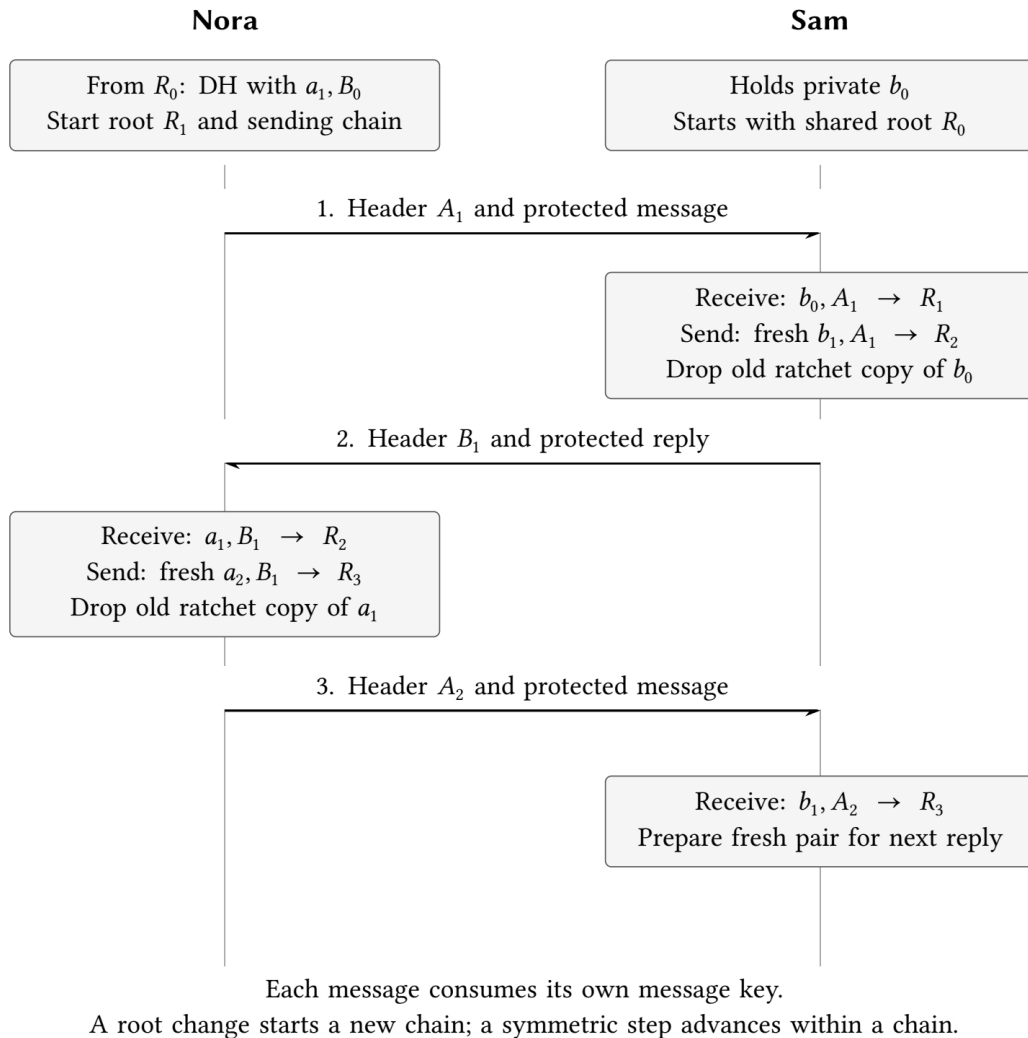


Figure 16.1. Three messages in a reduced classical Double Ratchet exchange.

and

are public ratchet keys;

and

are their private counterparts.

denotes successive root keys. Each root update also starts a directional chain, which supplies a separate message key. Old message and chain secrets are removed when no longer needed, apart from deliberately retained skipped message keys. The figure omits authentication setup, headers and skipped-key processing.

A message header identifies the sender's ratchet public key, its message number in the current sending chain and the previous sending chain's

length. The receiver can use those values to work out which chain and position apply. They require unambiguous encoding and authentication as part of the message context. The ratchet public key is not another encrypted copy of the private key, and merely changing an unauthenticated header must not permanently redirect the receiver's state.

Delayed messages and retained keys

Asynchronous delivery can put messages out of order. Suppose Nora sends positions 0, 1 and 2, but Sam receives position 2 first. His receiver advances the appropriate chain three times, retaining

K_0

and

K_1

as **skipped message keys** while using

K_2

for the arrived message. Later, position 0 can use the retained

K_0

without moving the chain backward. Position 1 can then use

K_1

. Successfully consumed keys are removed under the state rules.

The skipped-key lookup needs both the chain identity and message number. Position 0 in one DH chain is different from position 0 in another. The previous-chain-length field helps the receiver handle old-chain messages when a new peer ratchet value arrives. Without those distinctions, a delayed message could be confused with a current message at the same number. A bounded skipped-key store permits limited reordering without keeping every historical chain secret.

Retaining a skipped key changes what a later compromise can expose. An attacker who obtains that retained key may decrypt its matching recorded message, even though the main receiving chain has moved beyond it. Limits on the number and lifetime of stored skipped keys therefore balance delayed delivery, storage use and exposure. An unlimited message-number jump must not force unlimited derivation or storage. Old keys can expire, in which case the corresponding late messages may no longer be decryptable.

Authentication failure must also leave the established state intact. A receiver may prepare candidate changes while trying a message, but it must discard those changes if the authenticated decryption fails. Otherwise a forged header or damaged ciphertext could consume a skipped key, advance a chain or install a new ratchet value permanently. A replay within a session also needs rejection through the established state rules: a valid earlier ciphertext is not automatically a new message just because its tag is valid.

This is a stateful protocol. Correct encryption and decryption functions are not enough if a crash restores an old sending key, if two processes independently advance one state, or if a failed message commits half an update. Reliable state handling must preserve the intended one-time use and authentication decisions across those events. Describing the ratchet as changing keys is a helpful beginning; understanding what is stored, when it changes and which changes become trusted is the operational substance. [\[89\]](#)

16.4 Forward security and recovery after compromise

Forward security, in this discussion, concerns protection of earlier message keys against later exposure of specified current state. **Post-compromise security**, or PCS, concerns protection that can be regained for later communication after an earlier compromise, under the model's recovery conditions. Both statements need a time boundary and an account of what was exposed. Learning one message key, one chain key, the full ratchet state or an identity private key creates different capabilities.

If an attacker learns only one message key, the chain construction is designed not to reveal the other chain or message keys through that value. If the attacker learns the current sending chain key, the later positions in that chain become calculable. If the attacker learns the full current state, the root key and current DH private value may be available as well. A claim about one-message exposure must not be presented as protection against that larger state exposure.

Consider a temporary compromise of Nora's device after she sends the first message in Figure [16.1](#). Assume the attacker copies her relevant state, including

R_1

and

a_1

, then loses endpoint access and only observes the honest exchange. Sam's device has not been compromised. When Sam sends his fresh public value

B_1

, it is tempting to say that a new public key immediately heals the conversation. But the attacker still has

a_1

and can calculate

$\text{DH}(a_1, B_1)$

.

The attacker can therefore follow the next root step from the copied

R_1

to

R_2

and obtain the chain protecting Sam's second message. Freshness of

B_1

alone does not remove that capability, because one private input to the agreement is already exposed. This is why the recovery boundary cannot be inferred from the appearance of any new public value. We must follow the private inputs and root dependencies used for the actual message direction.

Nora then generates fresh

a_2

after the compromise has ended. For her third message, the root update includes

$\text{DH}(a_2, B_1)$

. Sam calculates the same value using

b_1

and

A_2

. In this scenario the attacker knows neither

a_2

nor

b_1

. Under the agreement and derivation assumptions, the recorded public values do not provide that new secret. The new sending chain can therefore regain protection from the earlier copied state. This explanation assumes

honest processing and the stated limited attacker; it is not a rule that every compromise heals after exactly three messages.

Following the exposed and fresh inputs

The next optional program continues the earlier session, reusing `chain_step`. It needs the `cryptography` package for X25519, HKDF and AES-GCM. The first root update is treated as already completed. The copied variables represent the attacker's specified exposure. The program checks agreement on the new chains and shows that the earlier chain's key does not open a message under the fresh chain. It is a small calculation of the scenario, with its own context string and encryption choice, rather than a Signal message-format implementation.

Listing 16.2. A copied old private input still follows one step; fresh uncompromised agreement inputs change the next chain.

```
import os
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x25519 import (
    X25519PrivateKey,
)
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

def root_step(root, private, peer_public):
    shared = private.exchange(peer_public)
    output = HKDF(
        algorithm=hashes.SHA256(), length=64,
        salt=root, info=b"book-ratchet-example-v1",
    ).derive(shared)
    return output[:32], output[32:]

r1 = os.urandom(32)
a1 = X25519PrivateKey.generate()
copied_r1, copied_a1 = r1, a1
b1 = X25519PrivateKey.generate()

r2, sam_chain = root_step(r1, b1, a1.public_key())
read_r2, nora_receive = root_step(r1, a1, b1.public_key())
known_r2, known_chain = root_step(
    copied_r1, copied_a1, b1.public_key())
assert (r2, sam_chain) == (read_r2, nora_receive)
assert (r2, sam_chain) == (known_r2, known_chain)

a2 = X25519PrivateKey.generate()
r3, nora_chain = root_step(r2, a2, b1.public_key())
read_r3, sam_receive = root_step(r2, b1, a2.public_key())
assert (r3, nora_chain) == (read_r3, sam_receive)
_, new_key = chain_step(nora_chain)
```



```

_, earlier_key = chain_step(known_chain)
nonce = bytes(12) # One encryption under this new message key.
context = b"book-example|Nora-to-Sam|chain-3|message-0"
sealed = AESGCM(new_key).encrypt(nonce, b"Third message", context)
assert AESGCM(chain_step(sam_receive)[1]).decrypt(
    nonce, sealed, context) == b"Third message"
try:
    AESGCM(earlier_key).decrypt(nonce, sealed, context)
except InvalidTag:
    print("Earlier chain key does not open the fresh-chain message")
else:
    raise AssertionError("unexpected acceptance")
print("Both endpoints derive the same new root and chain")

```

The fixed zero nonce is used for just one encryption under this newly derived example message key. It illustrates the one-message-key interface, not permission to reuse a nonce with the same AES-GCM key. The chosen AES-GCM record and context are teaching choices. The Double Ratchet specification defines an abstract encryption interface and discusses nonce handling and more misuse-resistant instantiations; this listing should not be mistaken for its recommended concrete packet format.

The equality checks establish correctness for the generated values. Rejecting the earlier key shows that one specific wrong candidate fails; it does not experimentally prove secrecy against all computations available to an adversary. The explanation of recovery still depends on the hardness assumption, the KDF construction, the exposure model and correct deletion. Python variables and their copies in this demonstration do not establish secure erasure.

PCS cannot promise recovery while an attacker continues to read every new private key from the endpoint. A malicious program that remains installed can obtain

a_2

as it is generated or simply read the displayed plaintext. Nor does the passive scenario cover an attacker continuously substituting ratchet values to maintain separate conversations. Restoring a trustworthy endpoint, obtaining appropriate fresh inputs and allowing the required honest communication are part of the recovery conditions.

Identity-key exposure creates an additional problem. Even if one existing conversation advances beyond an exposed ratchet state, a copied identity credential may permit impersonation in a new setup. A policy for replacing credentials and re-establishing trusted identity associations is therefore

separate from the mathematical evolution of an existing chain. The recovery mechanism does not remotely retract a stolen key or erase plaintext already copied by an attacker.

The research literature formalizes these distinctions. Cohn-Gordon, Cremers and Garratt’s 2016 work on post-compromise security separates threat models and studies how later honest exchanges can restore particular guarantees. The 2017 analysis by Cohn-Gordon, Cremers, Dowling, Garratt and Stebila examines a historical abstraction of Signal’s initial agreement and ratcheting as multi-stage key exchange. Its result concerns that model, with explicitly stated limitations, rather than every later protocol revision or application using the Signal name. [\[93\]](#)[\[92\]](#)

For example, that historical Signal analysis assumes honest distribution of the relevant identity and prekey information within its model. It does not analyze the complete out-of-band verification channel, retained skipped-message-key storage, all multi-device behavior or implementation side channels. Those omissions do not make the modeled result useless; they tell us where it applies. A well-supported security explanation keeps the theorem, the protocol specification and the software deployment as connected but distinct evidence.

16.5 PQXDH and post-quantum ratchets

A quantum threat introduces a new time question. An attacker could record today’s encrypted exchanges and later obtain a computer capable of defeating the conventional public-key agreement used in them. This is often called a **harvest-now, decrypt-later** attack. It does not require the attacker to decrypt the ciphertext at collection time. The value of a long-lived private conversation can make protection against later analysis relevant before such a computer is available.

PQXDH, Post-Quantum Extended Diffie–Hellman, adds a post-quantum key-encapsulation contribution to the initial setup. We use Revision 3, last updated in January 2024, as identified by the published specification. Its security goals and authentication limitations must be read for that revision. The public document still gives CRYSTALS-Kyber parameter examples;

those examples should not be silently renamed to every later ML-KEM encoding or described as the parameters of every current application. [\[90\]](#)

Let
 P_Q

denote the selected recipient KEM public key. The sender performs

$$(16.9)$$

C_Q

is a KEM ciphertext sent to the recipient, while

Z_Q

is the encapsulated shared secret. The recipient uses the corresponding private decapsulation key and

C_Q

to recover the matching

Z_Q

. These are the KEM roles introduced in Chapter [10](#): the ciphertext is not the message body, and the shared secret is not transmitted as plaintext.

For the PQXDH path also using the optional curve one-time prekey, the combined input is represented as

$$(16.10)$$

The first four values are the prescribed curve agreement outputs, and

Z_Q

is the additional KEM secret.

KDF_P

denotes PQXDH's specified HKDF derivation, with its prefix and context identifying the application, curve, hash and KEM parameters. It is not automatically identical to

KDF_X

in the classical example. On the path without a curve one-time prekey,

Z_4

is omitted under the specification.

Sam publishes signed post-quantum prekeys as well as the required curve information. Nora checks their signatures and includes the chosen KEM key identifier and encapsulation ciphertext in the initial message. A signed one-time KEM prekey is preferred when available; a signed last-resort KEM prekey provides the defined fallback. The retained lifetime of a last-resort private value changes the compromise conditions. Where a KEM does not itself bind its public key into the ciphertext as required by the construction, the specified associated-data binding must also be included.

The new secret can protect the initial agreement against passive later quantum analysis under the selected KEM and other construction assumptions. It does not make every form of authentication quantum-secure. The cited PQXDH revision still relies on conventional discrete-logarithm-based identity authentication. An active attacker with the relevant quantum capability is outside that authentication guarantee. Combining a post-quantum secret with conventional identity keys must therefore be described by its actual goals, rather than shortened to the claim that every part is quantum-proof.

Initial protection and ongoing recovery are also different tasks. Suppose a quantum-resistant initial secret is later copied from an endpoint along with the conversation state. A classical DH ratchet can add unknown inputs against the conventional passive attacker described earlier. But an adversary who stores subsequent public ratchet keys and later solves those DH problems could follow the recorded updates from that copied state. A post-quantum initial setup alone does not establish post-quantum recovery after a later state compromise.

The current Double Ratchet specification therefore also describes a **Sparse Post-Quantum Ratchet**. Its underlying **Sparse Continuous Key Agreement**, or SCKA, exchanges public protocol data over successive messages and emits new shared secrets when enough information has been exchanged. Sparse means that an independent new agreement secret need not appear with every application message. Ordinary message keys can still advance while the next larger public-key exchange is in progress.

Each new shared secret has an **epoch identifier**, an ordered label for the agreement generation. The surrounding ratchet uses the epoch information

to select the appropriate sending and receiving chains and mix new outputs into root state. An epoch is not a timestamp, and a new application message does not necessarily create a new epoch. Correct association between messages, epochs and chains is necessary for agreement and deletion, especially when messages are lost or delivered out of order.

ML-KEM Braid is one specified SCKA construction. It uses ML-KEM's structure and an incremental interface to overlap parts of the exchange, rather than sending a complete large object and waiting at every step. Public-key and ciphertext components travel in encoded chunks. Erasure coding helps the recipient reconstruct those objects despite missing chunks. The word erasure here describes missing transmitted pieces; it is unrelated to securely erasing a private key from storage. [\[91\]](#)

The larger exchange creates a practical recovery tradeoff. More bytes carried in each message can help the agreement advance sooner, but uses more bandwidth. If one side never replies, the required exchange may never finish. Dropped messages can delay a sparse ratchet's progress even when application messages remain decryptable through established chains. The vulnerable message set after a compromise therefore depends on traffic behavior and delivery, not just the name of the KEM or a fixed number of elapsed seconds.

The **Triple Ratchet** combines the classical Double Ratchet and a Sparse Post-Quantum Ratchet. Each supplies a message-key contribution. A specified hybrid KDF combines the two into the encryption key rather than encrypting the message twice and assuming that duplication proves security. The intended combined protection depends on that composition's assumptions, context and state handling. The word triple does not mean three users, three identical ciphers or a universal multiplication of security strength.

These are related specifications at different layers: PQXDH starts an asynchronous session, SCKA emits continuing agreement secrets, and a ratchet uses state and message-key derivation around those outputs. Revision 4 of the Double Ratchet specification, dated November 2025, describes the classical, sparse and combined variants. ML-KEM Braid's published account also identifies a symbolic ProVerif analysis of its modeled properties. Such a model treats cryptographic operations through

specified abstractions; it is not a direct audit of every device implementation or a new proof of ML-KEM's underlying mathematical assumption. Later chapters develop post-quantum mechanisms and verification methods more fully. [\[89\]\[91\]](#)

16.6 Endpoints, backups and metadata

The best place to understand an encrypted conversation's boundary is the point where a recipient reads it. Sam's device must obtain enough secret material to display the message. A malicious keyboard, compromised application, screen-capture tool or person with access to the unlocked conversation can obtain plaintext without solving X25519 or reversing a hash. The ratchet protects keys under an exposure model; it does not make the act of displaying a message invisible to the endpoint that displays it.

Device security and cryptographic protocol security therefore support one another. A protected key store can reduce extraction of some credentials, while process isolation and appropriate permissions can restrict other access. However, a component authorized to receive plaintext can misuse that authorization. Claims about non-exportable identity keys do not automatically describe the protection of ratchet state, message caches or the interface returning decrypted content. Those objects need their own boundaries and evidence.

Secure deletion means removing secret material in a way that meets the intended recovery-resistance assumptions, not simply removing its name from a program. Ratchet arguments often rely on old message, chain and private agreement values becoming unavailable after their permitted use. Copies in memory, storage journals, crash reports or snapshots can defeat that assumption. Chapter [14](#) explains why deleting one variable does not demonstrate destruction of every usable copy.

An archived plaintext history is a separate exposure path. The ratchet can erase the key that originally protected a message while an endpoint retains the readable message in its local database. A later compromise that reads that history does not need to recover the erased message key. This is not a contradiction in the mathematics: the historical-key protection and the local-history policy concern different stored objects.

Backups must be assessed in the same way. A backup may contain plaintext history, encrypted history, recovery secrets, ratchet state or a combination. Saying that a backup is encrypted leaves open who controls the recovery key, what authenticates restoration and whether a weak password permits offline guessing. An independently protected backup can be useful, but its security is a separate construction. It does not inherit all properties of a ratchet solely because the original messages came from an encrypted conversation.

Restoring old live state can also repeat secrets. If a sender resumes from a snapshot preceding a message that was already sent, it can derive that message's key again. Under a construction that assumes one encryption per message key, this can violate the intended use. Reinstalling an application, moving devices or restoring a backup therefore requires a defined state and identity transition. It should not be treated as copying a normal document and continuing every old counter unchanged.

Multiple devices enlarge the set of authorized endpoints. If Nora's phone and laptop both receive copies, each needs an authenticated participation and session arrangement. Removing the laptop should change future distribution, but cannot retract a plaintext copy it already received. A newly linked device must not acquire authority solely because the delivery service supplied an unverified new key. The two-party ratchet explains each protected pair's evolving state; the device-directory and membership policy decides which pairs should exist.

Metadata is information about the communication rather than only its message body. Delivery times, approximate message sizes, recipient routing and network addresses can reveal relationships or activity. The base Double Ratchet header includes public ratchet information and counters; it is authenticated but not necessarily hidden. Its header-encryption variant changes that part of the exposure, with additional key state and processing. It does not automatically hide the transport route, all server observations or the users' behavior.

Message padding, traffic-shaping or specialized delivery designs can address selected observations, but their scope must be stated. A confidentiality claim about message contents is different from anonymity or a guarantee that no observer can link two participants. Likewise, an

encrypted attachment may leave file size or access timing visible through another service. The relevant question is what each actor can observe across the actual system, not whether every product screen displays a lock symbol.

An authenticated message also has limited evidential meaning. Shared-key message authentication and the deniability goals of particular setup protocols do not automatically create a public signature on every chat message. **Deniability**, in this cryptographic setting, concerns what protocol evidence can establish to an outside party under a defined model. It does not guarantee that screenshots, a cooperating participant, an actively involved observer or an exposed endpoint can supply no evidence of a conversation. Legal conclusions do not follow from a protocol name.

Finally, the handling of a key change or authentication failure affects the reader's real protection. An application that silently accepts a substituted identity, resends sensitive content to a newly unverified key or exposes decrypted bytes after a failed check changes the security story. Conversely, a failure can prevent unsafe acceptance without ensuring that the message is eventually delivered. Content confidentiality, identity trust, availability and a useful recovery process remain distinct goals.

The lesson of changing keys is precise rather than absolute. Prekeys make an offline start possible; identity checks establish whom the setup is intended to reach; symmetric chains limit the effect of later state exposure on earlier erased keys; and fresh agreement inputs can permit conditional recovery for later messages. Post-quantum additions address another class of adversary and require their own versioned claims. All these mechanisms depend on the surrounding state, endpoints and retained copies. The next chapter examines how good cryptographic ideas fail when those conditions are broken.

Questions to check your understanding

1. Why does TLS to a delivery server not by itself establish end-to-end message secrecy? What work do public prekeys permit while the recipient is offline?
2. In the classical setup, distinguish an identity key, a signed prekey, a one-time prekey and an ephemeral key. Why are public-pool removal and private-key deletion different events?
3. Explain the paired computations in Equation [16.4](#). Who holds each private input, and when is Z_4 included?
4. Why does a valid signature under an unverified substituted identity key not establish the intended person's identity? What replay issue can occur without a one-time prekey?
5. In Equation [16.7](#), identify C_j , K_j , 01 and 02 . Why can a copied current chain key predict later keys but should not reveal earlier erased keys?
6. What are the separate jobs of the root, sending and receiving chains? Why does receiving a new peer DH public value normally lead to two root updates?
7. If positions 0, 1 and 2 arrive in the order 2, 0, 1, what must the receiver retain? What must happen to candidate state changes after failed authentication?
8. In the temporary-compromise scenario, why can the attacker follow the a_1, B_1 step but not obtain the next fresh DH output merely from A_2, B_1 ? Which assumptions make recovery conditional?

9. Distinguish PQXDH, SCKA, ML-KEM Braid and the Triple Ratchet. Why do a post-quantum initial secret and a new application message not automatically mean a fresh post-quantum recovery step?
10. Why can local history, skipped keys, backups, additional devices and metadata remain relevant even when message-key erasure and ciphertext checks work as intended?

17. How Good Cryptography Fails

Part V: Building Secure Systems

Nora's service uses recognized algorithms. Its encrypted files open correctly, its signatures verify, and its secure connections complete. Yet an attacker may still find a useful path through repeated secret values, an error response, the time taken by a calculation or a mistaken interpretation of authenticated bytes. These failures do not all mean that the mathematics of the underlying cipher has been defeated. They can mean that the implemented system gives the attacker information or capabilities that the security argument did not allow.

A cryptographic construction has conditions on its inputs, state and use. An implementation must preserve those conditions while dealing with malformed requests, concurrent operations, storage failures and the behavior of its processor. The surrounding application must then act on the result correctly. These are connected layers of one protection goal. A correct equation can coexist with an unsafe error path, just as a correct signature check can precede a wrong authorization decision.

We will examine how those gaps arise and how to describe them precisely. Three historical cases provide anchors: Bleichenbacher's 1998 RSA padding-oracle attack, Brumley and Boneh's 2003 remote timing work, and the KyberSlash implementation vulnerabilities analyzed in 2025. Each case has a particular attacker, interface and affected setting. The examples explain mechanisms rather than asserting that every current library remains vulnerable. One optional program uses a deliberately tiny local model to make information leakage visible without contacting a real service.

17.1 Secret reuse and poor randomness

An input can be mathematically well formed while being unsuitable for its security role. A predictable AES key is still a valid-length byte string. A repeated nonce can still pass an interface's length check. A signature

produced with an exposed per-message scalar can still verify. Correctness confirms the required calculation for those inputs; it does not establish that the inputs satisfy the distribution, secrecy or uniqueness conditions supporting security.

Chapters [7](#) and [13](#) showed two different reuse failures. Reusing stream-encryption material can expose a relationship between plaintexts. Reusing a secret signing scalar can create equations that reveal a longer-lived private key. Their details differ, so a rule such as never repeat any public number would miss the point. We must identify the value's role: some inputs require uniqueness, some unpredictability, some confidentiality, and some a specified combination.

Random choice does not guarantee uniqueness. If a construction uses independently uniform

b

-bit nonces, a small-collision-probability estimate after

q

encryptions under one key is

$$(17.1)$$

Here

b

is the number of bits in each nonce,

q

counts the nonces selected under that key and

p_{coll}

estimates the probability that at least two match. The approximation adds the matching probability over the

$q(q-1)/2$

pairs. It is useful when collisions are still unlikely, with independent uniform choices; it is not an exact formula for arbitrary inputs or large

probabilities. Chapter [5](#) develops the underlying collision reasoning.

For example, with 96-bit nonces and one million independently uniform selections, the estimate is about

6.3×10^{-18}

. This is a calculation under the stated model, not evidence that a particular device generates such values, or a complete security limit for an AEAD construction. Biased generation, repeated generator state or a deliberate fixed value invalidates that model. A correctly maintained counter can avoid accidental repetition within its assigned range, but it has different operational requirements from independent random selection.

Counter allocation becomes difficult when several processes share a key. Two machines can each start at zero and locally obey an instruction to increment after every use, while globally repeating the same key/nonce pairs. A crash can restore a previously used counter; a cloned machine image can duplicate both its key and allocation state. The protection therefore needs a defined scope for uniqueness, including concurrent writers, restarts and restored snapshots. Checking one process's source code for an increment is not sufficient evidence.

Random generation has related state dependencies. A cryptographic generator expands protected state into outputs according to its design. Copying its state can copy the basis for future outputs. Replacing failed generation with a timestamp, an empty value or a fixed demonstration key may keep an application running while removing the intended unpredictability. The failure should be handled through the selected system's documented behavior rather than silently changing the generation rule.

Passwords create another common misunderstanding. Hashing a weak password into a 32-byte result does not give it the uncertainty of a uniformly selected 256-bit key. An attacker can guess the password and repeat the public derivation. A suitable password-based KDF can raise the cost of each guess, but does not turn a short predictable password into an independently random secret. This is why Chapter [9](#) separated password processing from HKDF expansion.

Finally, deterministic generation is not necessarily a weakness. A specified deterministic signature procedure derives its per-message secret

from protected key material and the message under defined rules. That is different from substituting one fixed scalar for every signature. Similarly, a ratchet deliberately derives successor state, but exposure of its current state has the consequences explained in Chapter [16](#). The useful question is which prediction or repetition the design permits, not whether the word random appears in every step.

17.2 Padding oracles and observable errors

A receiver often has to reject malformed data. An **oracle**, in cryptanalysis, is an interface that supplies some useful answer about an input, even when it does not return a secret directly. A **padding oracle** reveals whether decrypted data satisfy a particular formatting or padding condition. The answer might be an explicit error message, a connection difference or an observable timing pattern. An attacker can use repeated answers to learn more than the designer expected each individual error to disclose.

An **adaptive chosen-ciphertext attack** chooses later ciphertext queries using information from earlier responses. The word adaptive is important: a successful response can narrow the next question. The attacker does not need permission to ask for the target plaintext in an ordinary successful transaction. Instead, related ciphertexts and the receiver's behavior supply constraints on that plaintext.

Case study: a small answer about RSA formatting

Bleichenbacher's 1998 paper studies RSA encryption with the historical PKCS #1 v1.5 encryption format and an interface revealing whether a decrypted block conforms. That format begins with bytes 00 02, followed by at least eight nonzero padding bytes, a zero separator and the message. Acceptance reveals a fact about the private decryption result. The attack concerns that formatting interface and RSA's multiplicative structure; it is not a general factoring algorithm or an attack on every RSA encryption scheme. [\[96\]](#)

Let

n

be the RSA modulus,

e

its public exponent and

d

its private exponent. Suppose the target ciphertext is

$c \in \mathbb{N}^*$ and n

, where

m

represents the encoded block as an integer. For a selected positive multiplier

s

, an attacker can form

$$(17.2)$$

The first operation needs only public information. The second equality explains the private computation's result using RSA correctness and multiplication, as developed in Chapter [11](#). It does not say that the attacker already knows

m

. It says that a response about the new decryption constrains a known multiple of the original unknown value.

If the encoded RSA block has

k

bytes, define

$B = 256^{k-1}$

. A conforming block whose first two bytes are 00 02 must satisfy

$$(17.3)$$

for the integer

r

accounting for reduction modulo

n

. The quantity

$$ms - rn$$

is the remainder

$$ms \bmod n$$

. This interval condition follows from acceptance, but does not by itself establish full format validity: the nonzero padding, separator and remaining checks still matter. Treating the two leading bytes as a complete padding check would describe a different oracle.

For a successful selected query, the same inequality gives

$$(17.4)$$

The ceiling brackets mean round upward to the next permitted integer; the floor brackets mean round downward. We add

$$rn$$

to the endpoints, divide by the known positive

s

, and retain only whole-number candidates. The attacker must also account for all possible

r

values and intersect the resulting intervals with earlier information. A successful response does not announce

r

or immediately provide the complete message.

Use scaled numbers to see the reasoning without a long encoded block. Suppose the current candidate range is 200 through 299, the modulus is 1147 and the selected multiplier is 10. These are teaching values with a decimal interval, not PKCS #1 padding parameters. If the oracle accepts the multiplied value in that same interval, the possible modular quotient in this example is

$$r = 2$$

. The updated bounds are

and

$$\lfloor (299 + 2 \times 1147) / 10 \rfloor \\ = 259$$

One accepted query has narrowed 100 candidates to ten.

For the particular hidden value 250, multiplication gives

$$250 \times 10 = 2500$$

, and subtracting

$$2 \times 1147$$

leaves 206, which the model accepts. The attacker-facing answer is only acceptance; the explanatory calculation here uses the known teaching value so that we can check the result. The following optional program reproduces this local example with disclosed tiny RSA keys. It is an interval-oracle model, not a complete Bleichenbacher implementation or a valid PKCS #1 encoder.

Listing 17.1. A local decimal-interval oracle narrows a small RSA teaching example. No network service is contacted.

```
p, q, e = 31, 37, 7
n = p * q
d = pow(e, -1, (p - 1) * (q - 1))
hidden_message = 250 # Public teaching value, not a real secret.
target = pow(hidden_message, e, n)

def interval_oracle(ciphertext):
    value = pow(ciphertext, d, n)
    return 200 <= value <= 299

s = 10
query = target * pow(s, e, n) % n
assert interval_oracle(query)
remaining = []
```

```

    guess for guess in range(200, 300)
    if 200 <= guess * s % n <= 299
]
assert target == 872
assert query == 173
assert remaining == list(range(250, 260))
print("Accepted related query")
print("Remaining candidates:", remaining)

```

The predicate returns one bit of information for each query, but the information is not independent of the chosen multiplier and previous state. Successful adaptive queries can progressively isolate an unknown value. Real query counts depend on the exact acceptance conditions and search strategy. Bleichenbacher's reported experiments used a constructed oracle, with different keys and many queries; those figures should not be converted into a universal cost for every historical or current server.

The same principle with a different encoding

CBC padding oracles illustrate a related mechanism with different mathematics. In cipher block chaining, a decrypted block is combined with the preceding ciphertext block:

(17.5)

P_i

is plaintext block

$\cdot$
 i

,

C_i

its ciphertext block,

K

the block-cipher key and

D_K

the block decryption function. For the first block, the IV takes the preceding-block role. Changing a byte of

C_{i-1}

changes the corresponding byte of

P_i

through XOR without requiring knowledge of

K

. This relation is not the RSA multiplication in Equation [17.2](#).

For a simple PKCS #7-style tail check, a final plaintext byte of 01 can indicate one padding byte. Suppose the unknown last byte of

$D_K(C_i)$

is 6B. A chosen preceding byte 6A makes the resulting last plaintext byte 01, because $6B \text{ XOR } 6A = 01$. An observed successful one-byte padding condition can therefore identify that intermediate byte as $6A \text{ XOR } 01 = 6B$. If the original preceding byte was 2A, the original plaintext byte was $6B \text{ XOR } 2A = 41$, the ASCII byte for A.

This small relation explains the leak; a complete attack must distinguish other valid padding lengths, handle the whole block and match the actual protocol's rules. In particular, historical TLS CBC padding uses its own length-byte convention rather than this PKCS #7 example. Lucky Thirteen, published by AlFardan and Paterson in 2013, showed how timing differences could survive attempted uniform error handling in the older TLS/DTLS CBC processing. Its treatment of padding and MAC work must not be confused with TLS 1.3's AEAD record construction. [\[99\]](#)

Replacing two error strings with one does not establish that an oracle has disappeared. Response lengths, subsequent work, connection behavior and timings may still distinguish conditions. Appropriate constructions and carefully implemented failure handling aim to avoid exposing those distinctions. AEAD interfaces must withhold unauthenticated plaintext from the application, and older modes need their own correct composition and processing. A label saying decryption failed is not a security argument about everything observable before and after that label.

17.3 Time, caches and secret-dependent work

A **side channel** reveals information through behavior beyond the intended cryptographic outputs. Execution time, memory-access patterns, power consumption and electromagnetic effects are possible examples.

They require different observation capabilities and defenses. A network timing attacker need not have a power probe; a local cache observer need not see a useful network error. A precise account states which channel is visible and which secret-dependent behavior contributes to it.

Kocher's 1996 work explained how private-key calculations could leak secrets through their execution times. A program might conditionally multiply according to an exponent bit, take different reduction paths or use instructions whose time depends on operands. The attack does not require a mathematical error in the final result. An implementation can return the correct value for every input while the route to that value reveals something about its private input. [\[97\]](#)

Case study: observations across a network

Brumley and Boneh's 2003 experiments extended the practical concern to general-purpose servers. They studied an OpenSSL RSA private-operation implementation and recovered a 1024-bit private key in a local-network experimental setting. The paper discusses OpenSSL 0.9.7 and comparable results with 0.9.6g; it also reports that enabling blinding by default in 0.9.7b defeated their demonstrated attack. These are historical versions and configurations, not a claim that all later OpenSSL releases expose the same behavior. [\[98\]](#)

Their analysis used details of optimized arithmetic, including CRT and multiplication/reduction behavior, rather than only a textbook branch on a key bit. The attacker sent chosen inputs, measured response behavior and used hypotheses about the computation to distinguish candidate secret information. The experimental network included ordinary communication equipment and noise. That does not mean every noisy internet path permits the same attack at the same cost; it establishes that network noise alone was not a sufficient defense in the tested setting.

An illustrative measurement model is

T_i

is measurement

i

,

T_0

a baseline,

b

a hidden condition represented by 0 or 1, and

δ

the timing contribution associated with that condition. The term

ϵ_i

represents noise. This is a simplified statistical model, not the exact Brumley–Boneh attack. It shows why a small secret-dependent difference can remain meaningful when repeated observations reduce uncertainty about an average.

If noise terms are independent with zero mean and standard deviation

σ

, the standard deviation of the average of

N

measurements is

$\sigma / \sqrt{N}$

. For the difference of two independent averages of that size and equal noise variance, it is

$\sigma \sqrt{2/N}$

. The square root means the nonnegative number whose square gives the quantity inside. Repetition does not make a systematic leak vanish; under these assumptions it can make that leak easier to distinguish.

For scale only, suppose

δ

is one microsecond and

σ

is 20 microseconds. With 20,000 observations in each group, the modeled standard deviation of their difference is

microseconds. The one-microsecond gap is five times that quantity. This calculation is not a measured attack success rate, a probability of key recovery or a universal sample requirement. Real observations can be dependent, biased or unstable, and actual attacks must justify their grouping and inference.

Constant-time programming aims to prevent relevant control flow, memory access and operation timing from depending on secret data under a stated leakage model. It does not require every request of every public length to take the same number of seconds. Public input lengths and environmental scheduling can affect elapsed time. The important requirement is that the permitted observations do not distinguish secret choices that the protection should hide.

A comparison loop that returns at the first unequal byte can reveal how long a prefix matched. A memory lookup indexed by a secret can reveal an accessed cache location even if the source has no visible conditional statement. A branch-free arithmetic expression can still compile to a variable-time instruction. These are different reasons that counting the number of lines or removing every `if` from source code is not a complete timing defense.

A documented comparison interface also has a scope. PyCA's `bytes_eq` compares equal-length byte strings without making the matched prefix the stopping condition, but its documented behavior returns immediately for different lengths. That is compatible with a model in which length is public; it is not a promise to conceal a secret length. Similarly, the collected OpenSSL blinding code separates preparation of a random invertible factor, conversion of an operand and removal of the factor afterward. Those paths help connect the mathematical construction to software responsibilities. Reading them does not establish the timing behavior of every compiled provider, platform or application caller. [\[102\]](#)

One countermeasure for RSA private operations is **blinding**, which randomizes the operand while preserving the recoverable result. Choose

fresh suitable

r

with an inverse modulo

n

, then compute

(17.9)

C

is the original ciphertext and

C_b

the blinded one. The private operation produces

u

, which corresponds to the original plaintext multiplied by

r

. Multiplication by the inverse

r^{-1}

removes that factor. The fresh unpredictable operand makes a chosen external ciphertext less directly related to the private computation's internal timing. Correct random generation, inverse handling and the rest of the implementation still matter; blinding is not certification that every side channel is absent.

Random delays address a different thing: they add variation without necessarily removing secret-dependent behavior. If the added noise can be averaged away, the underlying signal remains. Rate controls can raise the cost of repeated observations, but do not establish that the computation is secret-independent. Effective defenses need to match the leakage mechanism and threat model, rather than rely only on making measurements inconvenient.

17.4 Encoding and protocol confusion

Cryptography protects encoded inputs. The receiver must interpret those inputs with the same boundaries and meaning that the sender intended. A hash, MAC or signature can be calculated perfectly over bytes that the application then parses ambiguously. This is a failure of the protected representation or its use, not necessarily a collision in the hash function or a forged signature.

Suppose a record contains two fields. Simply joining *ab* with *c* gives the same bytes as joining *a* with *bc*. A signature over the result *abc* cannot establish which pair was intended when no other framing rule distinguishes them. Both interpretations use exactly the same signed input. No attacker has found two different inputs with the same digest in this example.

A defined length encoding can separate those pairs. With a one-byte length before each field, the first becomes 02 61 62 01 63, while the second becomes 01 61 02 62 63. Here 61, 62 and 63 are the ASCII bytes for *a*, *b* and *c*. This is a small framing illustration, not a recommended universal protocol. Real formats must specify length ranges, byte order, permitted fields and rejection of malformed or conflicting structures.

Canonical encoding selects an agreed representation where the construction requires one. The application must know whether signatures cover raw received bytes, a specified canonical form or another structured representation. A verifier and a business-logic parser that disagree about duplicate fields, integer interpretation or character normalization can attach different meanings to the same accepted object. Arbitrarily normalizing data after a cryptographic check does not solve that disagreement.

Domain separation distinguishes uses of a cryptographic operation through specified context or labels. For example, a payment authorization and a diagnostic acknowledgment should not become interchangeable merely because both contain a number and verify under one key. Binding an operation type, protocol version, participant roles and relevant identifiers into the authenticated construction can restrict such confusion. Those fields must themselves have exact encodings and be enforced by the receiver; a context string that is ignored after verification supplies little application policy.

Algorithm identifiers are also part of the security decision. A message should not be allowed to choose an arbitrary interpretation of the

verification key or a weaker processing route outside the protocol's rules. A public RSA verification key is not a shared MAC secret just because an interface accepts byte strings for both. Similarly, accepting a message under one role does not authorize treating it as a message from the opposite role. The TLS CertificateVerify context and directional traffic secrets in Chapter [15](#) illustrate deliberate separation.

A **downgrade** causes parties to use a weaker version or parameter choice than the intended negotiation should allow. Listing several algorithms is not itself a defense if an attacker can alter the selection without an enforced binding and policy. Some compatibility fields have historical values even in newer protocols, so a defender must follow the specified negotiation rather than guess a version from one familiar header byte. Rejecting an unsupported choice is a meaningful security decision, even when permissive acceptance would appear more convenient.

Replay and rollback are further context questions. A signed older program can still verify after the publisher has released a corrected version. A MAC-valid payment request can still be a duplicate. A server certificate can have a valid path while representing a different service name. Version state, reference identity, authorization and freshness must be checked at their own boundaries. It is misleading to call all these results cryptographic forgeries when the attacker may be reusing completely authentic bytes.

These examples explain why protocol descriptions include encoding and processing rules in addition to algorithms. The rules determine which mathematical input corresponds to which application statement. A secure composition needs compatible interpretations from the key-selection step through authentication and use. Treating serialization, parsing or application context as unrelated details can leave a mathematically protected object with the wrong practical meaning.

17.5 Compromised endpoints and exposed state

An attacker who reads plaintext from the endpoint need not attack its transport encryption. A service can terminate TLS correctly and then write a password into an accessible log. A messaging device can perform the ratchet correctly while malicious software captures the displayed conversation. The missing protection is at another boundary. Increasing a

cipher's key length cannot repair an authorized plaintext output that is exposed to an unauthorized observer.

Key extraction and operation misuse remain different capabilities. A hardware module may prevent exporting raw private bytes while allowing a compromised caller to request signatures on unauthorized objects. A recovery service may protect its master key while an attacker controls an account permitted to ask for decryption. The surrounding authorization and message meaning determine whether the cryptographic operation serves the intended person or the attacker.

Partial state exposure also needs a precise account. Learning one message key can expose its message; learning a current chain key can expose the remainder of that chain; learning a root and current private ratchet input can expose additional updates. Chapter [16](#) worked through those differences. Describing a system as recovering after compromise without naming the compromised material, continuing attacker access and necessary fresh inputs makes the claim too vague to assess.

The same issue appears in debug facilities and backups. A key log intended to diagnose a connection can decrypt matching captured traffic if it records sufficient secrets. A crash report can retain private state that ordinary operation was meant to erase. A backup can preserve readable history or restore an already-used key and counter. These objects are part of the exposure analysis even if they do not appear in the central encryption diagram.

Faults can affect integrity as well as secrecy. A corrupted private operation might produce an incorrect value; whether that becomes exploitable depends on the construction, outputs and attacker capabilities. Detection and error handling must ensure that an invalid result is not released as if it were an ordinary valid one. We do not infer a fault attack from every failed calculation, but we also should not assume that all implementation failures merely cause harmless unreadable data.

Unsafe memory handling can undermine the isolation assumed by an algorithm. Out-of-bounds access, a stale pointer or another implementation defect can expose keys or change a security decision without defeating the mathematical primitive. Memory-safety techniques reduce particular classes of such failures, while functional correctness and side-channel resistance

address different questions. None of these properties automatically implies all the others.

Likewise, a trusted cryptographic library cannot determine whether every application caller has selected the right key, context and permissions. Its interfaces can reject many malformed inputs or enforce useful restrictions, but the caller still supplies application meaning. A successful operation on attacker-selected metadata may be a completely correct library result inside an incorrect authorization flow. This is one reason to trace the whole path rather than stop when a maintained library appears in the dependency list.

Responding to exposure therefore begins with capabilities and dependencies. Determine what secret material, plaintext, credentials or operation authority could have been obtained, then restore a trustworthy execution and access path before relying on replacements. Rewrapping an exposed data key leaves that data key unchanged, as Chapter [14](#) explained. Updating a derived secret deterministically does not introduce independent unknown material when its predecessor is known.

Recovery also has limits that should remain visible to the reader. New protection can secure later material under appropriate conditions. It cannot erase plaintext already copied by an attacker or prove that no earlier disclosure occurred. An honest report separates what was corrected, what the available evidence establishes and what remains uncertain. That precision is more useful than declaring a system safe merely because its latest encryption test passes.

17.6 Lessons from corrected implementations

A correction must remove the relevant failure mechanism while preserving the intended function. It can require a new construction, a stricter state transition, changed error handling, secret-independent arithmetic or a corrected application policy. Each response should be assessed against the capability the attacker used. Changing an unrelated visual output or replacing a key without addressing the leak can leave the same attack path available.

Case study: KyberSlash and variable-time division

KyberSlash provides a concrete post-quantum implementation lesson. Bernstein and colleagues’ 2025 paper analyzes two timing vulnerabilities in several Kyber implementations, including the November 2023 reference version. The attacked function is decapsulation under a target secret key, with chosen ciphertexts and observable timing. The paper demonstrates particular attacks on Arm platforms and identifies corrections. This is not a mathematical solution of the lattice problem underlying the KEM, nor a statement that every ML-KEM implementation is vulnerable. [\[100\]](#)

One affected message-decoding step maps a canonical coefficient

$$t$$

, between 0 and

$$q - 1$$

, into a bit, with Kyber’s coefficient modulus

$$q = 3329$$

. A simplified arithmetic presentation of that step is

$$(17.10)$$

Here

$$t$$

is the coefficient being decoded, 1664 is the integer half of 3329 rounded down, and

$$b(t)$$

is the resulting bit. First double

$$t$$

, then add 1664, divide by 3329 and discard the fractional part. Finally keep the remainder on division by 2. This is one reduced decoding operation, not the complete KEM or an implementation recommendation.

At

$$t = 832$$

, the numerator is

$t = 100 \dots 800 \dots 600$
, so the quotient rounded down is 0 and the output bit is 0. At

$t = 833$
, the numerator is 3330, giving quotient 1 and bit 1. At

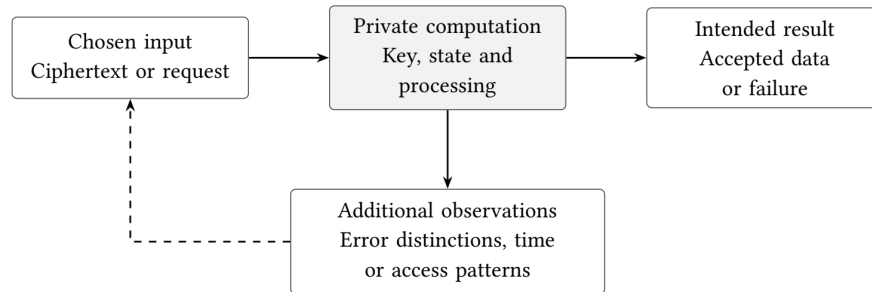
$t = 2497$
, the numerator is 6658, giving quotient 2 and bit 0 after reduction modulo 2. The arithmetic can be correct in every case while the executed division takes different time for different secret-dependent numerators.

The vulnerable source used division by a public constant. Developers might expect a compiler to replace such division with fixed-latency arithmetic, but compiler choice, optimization settings and target hardware can affect the generated instructions. The paper shows circumstances where variable-time division instructions remained. A source-level check for secret branches or secret-indexed memory alone would not establish that these instructions reveal no timing information.

KyberSlash1 concerns message decoding in the underlying decryption. KyberSlash2 concerns compression work during re-encryption inside decapsulation. That second context is instructive: bytes that will eventually be part of a public ciphertext in an ordinary encryption can depend on a private intermediate when generated for a decapsulation check. Labeling all encryption outputs public without tracking the call context can therefore miss a leak. The relevant question is when a value is allowed to become public, not just which helper function contains it.

Table 17.1. Three bounded historical failure settings. The affected conditions identify the case studied; the table is not a vulnerability assessment of current products.

Case	Attacker’s useful observation	Boundary of the result
Bleichenbacher, 1998	Formatting answers about related RSA ciphertexts	Historical PKCS #1 v1.5 oracle setting, with acceptance details governing the attack.
Case Brumley– Boneh, 2003	Attacker’s useful observation Timings of RSA private operations over a tested local network	Boundary of the result Specific old OpenSSL configurations; blinding changed the tested result.
Case KyberSlash, 2025	Attacker’s useful observation Secret-dependent division timings during chosen-ciphertext decapsulation	Boundary of the result Affected historical Kyber code, compiler and processor combinations; not a lattice-problem break.



The next query may depend on the observed answer.
A correct final value does not establish that no extra information leaked.

Figure 17.1. A failure investigation follows both the intended result and additional observations. An attacker can adapt later inputs using an error distinction or timing evidence. Protecting the private calculation requires checking the relevant observable behavior and application use, not only its mathematical result.

These cases suggest a disciplined way to evaluate a reported correction. Identify the algorithm and version, the relevant processing path, the attacker-controlled input and the observable difference. Then ask which change removes that dependence or makes the query ineffective. Confirm that required valid operations still work and that the identified invalid or adversarial path no longer has the tested behavior. The resulting claim should remain limited to that evidence.

Automated analysis can help locate likely misuse. CryptoGuard, presented at ACM CCS in 2019, studies detection of cryptographic API misuse in large Java projects, including predictable randomness, exposed secrets and deficient verification behavior. Such tools analyze specified patterns and program flows, with tradeoffs involving false alerts and missed cases. Detecting a misuse is different from proving that the underlying cipher is weak; failing to report a misuse is different from proving that an entire application is secure. [\[101\]](#)

Source code, compiled behavior and deployment configuration also need to agree. Enabling a defense in a library is insufficient if the application disables it or follows another path. A source correction may need to reach the actual binary and the configured service before the studied exposure changes. Conversely, a historical report about one unprotected version should not be repeated as an unqualified warning about all implementations carrying the same algorithm name.

One further lesson concerns the order of trust decisions. Decode only under specified rules, authenticate before using data whose integrity is required, enforce context and authorization, and commit state only after the necessary checks succeed. Exact processing order depends on the construction, so this is not permission to reorder a standardized protocol casually. It is a reminder to identify which data are trusted at each stage and which observations can occur before they become trusted.

Good cryptography fails in practice when the implemented system leaves the argument's conditions behind. Secret reuse can remove unpredictability or uniqueness; oracles can expose private intermediate structure; timings can reveal secret-dependent work; and context mistakes can turn authentic bytes into the wrong action. The appropriate response is to repair the identified boundary and gather evidence for the revised claim. The next chapter examines how tests and verified software contribute to that evidence, including what each method can and cannot establish.

Questions to check your understanding

1. Why can a valid-length key or nonce be unsuitable for a construction? Which assumptions support Equation [17.1](#), and why can machine cloning invalidate them?
2. What is an oracle, and what makes a chosen-ciphertext attack adaptive? Does one-bit acceptance information necessarily remain harmless after many selected queries?
3. In Equation [17.2](#), which operation uses only public information? Why does an accepted PKCS #1 query imply an interval but not vice versa?
4. In the decimal model, derive the bounds 250 and 259 from

$$n = 1147$$

,

$$s = 10$$

, the accepted interval 200–299 and

$$r = 2$$

. What has one query not yet established?
5. Explain how the CBC XOR relation differs from RSA multiplication. Why must the PKCS #7 byte example not be treated as TLS CBC's exact padding convention?
6. In the timing model, identify

$$T_0$$

,

$$\delta$$

,

$$b$$

and

$$\epsilon_i$$

. Why can repeated independent measurements reveal a small difference, and why is the modeled sample count not a measured attack rate?

7. How does RSA blinding preserve the final plaintext? Why do random delays or removing source-level branches not prove the absence of side channels?
8. Why is the two-field `abc` ambiguity not a hash collision? Distinguish framing, canonical representation, domain separation and authorization.
9. What kind of failure did KyberSlash demonstrate? Evaluate the reduced decoding formula at
 $t = 833$
, and explain why a correct output does not establish secret-independent execution.
10. Why do a passing round-trip test, a clean misuse-detector report and a historical patch each support different conclusions? What evidence is still needed about actual configuration and exposed state?

18. Testing and Verified Software

Part V: Building Secure Systems

Nora replaces an encryption component after investigating an error. Her demonstration still opens a protected file, and the program reports success. That is useful evidence, but it leaves several questions unanswered. Would the program produce the bytes required by the specification? Would it reject a modified tag? Could its optimized arithmetic read outside a buffer? Could its execution reveal something about a secret even when the final result is correct? Chapter [17](#) showed why those questions cannot be answered by one successful recovery.

Testing and formal verification approach these questions in different ways. A **test** executes a program under selected conditions and compares the observations with an expectation. **Formal verification** uses mathematical models and proof tools to establish a stated property of the behaviors covered by the model. Testing can expose a defect with one counterexample. A proof can cover a whole class of permitted inputs, but its meaning depends on the specification, assumptions and connection to the program that actually runs.

We will first examine what makes a test informative, then follow negative cases and the guarantees of verified cryptographic components. HACL*, HACLxN and EverCrypt provide concrete research examples. An optional Python program checks a published ChaCha20-Poly1305 result and selected rejection cases. You can follow the reasoning without running it: the main task is to identify the expected result, the evidence that supports it and the conclusions that remain outside its scope.

18.1 Known-answer tests and independent expectations

A **known-answer test** supplies defined inputs and an expected output obtained from an identified source. The implementation under examination calculates its result, and the test compares the two. A published encryption vector might specify a key, nonce, associated data, plaintext, ciphertext and

tag. These are exact byte strings, not suggestions about equivalent-looking text. If the specification says that a value is little-endian, interpreting its hexadecimal display in another order changes the input and therefore the question being tested.

The expected answer should not simply be whatever the tested program produced earlier. Saving that result and checking that it repeats can detect later changes, but it does not establish that the original result was correct. We need an independent basis for the expectation: a carefully checked calculation, an identified standard's vector or another appropriately validated reference. Independence is a matter of how the evidence was obtained. Two wrappers that call the same underlying function do not provide two independent implementations merely because their names differ.

A **round-trip test** instead checks that a recovery operation undoes a forward operation. That is valuable for finding some interface and representation mistakes. However, two operations can agree with each other while both disagree with the intended specification. They may share the same wrong byte order, omit the same authenticated field or interpret one parameter incorrectly in matching ways. A round trip is therefore one kind of test, rather than a substitute for an externally established result.

Consider a teaching transformation on the integers 0 through 255, the range of one byte. The intended rule adds one and keeps the remainder after division by 256. A mistaken forward rule adds two, while its matching recovery rule subtracts two, both with the same remainder operation:

(18.1)

Here
 x
is an input byte,
 S
is the intended specification,
 F

is the mistaken forward operation and

R

its matching recovery operation. For

$$x = 10$$

, the intended output is 11, while

F

produces 12 and

R

turns 12 back into 10. In fact,

$$R(F(x)) = x$$

for every byte, despite

F

disagreeing with

S

everywhere. These are arithmetic teaching functions, not proposed encryption algorithms.

Tests should also explore boundaries. For the intended rule,

$$S(255) = 0$$

, since

$$255 + 1 = 256$$

and the remainder is zero. A program that returns the ordinary integer 256 has left the byte range. Testing only inputs below 255 would miss this particular defect. Empty messages, partial blocks, maximum permitted lengths and transitions between storage sizes have similar importance in actual implementations, although their expected behavior must come from the relevant interface. A boundary is not automatically invalid; it is a place where the rule deserves explicit checking.

Random testing can explore a broader space, but a large trial count does not guarantee that a particular boundary was reached. In a simplified model, let

ϵ

be the fraction of inputs that expose a defect under a stated sampling distribution. If

N

draws are independent, the probability that all of them miss that set is

$$(18.2)$$

The symbol

$\Pr$

denotes probability,

N

counts the draws and the exponent means multiplying the same miss probability

N

times. The formula assumes that the revealing set and sampling distribution are fixed. It does not estimate the fraction of unknown vulnerabilities in a cryptographic library.

For the byte-boundary defect above, exactly one of 256 uniformly sampled inputs reveals the missing wrap, so

$$c = 1/256$$

. After 256 independent draws with replacement, the miss probability is

$$(255/256)^{256}$$

, about

$$0.367$$

, or 36.7 percent. Repeating a random draw is different from deliberately testing all 256 distinct inputs. An exhaustive check is practical for this tiny domain; the combined domain of real keys, messages, state and configurations is much larger. Moreover, an attacker can deliberately choose a known revealing input instead of sampling like the test.

Property-based testing generates cases and checks a general relationship, such as an output-length rule or agreement between incremental and whole-message processing. **Differential testing** compares implementations or execution paths. Both can discover mistakes beyond a small vector collection, but the property or reference can itself be wrong. Two implementations can share a bug, and a length-preserving function can still compute the wrong bytes. Complementary expectations are useful because they constrain different parts of the behavior.

A **regression test** records a case that previously exposed a defect, together with its correct outcome. Its purpose is to detect a recurrence when code changes. The expected failure should be specific enough to distinguish a repaired cryptographic check from an unrelated crash. If a test intended to reject a modified tag instead fails because the library never imported, it has not demonstrated tag verification. The test harness, which prepares inputs and interprets results, is part of the evidence and deserves examination too.

18.2 Negative tests with Wycheproof

A **negative test** asks whether the system refuses an input or operation that the relevant rule requires it to refuse. For authenticated encryption, a valid ciphertext should recover the expected message, while selected changes to protected bytes or authenticated context should fail. For signatures, a verifier needs examples of valid evidence and carefully chosen invalid encodings or mathematical values. Running only successful examples leaves the acceptance boundary largely unexplored.

Project Wycheproof supplies public test vectors designed to probe known attacks, specification inconsistencies and implementation mistakes. The collected repository is maintained within C2SP. Its cases are organized as data with accompanying schemas, so a library developer can map them to a particular interface. This is useful precisely because unusual cases may require more thought than an ordinary demonstration. A vector is a precise question about behavior, not an assurance label attached to any program that downloads the file. [\[108\]](#)

The AEAD schema separates the key, nonce, associated data, plaintext, ciphertext body and tag. It also records a case identifier, explanatory flags and an expected classification. Some library interfaces return the ciphertext body and tag together, while the data file stores them separately. A harness must combine them in the documented order before decryption. If it accidentally includes the nonce in that concatenation, a rejection could reflect the harness's input mistake rather than the behavior the vector was designed to examine.

Size units also matter. A group field can report a nonce length in bits, while a library argument is an array whose length is counted in bytes. Ninety-six bits correspond to twelve bytes because each byte contains eight

bits. Reading a size field as a byte count could select the wrong cases or construct the wrong input. A hexadecimal string also represents bytes: two hexadecimal digits encode one byte. Comparing the number of printed characters with a specified byte length confuses representation with data.

The result labels require interpretation. A `valid` case expects the defined successful behavior. An `invalid` case expects rejection under its relevant rules. An `acceptable` label is not a universal instruction that every application must accept the input; its meaning must be read with the algorithm's notes, flags and the selected API or policy. A responsible harness records how such cases are classified. It should not silently count every exception as a successful negative test or every permissive result as a desirable feature.

The chosen profile must be stated before results are counted. A file may contain inputs for several nonce sizes, while an interface supports only one selected size. Excluding an out-of-profile group can be legitimate, but the report should say what was excluded and why. Unsupported, skipped, accepted and rejected are different outcomes. If a required valid case cannot run, reporting it as a pass would hide a coverage gap. Conversely, a deliberately narrower protocol need not accept every format supported by a general-purpose primitive.

For the accompanying local checks, we selected the 96-bit-nonce, 128-bit-tag groups from two collected files: AES-GCM and the IETF ChaCha20-Poly1305 construction. Together those groups contain 513 cases: 372 marked valid and 141 marked invalid. In the tested interface, every selected valid case matched encryption and plaintext recovery, and every selected invalid case was rejected. Another 128 cases belonged to other nonce-size profiles and were excluded from this selected run. No `acceptable` cases occurred in the selected groups. [\[112\]](#)

Those counts describe two data files, one profile selection and one installed library environment. They do not describe the whole Wycheproof project, every optimized backend or every supported operating system. A future test-data change can also change the denominator. Preserving the selected data identity and the counting rule is therefore as important as preserving the number. A statement such as all tests passed is incomplete until a reader can identify what all includes.

Negative tests can also examine application state. Suppose Sam receives a modified message and authentication fails. The application should not nevertheless record the message as delivered, release unverified contents or discard state needed for a legitimate retry unless its specified protocol requires that action. The relevant test needs observations of the application's effects, not just an exception from the cryptographic call. Chapters [15](#) and [16](#) showed why message order, retries and state transitions are part of the protection.

Fuzzing systematically supplies many generated or mutated inputs while observing properties such as crashes, parser behavior or disagreements. It can find cases the original test author did not anticipate. Its value still depends on which paths are reached and which outcomes are recognized as wrong. A parser that accepts a dangerous structure without crashing may escape a crash-only harness. An instrumented build that detects invalid memory access supplies a different observation from an ordinary build. Results should identify that distinction.

The useful outcome of a failed case is an explanation. First check the vector's interpretation, the intended algorithm variant and the interface's documented contract. Then determine whether the disagreement lies in the harness, the implementation, the specification or an explicitly different policy. Once understood, the case can become a regression test. Deleting a difficult vector merely to restore a green report would remove evidence while leaving the underlying question unanswered.

18.3 What functional and memory-safety proofs establish

A **specification** states the behavior or property an implementation is meant to satisfy. It can describe an algorithm with simple mathematical functions, the valid memory objects available at a call or which observations may depend on a secret. Specifications make an intention precise enough to reason about, but precision does not guarantee that the intention was chosen correctly. A perfectly formal description can still omit a required authentication check or encode the wrong byte order.

For a simplified stateless calculation, write

I

for the implementation and

S

for the specification. Let

D

be the input domain and

$P(x)$

the conditions required before processing input

x

. A functional-correctness claim can have the form

(18.3)

The notation

$x \in D$

says that

x

belongs to the stated domain. The arrow means implies: when the required conditions

$P(x)$

hold, the implementation's output agrees with the specification's output.

The phrase for every concerns all inputs covered by that claim, rather than just the inputs selected for a test. For stateful software, the description must also account for the relevant initial state, changes and permitted observations.

A **precondition** is a condition required before an operation. A **postcondition** describes what must hold afterward for the behavior being specified. For a function writing a tag into an output array, a precondition might require that the array contain sufficient writable storage and that overlapping inputs follow a permitted rule. A postcondition can identify the resulting tag bytes and which other memory is unchanged. The proof connects the two. It does not supply missing storage to a caller that violates the precondition.

These conditions have concrete mathematical content. An array of length

L

has indices 0 through

$L - 1$

. Accessing index

i

requires

(18.4)

Here

L

counts the available elements and

i

selects one of them. If

$L = 16$

, the last valid index is 15, so the expression

$i \leq L$

would incorrectly include index 16. A memory-safety argument must also account for whether the array is still alive, which permissions exist and how related references are used. Checking the numerical bound alone is not a complete proof of safe memory access.

Memory safety rules out specified violations of the memory model, such as accessing outside an allocated object or using an object after it has been released. It is especially important in low-level cryptographic code because an unintended read can expose secret material and an unintended write can corrupt a security decision. However, a memory-safe function can still return the wrong digest or authenticate the wrong field. Memory safety and agreement with the algorithm specification answer different questions.

Machine arithmetic introduces another boundary. Mathematical integers have no fixed storage width, while a 32-bit unsigned word has a defined finite range. Addition modulo

2^{32}

keeps only the remainder after division by

2^{32}

. In that model, adding one to

$2^{32} - 1$

gives zero. Some cryptographic operations deliberately require this wrap, while a length calculation may need to detect an overflow instead. A correctness proof must relate the chosen representation to the intended arithmetic. It cannot assume that every overflow is either always harmless or always forbidden.

Secret independence concerns permitted observations of executions with different secret values. Under a particular model, a proof can establish that secrets do not affect branches, accessed addresses or certain variable-time operations. This is distinct from requiring ciphertexts or signatures to be identical for different keys. Those are intended outputs of the cryptographic function. The model identifies which extra observations should not disclose secrets and which public quantities, such as allowed input lengths, may influence execution. [\[109\]](#)

Such a guarantee must be read with its leakage model. Preventing secret-indexed memory access is valuable, but does not automatically model electromagnetic emissions, all speculative execution effects or every data-dependent instruction on every processor. Chapter [17](#) explained why a correct division result can still have unsafe timing in a particular compilation and hardware setting. Functional correctness and secret independence can be proved at different layers; stating one property without the layer and model loses essential information.

A proof of algorithm implementation is also different from a **cryptographic security proof**. The first might show that a program computes the specified ChaCha20-Poly1305 function. The second connects a construction's protection goal to assumptions and an attack model, often with quantitative limits and input conditions. If a caller repeats a nonce outside those conditions, faithful implementation of the function does not restore the lost guarantee. Likewise, proving a legacy function correct does not make its security level suitable for a new application.

Formal tools help discharge the many obligations involved. A **proof assistant** checks mathematical reasoning expressed in a formal language; automated solvers can help establish arithmetic and logical consequences. The result is not produced by simply labeling a source file verified. The code, specification and supporting arguments have to fit the tool's rules. Changes can require renewed checking, and a failed proof attempt may indicate a missing argument, an unsuitable specification or an actual defect. It does not by itself identify which explanation is correct.

The connection to the real program remains a chain of claims. A source proof refers to a particular program model. A compiler transforms that source, an assembler and linker may construct the executable, and an operating environment invokes it with actual inputs. Some steps may have their own correctness results; others remain trusted assumptions. A component proof is therefore meaningful evidence within a larger account, not a reason to stop examining the surrounding software.

Table 18.1. Different evidence answers different questions. Each conclusion retains the stated implementation, model and coverage limits.

Evidence Known-answer test	Direct question Does this execution match an identified expected result?	What remains outside it Untested inputs, other execution paths and the origin of a wrong expectation.
Evidence Negative test	Direct question Is this specified bad case refused, with the required effects?	What remains outside it Other bad cases and unobserved application behavior.
Evidence Functional proof	Direct question Does the implementation match its formal specification under its contract?	What remains outside it Whether that specification supplies the intended security goal.
Evidence Memory-safety proof	Direct question Are the modeled memory rules preserved under the assumptions?	What remains outside it Wrong algorithmic results, caller policy and unmodeled leakage.
Evidence Secret-independence proof	Direct question Are the specified extra observations independent of secrets?	What remains outside it Physical or processor effects outside the leakage model.

18.4 HAClXN, EverCrypt and the remaining trusted components

HACL* provides a concrete example of developing cryptographic software with explicit proof goals. The original 2017 paper by Zinzindohoué and colleagues describes a portable library whose source is verified for memory safety, functional correctness and secret independence. Its functional claims relate implementations to mathematical specifications derived from the algorithms. The paper explicitly distinguishes this work from proving the cryptographic security of every construction. We should preserve that distinction when explaining what the library contributed. [\[109\]](#)

The project uses F*, a language and verification framework, and Low*, a subset suited to low-level code with explicit memory behavior. A high-level specification can use mathematical sequences and integers for clarity. An optimized implementation instead needs machine words, arrays and concrete representations. Proofs relate those two descriptions. The reader does not need to learn F* syntax to understand the main obligation: the efficient procedure must have the specified meaning for every input covered by its contract.

This separation is useful because a readable specification and a fast implementation serve different purposes. A short mathematical field multiplication may be implemented using several machine words with carefully managed carries. Replacing that implementation with the formula on a page would ignore storage and performance constraints. Replacing the specification with a copy of every optimization would make independent examination difficult. The proof establishes the connection while keeping the descriptions suited to their roles.

HACLxN extends this approach to generic vectorized cryptographic code. **SIMD**, short for single instruction, multiple data, lets one instruction operate on several data lanes. For example, several independent word operations can be processed together. Cryptographic software can use such parallelism to process parts of an algorithm or several blocks efficiently. The challenge is to rearrange the computation without changing which mathematical result belongs to which input. The 2020 paper by Polubelova and colleagues develops shared code and proofs that can be specialized for different implementations and platforms. [\[110\]](#)

Sharing proof structure can reduce repeated work, but does not remove the need to justify a transformation. If one representation arranges words by

block and another arranges them by lane, a correspondence must explain the rearrangement. The operations on the rearranged data must then preserve the intended calculation. This is more informative than observing that one vectorized run happened to equal one scalar run. Differential tests remain useful, while the proof addresses the general relationship expressed by the models.

There is a clear compiler boundary. The HACLxN paper describes translation from verified Low* programs into C through the toolchain historically called KreMLin; the collected project documentation uses the name KaRaMeL. Its discussion explicitly identifies the remaining trust in mainstream C compilers used for the reported SIMD builds. A compiler can introduce a functional mistake or behavior outside a source-level leakage argument. The paper's account is not a claim that all later compilers or all current target architectures inherit a universal machine-code guarantee.

A **trusted computing base**, often abbreviated TCB, consists of the components and assumptions that the argument still relies on rather than establishing within its own proof. This can include the formal specification, the soundness of the checking tools, machine-instruction models and particular translation or execution steps. Calling something trusted describes a dependency in the argument. It does not mean that someone has proved it flawless, and it does not mean that formal verification has no value because dependencies remain.

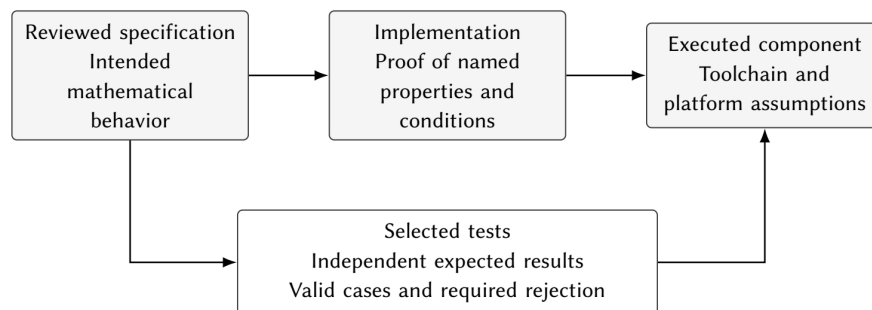
EverCrypt addresses another practical problem: presenting multiple cryptographic implementations through a coherent provider. Its 2020 paper by Protzenko and colleagues distinguishes **agility**, choosing among algorithms for a functionality, from **multiplexing**, choosing among implementations of one algorithm. Selecting SHA-256 or another hash is an algorithm choice. Selecting a scalar or vectorized implementation of the same specified ChaCha20-Poly1305 operation is an implementation choice. The meanings should not be confused simply because both choices occur below a common interface. [\[111\]](#)

EverCrypt combines code developed through the HACL* and Vale approaches. Vale supports verification of low-level assembly routines, which can be important when performance depends on particular machine instructions. Connecting C-like and assembly-like components requires

compatible models of calling conventions, memory and the intended operation. A fast routine that calculates the correct value in isolation is not enough if its caller passes the wrong representation or assumes that different registers and memory remain unchanged.

The collected ChaCha20-Poly1305 provider source makes implementation selection visible: selected paths choose a vectorized implementation when the relevant capability is available, with a scalar route also present. Reading that dispatch is useful evidence about the code's structure, but it is not a new execution of every route. The paper explains the verification design; the current collected source identifies a particular implementation arrangement. Neither observation justifies assigning the same proof status to an unrelated library merely because it implements ChaCha20-Poly1305. [\[111\]](#)

Figure [18.1](#) separates these responsibilities. A reviewed specification describes what should be calculated. The implementation and its proof supply one connection, and the toolchain must preserve the relevant properties on the route to execution. Tests compare selected observations with independently established expectations. The application surrounds the entire route with inputs, state and use of outputs. An error at one of those boundaries cannot be dismissed just because another arrow has strong evidence.



Application boundary: prepare permitted inputs, preserve state rules, handle failure and use accepted outputs correctly.

Figure 18.1. Evidence connects a specification to an executed component and its use. A proof covers named properties and assumptions; tests compare selected observations. The application's inputs and use of results remain part of the security argument.

The EverCrypt paper also describes precautions around its own trusted specifications: independent review against the original algorithms and execution of extracted specifications on standard vectors. This is an important connection between proof and testing. An implementation can faithfully compute a mistaken specification. Testing the specification against external examples can expose an incorrect constant or byte-order interpretation before that mistake becomes the target of a larger proof. The methods reinforce one another when their expectations come from distinct evidence.

Testing continues at deployment boundaries too. The HAClXN account describes review, platform testing and staged integration alongside verified source. That is a historical workflow, not a verified description of every present-day browser release. Its lesson is that operating systems, compiler settings, bindings and integration can add questions beyond a component's formal result. A caller written in another language must still respect lengths, object lifetimes, permitted overlap and failure handling.

Finally, a verified provider cannot make an unsafe host application behave safely. If another component exposes the process's memory, the provider's private state may become readable without violating its own algorithmic specification. If a caller ignores an authentication failure or supplies repeated nonces, the problem concerns the surrounding use. The appropriate response is to carry the component's contract into the application design, identify remaining trusted steps and test the boundaries that the component proof does not cover.

18.5 A reproducible library example

We can now assemble a small evidence record. The example uses the IETF ChaCha20-Poly1305 AEAD construction already explained in Chapter [7](#). Its independent expected bytes come from RFC 8439, Section 2.8.2. The earlier chapter identified the tag and authentication-input lengths; here the purpose is different: verify the complete library result and explicitly check selected rejected cases. The key, nonce and message are published test data, so no generated secret or live network service is involved. [\[15\]](#)

The Python interface takes a 32-byte key and a 12-byte nonce and returns the ciphertext body followed by the 16-byte tag. The program therefore stores the published ciphertext and tag together in `expected`. It compares encryption with those independent bytes before checking decryption. Reversing the same library operation alone would provide weaker evidence. The decryption failure expected for a changed protected input is `InvalidTag`; an unrelated exception is not counted as equivalent success. [\[112\]](#)

Listing 18.1. Checking one complete published AEAD result and five selected rejection cases. The constants are public test material.

```
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.primitives.ciphers.aead import (
    ChaCha20Poly1305,
)

key = bytes(range(0x80, 0xa0))
nonce = bytes.fromhex("070000004041424344454647")
aad = bytes.fromhex("50515253c0c1c2c3c4c5c6c7")
message = (
    b"Ladies and Gentlemen of the class of '99: "
    b"If I could offer you only one tip for the future, "
    b"sunscreen would be it."
)
expected = bytes.fromhex(
    "d31a8d34648e60db7b86afbc53ef7ec2a"
    "4aded51296e08fea9e2b5a736ee62d63"
    "dbea45e8ca9671282fafb69da92728b1"
    "a71de0a9e060b2905d6a5b67ecd3b369"
    "2ddb7f2d778b8c9803aee328091b58fa"
    "b324e4fad675945585808b4831d7bc3ff"
    "4def08e4b7a9de576d26586cec64b6116"
    "1ae10b594f09e26a7e902ecbd0600691"
)

def flip(data, index):
    return (
        data[:index]
        + bytes([data[index] ^ 1])
        + data[index + 1:]
    )

aead = ChaCha20Poly1305(key)
if aead.encrypt(nonce, message, aad) != expected:
    raise AssertionError("Published encryption result differed")
if aead.decrypt(nonce, expected, aad) != message:
    raise AssertionError("Published recovery result differed")

bad_cases = [
    (nonce, flip(expected, 0), aad),
    (nonce, flip(expected, len(expected) - 1), aad),
    (nonce, expected, flip(aad, 0)),
    (flip(nonce, 0), expected, aad),
    (nonce, expected[:-1], aad),
```

```
]
for trial_nonce, trial_data, trial_aad in bad_cases:
    try:
        aead.decrypt(trial_nonce, trial_data, trial_aad)
    except InvalidTag:
        continue
    raise AssertionError("A selected bad case was accepted")

print("Published encryption and recovery matched")
print("Five selected bad cases were rejected")
```

The selected changes concern the ciphertext body, tag, associated data, nonce and truncated protected input. The program must preserve the authentic original bytes for each case; applying changes cumulatively would ask a different question. The helper changes one bit by XOR with 1. Decryption remains an authenticated operation: the example never presents a partially recovered message as accepted plaintext after a failure. A test should also report unexpected errors, rather than catching every exception and printing success.

These altered inputs are fixed examples whose rejection is checked. They do not imply that a 128-bit tag makes forgery mathematically impossible or that five mutations exhaust the attacker space. Other tests can cover empty plaintexts, different lengths and permitted input boundaries. The broader selected Wycheproof run in Section [18.2](#) complements the single published vector. It still concerns selected executions, not a proof of every supported operation.

A reproducible record identifies more than the Python package name. The local run used `cryptography` 50.0.2 and reported OpenSSL 4.0.3 as its backend. That backend belongs to the installed runtime; it is not evidence that the separately collected OpenSSL source was built for this book. The record also identifies the input-data snapshot, selected profiles and expected failure class. Platform, build options and available processor features can affect the executed path, so another environment should record its own identity rather than copy this one.

There is also a difference between reproducibility and identical random output. When an operation legitimately generates fresh secrets or nonces, another run can produce different bytes while satisfying the same property. A known-answer test controls those inputs to compare an exact result. An integration test with fresh keys can instead check appropriately defined relationships and rejection behavior. Fixed public vectors belong in testing;

they should not quietly become permanent production keys or repeated nonce allocations.

Optional case: file evidence and a pinned publisher key

The confirmed REA research-tool repository supplies a useful example of testing conclusions outside a cryptographic library. Selected source paths calculate file digests and verify a detached OpenPGP signature against an expected publisher identity. A **detached signature** is stored separately from the data it authenticates. A **pinned key identity** is an expected identifier supplied through a trusted configuration or other trusted decision, rather than accepted solely because an untrusted download provides it. [\[115\]](#)

The file-hashing path reads bytes through an admitted file handle and checks selected file-state information around the read. This matters because a checksum refers to the bytes actually processed, not an abstract filename that is guaranteed to refer to one unchanging object forever. The inspected code supplies specific checks; it does not establish a universal proof that every filesystem race is impossible. As Chapter [8](#) explained, a matching unkeyed digest alone also does not identify a publisher if an attacker can replace both the file and its advertised digest.

The selected signature path reads a public key, checks its primary fingerprint against the pinned value, verifies one detached binary signature and waits for the verification result. A fingerprint is a compact identifier derived from a key; its connection to the intended publisher still depends on how the pinned value was obtained. Test definitions include changed payloads and a different primary key, along with other parsing and operation cases. These are inspected source and test definitions. We did not execute REA's complete application test suite or certify its dependency chain.

The useful questions are therefore specific: which bytes are covered, where does the expected key identity come from, and which disagreement causes rejection? File-size checks, content digests, key identity and signature verification have different roles. Passing one does not eliminate the need for the others. This example connects Chapters [8](#) and [13](#) to a real

source path without turning a research application into a new cryptographic standard or an application-cloning exercise.

Identifying an algorithm is another kind of evidence

Researchers also study how to identify cryptographic functions in binary programs. CryptoHunt uses bit-precise symbolic loop relationships and comparison with reference implementations. BinCrypto studies behavioral similarity through path-insensitive emulation, in which execution possibilities are summarized during analysis. Their results concern identifying candidate functions under the investigated data and transformations. An identification result does not establish safe key generation, correct nonce management or authenticated use by the surrounding program. [\[113\]](#)[\[114\]](#)

Likewise, a misuse detector asks whether specified patterns of API use violate modeled rules, as illustrated by CryptoGuard in the previous chapter. That differs both from recognizing a function in compiled code and from proving an implementation's memory or functional properties. A reader should ask which of these questions a tool answers before treating its report as evidence for another. We do not infer that REA implements these research methods merely because it also supports software inspection. [\[101\]](#)

We can now state a useful standard for technical claims. Identify the component, the intended property, the model or test cases, and the remaining dependencies. Keep the specification, actual program and report consistent when any of them changes. A passing test or a verified component is valuable because its scope can be explained, not because its name ends every further question. The next part of the book changes one of the largest assumptions in the attack model: what happens when an attacker can use a sufficiently capable quantum computer.

Questions to check your understanding

1. Why can a round trip pass even when both operations disagree with a specification? Evaluate $S(10)$, $F(10)$ and $R(F(10))$ in Equation [18.1](#).
2. What makes an expected output independent of the implementation under test? Why can two differently named wrappers still share the same mistake?
3. Under the stated random-sampling model, why is the chance of missing the one-byte boundary defect after 256 draws about 36.7 percent? Why is this not an estimate of library security?
4. In an AEAD vector, distinguish nonce bytes, associated data, ciphertext body and tag. What error could arise from mixing a bit count with a byte count?
5. Why should a test report distinguish invalid, acceptable, unsupported and excluded cases? What exactly do the selected 513 Wycheproof outcomes establish?
6. Read Equation [18.3](#) in words. What roles do the domain and precondition play, and why can a correct implementation still follow a mistaken specification?
7. For an array of length 16, why does $i \leq 16$ permit an invalid access? Why does checking an index bound alone not establish complete memory safety?
8. Distinguish functional correctness, memory safety, secret independence and cryptographic security. Give an application mistake that a verified primitive cannot prevent by itself.

9. What is the difference between algorithm agility and implementation multiplexing? Which trusted steps can remain between a source proof and an executed program?
10. What do the published AEAD check, the REA source inspection and a binary function-identification result each establish? Which facts would be missing from a claim that they certify an entire application?

VI. The Next Questions

19. Quantum Computers and Post-Quantum Cryptography

Part VI: The Next Questions

The earlier chapters asked what an attacker could calculate with available information and resources. That question has a physical dimension as well as a mathematical one. An algorithm that is difficult for known classical methods may have a different cost on a quantum computer. The change does not mean that every encrypted file suddenly becomes readable or that mathematics stops protecting information. It means that we must identify which computational assumptions change, which constructions depend on them and what kind of machine the attack requires.

Post-quantum cryptography develops classical algorithms intended to resist both classical and quantum attacks. A person using such an algorithm does not need a quantum computer. The term describes the attack model, rather than the hardware required by an ordinary sender. It also does not mean security against every possible future discovery. As with the constructions already studied, the argument depends on a defined scheme, parameters, implementation and use.

We will first build a working picture of quantum information, gates and measurement. Simple circuits will show how interference and entanglement enter a calculation, and an examination of hardware will explain why noise matters. IBM's systems and Qedma's error-handling research provide concrete examples of the engineering work. We can then distinguish Shor's and Grover's algorithms, develop noisy equations and connect the standardized post-quantum mechanisms to familiar cryptographic roles. The final sections examine migration, old recordings and the limits of forecasts.

19.1 What changes in the attack model

An ordinary computer is already a physical machine governed by the laws of nature. Calling another computer quantum does not mean that its components are made from an entirely different kind of matter. The

difference is how information is represented, controlled and measured. An ordinary digital design maintains states that can be treated as stable classical bits. A quantum processor deliberately controls quantum states and their relationships long enough to carry out a specified computation. It then produces classical measurement records that an ordinary computer can process.

One motivation came from physics itself. Richard Feynman's 1982 paper examined the problem of simulating physical systems with computers and the possibility of using quantum systems for that purpose. A detailed classical description of a general many-particle quantum state can become extremely large. A controlled quantum system offers another route to studying quantum behavior. This was an argument about a particular computational challenge, rather than a promise that every task would become faster. [\[126\]](#)

This distinction gives us two different meanings of simulation. A classical program can calculate the ideal behavior of a small quantum circuit by storing and updating its amplitudes. A quantum processor physically implements operations on quantum information. The small classical simulation is an excellent teaching and checking tool. It does not establish that the same method remains affordable for an arbitrary large circuit, nor that every large circuit necessarily defeats all specialized classical techniques. The structure of the circuit and the requested output matter as much as its qubit count.

A classical bit has two possible values, 0 and 1. A **qubit** carries quantum information using two reference states. A basis state is one of the reference states used to describe the system. We begin with a **pure state**, described by one normalized state vector. For one qubit, we write

$$(19.1)$$

The expression

$$|\psi\rangle$$

names the state; the vertical line and angle bracket are notation, not an absolute-value calculation. The states

$$|0\rangle$$

and

$$|1\rangle$$

are the reference states. The coefficients

$$\alpha$$

and

$$\beta$$

are **amplitudes**, numbers used to determine measurement probabilities. In general they are complex numbers. The squared magnitude

$$|\alpha|^2$$

is the probability of obtaining 0 in this basis, and

$$|\beta|^2$$

is the probability of obtaining 1. Their sum is one because those are the two possible outcomes of this measurement.

For example, take

$$\alpha = \sqrt{3}/2$$

and

$$\beta = 1/2$$

, both real in this example. Squaring gives

$$3/4$$

and

$$1/4$$

. Repeatedly preparing and measuring that state gives probabilities of 75 percent for 0 and 25 percent for 1; one measurement does not return both values. It also does not provide a list containing the exact amplitudes.

Preparing many copies and collecting measurement results is different from obtaining unlimited information from one state.

The word complex describes a precise number system, rather than an instruction to make the explanation difficult. Write a complex number as

$$a + ib$$

, where

$$a$$

and

b

are real numbers and

i

satisfies

$$i^2 = -1$$

. Its squared magnitude is

$$(a + bi)^* (a + bi) = a^2 + b^2$$

. Thus the squared magnitude of

$$(3 + 4i)/5$$

is

$$(3^2 + 4^2) / 5^2 = (9 + 16) / 25 = 1$$

The two components affect how amplitudes combine, while the squared magnitude gives a nonnegative probability contribution. One should not square a complex amplitude as if the answer itself were an ordinary probability.

A **mixed state** needs a more general description than one pure-state vector. For example, preparing

$$|0\rangle$$

or

$$|1\rangle$$

with equal classical probabilities, while forgetting which was prepared, gives an equal mixture. It is different from the pure superposition with amplitudes

$$1/\sqrt{2}$$

and

$$1/\sqrt{2}$$

. Both give equally likely 0 and 1 outcomes in the standard basis, but a later gate can distinguish their behavior. A density matrix is the mathematical description that covers both pure and mixed states. We will use simple preparations and probabilities to explain this distinction without requiring its full matrix formalism. [\[127\]](#)

An amplitude may also be negative. The square of

$$-1/\sqrt{2}$$

is

$$1/2$$

, just as for

$$1/\sqrt{2}$$

. That does not make the signs irrelevant: amplitudes can add or cancel before the probability is calculated. Measuring too early can lose the relationship that a later operation was intended to use. Consequently, a list of the probabilities for one immediate measurement does not always describe everything needed to predict a later experiment. [\[127\]](#)

Superposition refers to such a combination of basis states. A register of n

qubits can have amplitudes associated with 2^n

basis strings, but this description is not an interface that lets a user read 2^n

classical answers. A useful algorithm has to arrange operations so that measurement reveals information relevant to its task. Simply evaluating a problem in superposition and then measuring does not guarantee the desired answer.

Interference describes how amplitudes combine when operations bring contributions together. Relative phase, a relationship between amplitudes, can cause contributions to reinforce or cancel. Quantum algorithms exploit that structure to change the probability of useful measurement outcomes. This is the reason that a carefully designed algorithm can outperform an ordinary search, rather than a rule that a quantum computer automatically tries everything and reports every successful result. [\[117\]](#)

We should also distinguish **quantum cryptanalysis** from **quantum key distribution**. The first uses quantum computation to attack a protection. The second uses quantum communication techniques as part of establishing keys under its own physical and authentication assumptions. Post-quantum public-key algorithms take another route: they run on classical computers and rely on computational problems believed to resist the quantum algorithms known to us. These are related research areas, but they do not provide interchangeable system components.

The practical question is therefore not whether a product carries the word quantum. It is which task an attacker can perform, with which inputs and resources, against which construction. Before studying the attacks, we need to see how a quantum computation can turn an unobservable amplitude relationship into a useful classical outcome. This provides a more reliable foundation than imagining a processor that simply reads every possible answer at once.

19.2 Gates, interference and entanglement

A gate changes a state

A **quantum gate** is a specified operation on one or more qubits. In the ideal circuit model, a unitary gate preserves the total probability and has an inverse. Unitary is the mathematical condition that provides these properties. Measurement and reset are also useful operations, but they are not reversible unitary gates on the measured qubit alone. A complete program may combine all these operations with classical processing.

For one qubit, the **X gate** exchanges

$$|0\rangle$$

and

$$|1\rangle$$

. If the input is

$$\alpha|0\rangle + \beta|1\rangle$$

, the output is

$$\beta|0\rangle + \alpha|1\rangle$$

. Here

α

and

β

retain their earlier meaning as amplitudes. Exchanging them twice restores the original state. The **Z gate** instead leaves the amplitude of

$|0\rangle$

unchanged and reverses the sign of the amplitude of

$|1\rangle$

. Its output is

$$\alpha|0\rangle - \beta|1\rangle$$

. Repeating it also restores the original state, since changing a sign twice has no net effect.

The **Hadamard gate**, written

H

, mixes the two amplitudes in a way that makes interference visible:

$$(19.2)$$

The column on the right lists the input amplitudes in the order

$|0\rangle, |1\rangle$

, and the column on the left lists the output amplitudes in that same order. The prime mark means output, rather than differentiation. The square array is a **matrix**: each row tells us how to combine the input values. Multiply across the first row and add to obtain

$$\alpha + \beta$$

, then divide by

$$\sqrt{2}$$

. The second row gives

$$\alpha - \beta$$

, with the same divisor. The square root

$$\sqrt{2}$$

is the positive number whose square is 2.

Starting from

$$|0\rangle$$

means

$$\alpha = 1$$

and

$$\beta = 0$$

. The gate therefore gives amplitudes

$$1/\sqrt{2}$$

and

$$1/\sqrt{2}$$

. This state is called

$$|+\rangle$$

, pronounced plus. An immediate standard-basis measurement has probabilities

$$1/2$$

and

$$1/2$$

. Starting instead from

$$|1\rangle$$

, with amplitudes 0 and 1, gives

$$1/\sqrt{2}$$

and

$$-1/\sqrt{2}$$

. This is

$$|-\rangle$$

, pronounced minus. It has the same probabilities for that immediate measurement, but different behavior under another gate.

Turning a phase difference into an outcome

Apply
H

again to the plus state. Its first output amplitude is

$$1/\sqrt{2}$$

. Its second is

$$1/\sqrt{2}$$

. The result is exactly

$$|0\rangle$$

. The second gate has not made another independent coin toss. It has combined amplitudes that still have a definite relationship. The two contributions reinforce in one output and cancel in the other. Starting with the minus state gives the opposite result: the first output cancels and the second becomes 1, so

$$H|-\rangle=|1\rangle$$

.

Now compare two experiments that both begin in

$$|0\rangle$$

. In the first, apply

$$H$$

and then

$$H$$

; the final ideal outcome is 0. In the second, insert

$$Z$$

between the two Hadamard gates. The first

$$H$$

makes the plus state,

$$Z$$

changes it to the minus state, and the final

$$H$$

gives

$$|1\rangle$$

. Immediately before the final

$$H$$

, an ordinary measurement would give equal probabilities in both experiments. The final gate converts their different relative signs into different observable results.

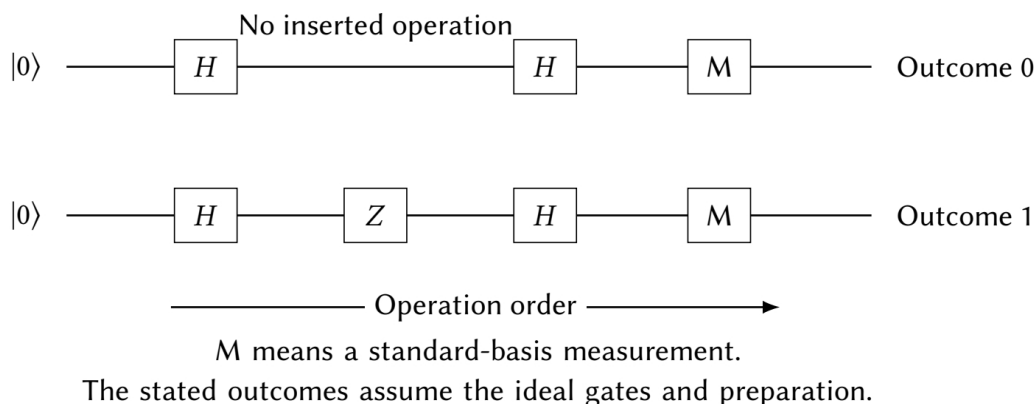


Figure 19.1. Two original one-qubit teaching circuits. Operations run from left to right. The inserted gate changes a relative sign, and the final Hadamard converts that difference into an ideal measurement outcome.

The two routes use the same preparation and measurement basis. Their different outcomes arise from the operation in the middle, not from a different interpretation of the displayed 0 and 1. This controlled comparison demonstrates why retaining phase relationships matters. Noise, an unintended measurement or loss of that relationship can change the prediction.

A **relative phase** is a phase relationship between components of the state; the sign change above is one simple case. Multiplying every amplitude by the same factor of magnitude one is instead a global phase. It does not change the measurable predictions of the state. A minus sign applied to both components is an easy example: later linear gates carry it through the whole state, and taking squared magnitudes removes it from every outcome probability. Changing just one component can affect the later interference, as the two circuits show.

Measurement is part of the experiment's definition. A standard-basis measurement asks for a 0 or 1 outcome in the chosen basis. Applying

H

first and then measuring implements a different question about the original state. It distinguishes plus from minus in the ideal model. Neither measurement gives an unrestricted way to recover a completely unknown state from one specimen. Repeated state preparation permits several

experimental settings to be investigated, while a single unrepeatable unknown state offers different information.

Two qubits and correlated measurements

For two qubits, the standard basis contains four strings:

00
,
01
,
10
and
11

. We will write the first qubit on the left in the following mathematical explanation. A **controlled-NOT**, or CNOT, uses the first qubit as a control and flips the second, its target, when the control is 1. On basis inputs its rule is

(19.3)

The symbols

a

and

b

are bit values, each either 0 or 1, and the comma separates the two qubits. The arrow means the operation maps the input to the output. The XOR symbol

$\oplus$

gives 1 when its two bit inputs differ. Thus

00
stays
00
,
01

stays

01

,

10

becomes

11

and

11

becomes

10

. On a superposition, the gate acts on the components together according to the same linear rule, rather than first measuring the control.

Begin with

$|00\rangle$

and apply

H

to the first qubit. The state becomes

$(|00\rangle + |11\rangle)/\sqrt{2}$

. CNOT leaves the first term unchanged and maps the second to

$|11\rangle$

, giving

(19.4)

The symbol

$|\Phi^+\rangle$

names this particular **Bell state**; the superscript plus is part of its name.

Each displayed component has amplitude

$1/\sqrt{2}$

and probability

$1/2$

. The two omitted components have zero amplitude. A joint standard-basis measurement therefore returns

00

or

, each with probability one half. Either qubit considered by itself gives 0 or 1 equally often, while the two records agree.

This is an **entangled state**: it cannot be represented as one independent pure state for the first qubit multiplied by another for the second. To understand the restriction, suppose each independent qubit had nonzero amplitudes for both 0 and 1. Their product would then contain nonzero contributions for all four strings. Removing both unequal strings while retaining both equal strings is impossible with such independent pure states. The pair has a joint state that cannot be assigned as two separate pure-state descriptions.

Equal measurement records alone do not prove entanglement. Two ordinary devices can share a random classical bit and both report it, producing the same equal-string frequencies in that one basis. The Bell state differs in its behavior when the measurement settings change. For example, applying

H

to both qubits leaves this Bell state unchanged. The shared classical mixture instead gives all four strings with equal probability after those operations. Measurements in additional settings therefore distinguish relationships that a single table of 0/1 counts can hide. A formal experimental entanglement claim must account for preparation, measurement errors and the assumptions of its test.

The same distinction applies to one qubit of the Bell pair considered alone. Its local state is an equal mixture, although the joint two-qubit state is pure. It cannot be assigned a separate pure state that reproduces all its local measurement behavior. Entanglement can therefore give a subsystem a mixed state even when no one used a classical random choice to prepare the whole system.

These correlations do not permit controllable communication faster than light. Measuring one side does not allow its observer to choose a particular random outcome and send that choice to the other side. The other observer's local frequencies remain unchanged when the remote outcome is not supplied. Recognizing the joint pattern requires comparing records through an ordinary communication process. Entanglement is a resource for

specified information-processing tasks, rather than a universal shortcut around communication constraints.

An optional calculation and an open-source route

The following program performs a small ideal-state calculation on an ordinary computer. A Python list stores four amplitudes, ordered as

00, 01, 10, 11

, and the two loops implement the stated Hadamard and CNOT rules. The index

t

is the target's bit value in the first loop. The final index expression stores the first-qubit output as the more significant bit. No hardware account, network request or random sampling is required.

Listing 19.1. An original four-amplitude Bell-state calculation on a classical computer.

```
from math import sqrt, isclose

state = [1.0, 0.0, 0.0, 0.0] # 00, 01, 10, 11
after_h = [0.0] * 4
for t in (0, 1):
    a0, a1 = state[t], state[2 + t]
    after_h[t] = (a0 + a1) / sqrt(2)
    after_h[2 + t] = (a0 - a1) / sqrt(2)

after_cx = [0.0] * 4
for control in (0, 1):
    for target in (0, 1):
        old = 2 * control + target
        new = 2 * control + (control ^ target)
        after_cx[new] = after_h[old]

probabilities = [abs(a) ** 2 for a in after_cx]
expected = [0.5, 0.0, 0.0, 0.5]
if not all(isclose(p, e, abs_tol=1e-12)
           for p, e in zip(probabilities, expected)):
    raise AssertionError("Unexpected ideal probabilities")
print([round(p, 6) for p in probabilities])
```

Its printed result is

[0.5, 0.0, 0.0, 0.5]

. Squaring the magnitude of each final amplitude produces these probabilities; rounding only makes the displayed decimal list easier to read.

The check allows a tiny numerical difference because floating-point arithmetic approximates numbers such as

$$\sqrt{2}$$

. This is not a measurement of an entanglement experiment, an error-mitigation benchmark or evidence that an ordinary computer has become quantum.

For a larger software framework, **Qiskit** is an open-source project with circuit-building tools, operators and compilation passes. Its Hadamard and controlled-X source definitions expose the operations used here, while its statevector tools initialize and evolve a classical amplitude representation. A statevector is simply the ordered list of amplitudes for the chosen basis. Reading that interface provides a direct connection between this explanation and an established programming framework. [\[129\]](#)

Index conventions must still be checked. Qiskit's displayed basis strings place higher qubit indices on the left, so a circuit acting on qubit 0 need not match a handwritten matrix that treats the leftmost qubit as the control. Our program explicitly uses its own first-qubit-left convention. The equal-string Bell probabilities happen to look the same under reversing these two labels; that coincidence does not make index order irrelevant for other circuits. Correct software use begins with an exact interpretation of its inputs and outputs.

A full statevector also has a clear storage cost. For

$$n$$

qubits it contains

$$2^n$$

amplitudes, where

$$n$$

counts qubits and the exponent counts the binary basis strings. If each complex amplitude uses sixteen bytes in a chosen classical representation, thirty qubits require

bytes for that array alone. This is sixteen gibibytes, where one gibibyte means

bytes. Temporary arrays and other program data require additional storage. The calculation explains a limitation of this particular simulation method; it does not prove that every thirty-qubit experiment is classically difficult.

Some circuit families have structure that allows much more efficient classical descriptions. The requested output also matters: estimating one quantity can be easier than reconstructing every amplitude. Consequently, evidence of quantum advantage needs a defined task, accuracy requirement, resource accounting and a serious comparison with suitable classical methods. A large statevector dimension by itself is not such a comparison. Small examples remain valuable precisely because their outputs can be independently understood and checked before considering a harder task.

19.3 Quantum hardware, noise and error handling

What an IBM quantum system contains

A real quantum computer contains much more than its quantum chip. An ordinary host program prepares the circuit and decides what to measure. Control electronics turn operations into the signals supported by the device. Readout electronics turn physical responses into classical records. Calibration and control systems maintain the relationship between the intended operations and the behavior of the actual apparatus. The quantum processor is one part of this larger arrangement.

IBM's superconducting systems offer a concrete example. A superconducting circuit can provide two chosen quantum energy states that serve as the qubit's 0 and 1 reference states. Those labels are logical descriptions of selected physical states, not little printed digits stored inside a refrigerator. Control signals implement operations, and coupling arrangements enable interactions between selected qubits. A cooling system maintains the very low temperatures required by this hardware approach. Different technologies, such as trapped ions or photonic systems, use other physical resources.

IBM introduced Quantum System Two in December 2023 with three Heron processors and supporting classical/control infrastructure. The first

Heron revision had 133 qubits; later revisions changed the device. Stating the revision matters, because the number in a launch announcement need not describe every processor with the same family name. Nor does putting three processors in a system automatically make their qubits one freely connected, error-corrected register. A claim about joint computation requires an account of the actual coupling and supported operations. [\[130\]](#)

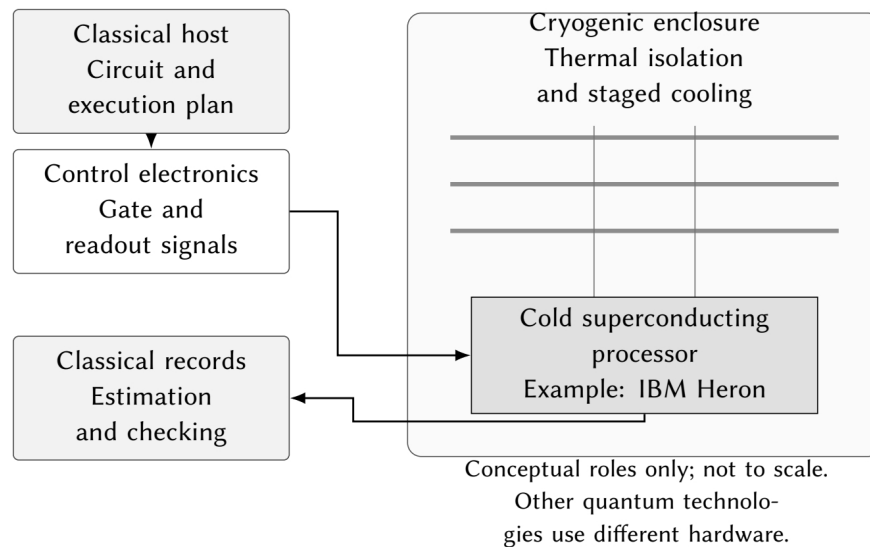


Figure 19.2. An original explanatory schematic for a superconducting quantum workflow, using IBM Quantum System Two and Heron as the hardware example. The layers show roles, not the product’s dimensions, exact wiring or a manufacturer photograph.

The cold processor needs ordinary computers and control equipment around it. Measurement records return to classical software for statistical analysis and application decisions. This makes quantum computation a coordinated physical and software process. An accurate explanation should identify which part of that process a photograph, gate count or benchmark actually describes.

Before execution, software may have to translate a circuit into the device’s native gate set and available connections. This translation is often called **transpilation**. If two qubits cannot interact directly, the chosen mapping may require additional operations or a different placement. Equivalent ideal circuits can therefore have different costs and noise exposure on a real processor. Qiskit’s circuit description is not a promise that every gate in a drawing is one equally reliable hardware instruction.

For a new reader, the sequence is useful to keep in mind: describe the desired computation, choose a physical mapping, execute, collect measurements and interpret them. Each stage can introduce a distinct problem. A correct ideal circuit can be mapped inefficiently; a carefully mapped circuit can drift away from its calibration; a good experiment can be summarized with a misleading statistic. Understanding the layers helps us avoid assigning every success or failure to the mathematical algorithm alone.

Why errors change the computation

Noise describes unwanted effects that alter preparation, operations, storage or measurement. An intended 0 may be reported as 1, an excitation may relax, or a phase relationship may become unreliable. **Decoherence** refers to the loss of quantum coherence through interactions with an environment, with consequences for the interference that a computation needs. It cannot generally be represented as merely adding an ordinary random bit to an otherwise perfect final answer.

The earlier

H

—

Z

—

H

example shows why phase matters. If the intended relative sign is lost before the final gate, the definite ideal prediction can become uncertain. A gate can also rotate a state by a slightly wrong amount in a repeatable way. Repetition of such a coherent error can behave differently from independent random mistakes. Device characterization therefore asks which errors occur and how they depend on time, operations and neighboring components.

Even a deliberately simple error model warns against reading a large gate count as a success count. Suppose

G

operations each have an independent probability

p

of a specified error event. In that model,

(19.5)

Here

P_{none}

is the probability that none of those events occurs,

G

is the number of operations and

p

is the per-operation event probability. Each factor

$1 - p$

is the chance of no event in one operation. Independence permits multiplication of those chances. For

$p = 0.001$

and

$G = 1000$

, the value is

$0.999^{1000} \approx 0.368$

, or about 36.8 percent.

This is an original teaching model, not a prediction for a Heron processor. An error event does not always change a particular measured quantity, and real errors need not be identical or independent. Some effects cancel, propagate, accumulate or correlate. The calculation's purpose is to show why a seemingly small local rate can matter across a long process. A hardware estimate needs a more appropriate noise model and the actual circuit.

Error suppression attempts to reduce unwanted effects during control or execution, for example by improving calibration, choosing a better mapping or using suitable control sequences. **Error mitigation** uses additional experiments and data processing to improve estimates of specified ideal quantities. **Quantum error correction** encodes quantum information and

uses error information to protect logical operations. These methods can complement one another, but they do not make the same claim about one individual physical run.

Measurements, estimates and Qedma's work

A **shot** is one prepared execution followed by the specified measurements. Many experiments estimate an **expectation value**: the probability-weighted average of possible measurement values. For a simple measurement, assign

+1

when the recorded bit is 0 and

−1

when it is 1. The sample estimate is

(19.6)

The hat marks an estimate,

μ

names the true mean of the experiment being sampled,

N

counts shots,

j

labels a shot and

o_j

is its assigned

+1

or

−1

value. On an ideal device, that true mean agrees with the intended ideal quantity. On a noisy device, it can differ. The summation sign means add all values from shot 1 through shot

N

, then divide by

$$N$$

. If 5,500 of 10,000 shots give 0 and 4,500 give 1, the numerator is

$$5500 - 4500 = 1000$$

, giving

$$\hat{\mu} = 0.1$$

. This number is an estimated average, not a probability of 0.1 for every possible interpretation of the experiment.

Even an ideal device produces sampling variation when outcomes are random. For independent identically prepared

$$+1/-1$$

outcomes, the variance is

$$1 - \mu^2$$

, and the standard deviation of their sample mean is

$$(19.7)$$

The symbol

$$\sigma_{\text{mean}}$$

measures the spread of repeated sample-mean estimates around their true mean. Variance is the average squared deviation; its square root gives a standard deviation in the original units. Since

$$\sigma_j^2 = 1$$

for either sign, the variance is

$$1 - \mu^2$$

. Independence makes the mean's variance smaller by a factor

$$N$$

. If

$$\mu = 0$$

, 100 shots give a standard deviation of 0.1 and 10,000 give 0.01. This is a scale of statistical variation under the model, not a guaranteed error bound for one run.

Bias is a different issue: the average estimate can differ systematically from the desired ideal quantity. The sampling-variance formula uses the true mean of the sampled outcomes, not an ideal target substituted for that

mean. More repetitions reduce ordinary sampling variation, but cannot by themselves remove a persistent bias from hardware or an incorrect model. A useful error-handling method must therefore state the quantity it estimates, how noise is treated and what uncertainty remains. A narrow-looking error bar can still be misleading if systematic effects were omitted.

Qedma develops software addressing this error-handling layer. Its QESEM framework combines characterization, circuit/control choices and statistical mitigation. Professor Netanel H. Lindner, a Technion physicist, is a co-founder and the company's Chief Technology Officer; Professor Dorit Aharonov is a co-founder and Chief Scientific Officer, alongside co-founder Asif Sinay. These affiliations identify who contributes to the work. The scientific evidence comes from the described methods, assumptions and experiments rather than from the job titles. [\[132\]](#)

The public QESEM research manuscript describes quasi-probabilistic mitigation, in which outputs from chosen executable circuits receive classical weights. Some weights can be negative. They are accounting coefficients in an estimator, not negative probabilities for a physical measurement. A useful schematic expression is

(19.8)

Here

μ_{ideal}

is the desired ideal mean,

ℓ

labels a component experiment,

μ_ℓ

is that experiment's mean and

w_ℓ

is its specified real weight. The sum means multiply each component mean by its weight and add the products. Such an equality needs the construction's mathematical and noise-model conditions. It is not an instruction to choose arbitrary weights until a result looks right.

For an original arithmetic illustration, suppose an exact model required weights 1.2 and

-0.2

, with component means 0.7 and 0.4. The combination is

$(2/3) - (1/3) = 1/3$

. These invented numbers are not a reported Qedma measurement or an implementation recipe. Independent estimates of those means bring variances that combine with the squares of their weights. Large absolute weights can therefore make accurate estimation expensive even when the modeled combination removes bias.

We can calculate that statistical cost in the same invented two-component setting. If each component has independent

$+1/-1$

outcomes, and the component sample means are independent, the variance of the weighted estimate is

$$(19.9)$$

The function

Var

means variance. The subscripts 1 and 2 identify the two component experiments,

w_1, w_2

are their weights,

μ_1, μ_2

their true means and

N_1, N_2

their shot counts. Each fraction is the variance of one component sample mean, as in Equation [19.7](#). Squaring its weight gives that component's contribution to the combined variance. Independence removes covariance terms; correlated data require those additional terms.

Using the values above and 10,000 shots for each component gives

$$\begin{aligned}
& [1.2^2 (1 - 0.7^2) \\
& + (-0.2)^2 (1 \\
& - 0.4^2)] / 10000 \\
& = [1.44 \times 0.51 \\
& + 0.04 \times 0.84] / 10000 \\
& = 0.0000768
\end{aligned}$$

Its square root is approximately 0.00876, the standard deviation of the combined estimate under this exact model. The true combined mean remains 0.76 even though finite estimates fluctuate. This worked calculation separates unbiasedness from finite precision and makes the extra resource question visible.

To halve a sampling standard deviation while keeping the same distributions and allocation proportions, the shot counts must be multiplied by four, because the deviation scales as the inverse square root of the sample size. Device time must also include state preparation, execution, readout and characterization. Qedma's practical documentation correspondingly treats circuit structure, measured quantities, target precision and hardware quality as inputs to its runtime assessment. A

comparison that reports only the final accuracy while omitting repetitions and calibration cost is incomplete. [\[136\]](#)

An **unbiased estimator** has the desired value as its expectation under the stated model. It does not return an exact answer in every finite experiment. Calibration errors, changes in hardware noise and approximations can also affect whether the assumptions match execution. The QESEM manuscript separates unbiased configurations from options that trade controlled bias for lower cost, and examines characterization and drift. Its reported physics and molecular examples demonstrate selected workflows, not every possible algorithm or processor. [\[134\]](#)

This contribution matters because a quantum computation is useful only when its result can be interpreted with defensible accuracy and resources. IBM's Qiskit Functions documentation exposes QESEM as a circuit service, connecting such methods to a practical execution route. Availability as a service still differs from a result reproduced for this book. We have inspected the public descriptions and research; we have not submitted a paid hardware job or independently repeated the company's benchmarks. [\[136\]](#)

Research also establishes limits. Ryuji Takagi, Suguru Endo, Shintaro Minagawa and Mile Gu derived lower bounds on mitigation costs under a defined framework, connecting lost state distinguishability to required statistical resources. The result is a reason to examine scaling assumptions when a method is described as efficient. It does not say that mitigation has no value, nor that a guarantee for one circuit and quantity is a universal bound for every computation. [\[135\]](#)

Logical information and fault tolerance

A **physical qubit** is a physical system used to carry quantum information. A **logical qubit** is an encoded unit protected through an error-correction scheme. The resources can include many physical components, repeated error checks and additional operations. A bare physical-qubit count therefore does not establish that a machine can complete an attack on a specified RSA key. Reliable long computations require an account of what the logical operations actually achieve.

At first, redundancy may sound as simple as copying an unknown qubit several times. The **no-cloning theorem** rules out a universal operation that perfectly copies every unknown quantum state while preserving the original. A simple way to see the difficulty is to consider a machine that copies basis states. Writing two kets next to one another means preparing their joint product state. Linearity would take

$$(00 + 11)/\sqrt{2}$$

to

$$(00 + 11)/\sqrt{2}$$

. Two independent plus-state copies would instead contain all four strings, each with amplitude

$$1/2$$

. The intended universal copy and the actual linear output are different states. [\[128\]](#)

Quantum error correction uses an encoded joint state rather than a set of independent copies. An **error syndrome** supplies information about specified error patterns without directly measuring the protected logical value. A **decoder** interprets that information and selects or tracks a correction according to the code and error model. Syndrome extraction, control and decoding can themselves be imperfect, so a full construction has to manage faults throughout this process.

A three-qubit repetition example illustrates one restricted case:

$$(19.10)$$

The amplitudes

$$\alpha, \beta$$

describe the original logical information. The three positions on the right are physical qubits encoding that one logical qubit. If a single X error changes the middle position, the state becomes

$$\alpha(000) + \beta(101)$$

. The first two bits disagree and the last two bits disagree in both components. Those two parity relations identify the middle position without revealing whether the logical component was 0 or 1. Flipping that position restores the encoded state.

This code corrects the specified single-bit-flip case, not arbitrary quantum noise. A Z error on one encoded position can change the relative sign of the two logical components while leaving their bitwise parity relations unchanged. More complete codes must address phase as well as bit effects and specify their fault assumptions. The restricted example explains what a syndrome accomplishes without pretending that three qubits solve the entire engineering problem.

Fault tolerance concerns performing the computation so that manageable physical faults do not spread into unmanageable logical errors. Threshold results, including the work of Dorit Aharonov and Michael Ben-Or, show that sufficiently reliable components under specified noise and control assumptions can support scalable reliable computation with overhead. A threshold is a conditional boundary in a model. It is not one universal percentage applying to every code, architecture, leakage process and algorithm. [\[131\]](#)

Lindner's research provides a further connection to physical architecture. With Terry Rudolph, he proposed a controlled source of photonic cluster states, in which emitted photons form an entangled resource for quantum information processing. A cluster state is a structured many-qubit entangled state used in specified computational schemes. Their 2009 paper studies a source proposal and its error properties; it is separate from demonstrating a large fault-tolerant computer or QESM's later software benchmarks. The distinction shows how one researcher's work can connect several layers without making every layer the same achievement. [\[133\]](#)

A theoretical algorithmic step is consequently not necessarily one second or one physical hardware instruction. A cryptographic attack needs reliable logical operations, suitable memory, connectivity and mathematical subroutines at the target scale. Progress in control, mitigation, correction and physical architecture is important evidence about those resources. A result about improved expectation values in one physics simulation does not itself supply a completed fault-tolerant implementation of Shor's algorithm. With these boundaries in place, we can examine what the algorithm would make possible.

19.4 Shor and public-key problems

Shor's work gives polynomial-time quantum algorithms for integer factorization and discrete logarithms in the relevant mathematical settings. Polynomial time means that the algorithm's abstract resource growth is bounded by a polynomial in the input length, under its computational model. It does not mean that every key can be broken instantly, or that a noisy device with a small number of operations already meets the model. The result nevertheless changes the long-term assumptions behind major public-key constructions. [\[116\]](#)

RSA uses a public modulus whose prime factorization gives a route to the private operation. Classical Diffie–Hellman, ElGamal and elliptic-curve constructions depend on related discrete-logarithm assumptions in their chosen groups. A sufficiently capable implementation of the relevant quantum algorithms would undermine those assumptions. Increasing an RSA key length does not change the problem into a post-quantum one, even though it can change the resources needed to attack a particular instance.

We can see part of the factoring idea with the small integer 15. Choose the base

$$a = 2$$

, which has no common factor with 15. The exponent

$$x$$

counts how many factors of 2 are multiplied, with the initial value

$$2^0 = 1$$

. Consider the powers of 2 reduced modulo 15:

The values return to 1 after four multiplications, so the **order** of 2 modulo 15 is

$$r = 4$$

: the smallest positive exponent satisfying

$$2^r \equiv 1 \pmod{15}$$

. The order describes a period in the modular calculation. We found it here by listing small powers; this is a classical teaching calculation, not a quantum factoring experiment.

When a suitable even order

r

is found for a base

a

modulo an integer

N

, form

(19.11)

Here

N

is the number being factored,

a

is the selected coprime base,

r

is its order and

gcd

means greatest common divisor. The notation

$r/2$

requires an even order. The two expressions can reveal nontrivial factors when the order and base satisfy the necessary conditions. If a base already has a common factor with

N

, an ordinary gcd calculation has found useful information before this step.

For

$N = 15$

,

$a = 2$

and

$r = 4$

, we have

$a^{r/2} = 2^2 = 4$

. The first expression is

$$\gcd(4 - 1, 15) = \gcd(3, 15) = 3$$

The second is

$$\gcd(4 + 1, 15) = \gcd(5, 15) = 5$$

Multiplying the results recovers

. Both factors are proper: neither is 1 nor the whole input 15.

Why should these expressions help? Since

$a^r - 1$
is divisible by
 N

, it can be written as

. In useful cases, factors of
 N

are distributed between those two terms. The gcd calculations extract the shared factors. The conditions cannot be omitted: an odd order gives no integer

$r/2$
, and some even-order choices produce only trivial gcd results. The procedure can need another choice or another run.

The difficult subroutine at cryptographic sizes is not performing the final gcd once the relevant order is known. Shor's quantum construction obtains information about periods through operations including modular exponentiation and a quantum Fourier transform, followed by classical processing. A Fourier transform exposes periodic structure in a mathematical representation; the quantum version is part of arranging amplitudes so that measurements carry information about that period. Recovering the order still requires appropriate interpretation and success conditions.

The order-four example lets us make that route more concrete. Imagine an ideal exponent register whose possible labels are 0 through 15. A second register receives the remainder

$$2^x \bmod 15$$

while preserving the first register and the joint quantum relationships. Among those sixteen exponents, the remainder is 1 at

$$x = 0, 4, 8, 12$$

. If the second register is measured and the result is 1, the exponent register is left in an equal superposition of those four labels. This conditional example illustrates a periodic state; it does not require the experimenter to observe all the exponents at once.

An appropriate quantum Fourier transform of that ideal sixteen-label state has support at labels

$$y = 0, 4, 8, 12$$

. Here

$$y$$

names the measured output label. Dividing by the register size gives fractions

$$y/16$$

of

$$0, 1/4, 1/2, 3/4$$

. The separation between useful Fourier labels reflects the inverse of the original four-step spacing. For example, the fraction

$$4/16 = 1/4$$

exposes the denominator 4, which can be tested as a candidate order by calculating

$$2^t \bmod 15$$

.

Some outcomes carry less information. The fraction

$$8/16 = 1/2$$

has denominator 2 after reduction, but

$$2^t \bmod 15 = 1$$

, not 1. It therefore does not establish that 2 is the order. The zero outcome is also unhelpful for this recovery. Repeated runs and classical checks address such cases. For general inputs, the register size need not align exactly with the period, so the measured fractions approximate rather than exactly equal the useful ratios. Classical continued-fraction processing, a method for finding simple rational approximations, enters the recovery. The small aligned example explains the mechanism without replacing that full analysis.

The modular calculation also has to be implemented reversibly inside the circuit. Intermediate work cannot be discarded arbitrarily while retaining the ideal coherent transformation. Additional qubits may hold working values, and operations can be reversed to clear temporary information when the construction requires it. This is one reason why the bit length of the public modulus is not a complete hardware resource count. The quantum arithmetic, logical protection and successful order recovery all contribute to the cost.

The consequence extends beyond decrypting one RSA ciphertext. A recovered private signing key can support forged signatures, while a solved agreement problem can reveal secret material from a recorded exchange. The exact consequence depends on the protocol and which public data were retained. Calling every effect a password attack would confuse the mathematical capability with unrelated account controls.

Forward secrecy also needs its original scope. In the TLS discussion, fresh ephemeral agreement helped protect earlier sessions against later disclosure of a long-term authentication key. That did not assert that the recorded ephemeral public shares remain hard to solve after a new algorithm defeats their mathematical problem. A sufficiently capable future quantum attack on the recorded agreement can therefore raise a different question from later theft of the server's signing key.

None of this establishes a date for a practical attack against a chosen deployed parameter. Shor's result identifies a fundamental algorithmic threat under a quantum computational model. Engineering estimates then ask what is needed to realize it at a particular scale. Treating the theorem as irrelevant because deployment is uncertain, or treating deployment as already complete because the theorem exists, would discard one side of the reasoning.

19.5 Grover and generic search

Grover's search algorithm provides a different kind of advantage. In an idealized search with

N

candidates and an appropriate way to recognize a solution, it uses on the order of

$\sqrt{N}$

oracle queries for a single marked solution. An **oracle query** here means an invocation of the specified recognition operation inside the quantum computation. It is not necessarily one call to an ordinary web service. Constructing that coherent operation has a cost of its own. [\[117\]](#)

For a search over every

k

-bit key, there are

$N = 2^k$

candidate strings. The corresponding idealized scale is

(19.12)

The number

k

counts key bits,

N

counts possible keys and the square root is the nonnegative number whose square gives

$$N$$

. Dividing the exponent by two explains the familiar search comparison. The equality is arithmetic; applying it to a concrete attack also requires the algorithm's model, a suitable checking circuit and a clear success condition.

With

$$k = 128$$

, the expression gives

$$2^{64}$$

as the ideal query scale. With

$$k = 256$$

, it gives

$$2^{128}$$

. These are not forecasts in seconds or complete hardware budgets. A reversible implementation of a candidate test may require many quantum operations and additional storage, with repeated error correction during the computation. A search argument must also ensure that the available known data distinguish the desired key sufficiently well.

The improvement is quadratic rather than the polynomial-time change Shor supplies for the public-key problems discussed above. A quadratic improvement can be serious without making every search practical. It also means that the response for symmetric cryptography need not be identical to replacing an RSA or elliptic-curve construction. Parameter choices and the whole protocol have to be assessed for their actual tasks.

A quantum search cannot generally be divided among many processors while retaining the same simple linear speedup assumed for ordinary independent guesses. Resource trade-offs depend on how the search is organized and which machine resources are being counted. Nor can the internal recognition circuit automatically query a rate-limited online password service in superposition. An offline key-search model, a remote login service and a password-derived encryption key are distinct settings.

We can see one ideal search step without a large register. Let four basis labels represent four candidates, with

$$11$$

the single marked candidate. An equal superposition gives amplitude

$$1/2$$

to each label, so each has probability

$$1/4$$

. The oracle changes the sign of the marked amplitude, producing the list

$$0/3/3/1/1-1/3$$

. This step has not yet increased its measurement probability: a negative amplitude of magnitude

$$1/2$$

still contributes

$$1/4$$

. Another operation must convert the sign pattern into a useful probability change.

For this small example, the next operation reflects each amplitude about the mean amplitude:

$$(19.13)$$

The symbol

$$a_j$$

is the amplitude of basis label number

$$j$$

, where

$$j = 0, 1, 2, 3$$

corresponds to

$$00, 01, 10, 11$$

. The bar marks the arithmetic mean, and

$$a'_j$$

is the new amplitude. Add the four amplitudes after the oracle and divide by 4 to obtain

1 1 1 1 1 1 1 1 1

[illegible]

are not automatically beneficial, because the state continues to evolve rather than remaining permanently concentrated on the answer.

The marking operation need not reveal the answer to the person designing the search. In key search, a candidate can be tested against available public information using the specified reversible computation. Knowing how to recognize a valid candidate differs from already knowing which candidate will satisfy the test. However, that recognizer still needs an exact definition and resources. If many candidates fit the available data, the single-marked-solution calculation no longer describes the experiment without adjustment.

This also explains a limit of the phrase try every key simultaneously. The amplitudes before marking already involve every candidate, but an immediate measurement would still be a random guess. The algorithm gains its advantage by using a structured sequence of marking and interference to change the output distribution. The useful result is a sampled candidate that can then be checked classically; it is not a returned spreadsheet containing the result of every candidate evaluation.

Hash functions require the same care. Finding an input matching a specified digest is a preimage task, while finding any two matching inputs is a collision task. Chapter [8](#) explained why their classical costs already differ. A generic quantum search statement for one target should not be copied as the cost of every collision problem, signature forgery or protocol attack. The construction's use of the hash and the permitted form of quantum access also matter.

The useful conclusion is not that quantum computers break all cryptography equally. Shor and Grover exploit different structures and supply different resource bounds. Symmetric encryption, hashing, public-key agreement and signatures therefore need task-specific analysis. This distinction helps us understand why post-quantum standards introduce new public-key mechanisms while still relying on carefully chosen symmetric and hash operations inside the constructions.

19.6 A first picture of lattices and LWE

A **lattice** is a set of points formed by integer combinations of independent basis vectors. A vector can be understood as an ordered list of coordinates or a directed displacement. In a two-dimensional example, take

$$b_1 = (2, 0)$$

and

$$b_2 = (1, 2)$$

. The lattice contains every point of the form

$$(19.14)$$

The bold symbols name the two basis vectors, and

$$\mathbb{Z}$$

means all integers, including zero and negative values. Thus

$$u$$

and

$$v$$

may move along either direction of a basis vector, but may not take arbitrary fractional values. Taking

$$u = 2$$

and

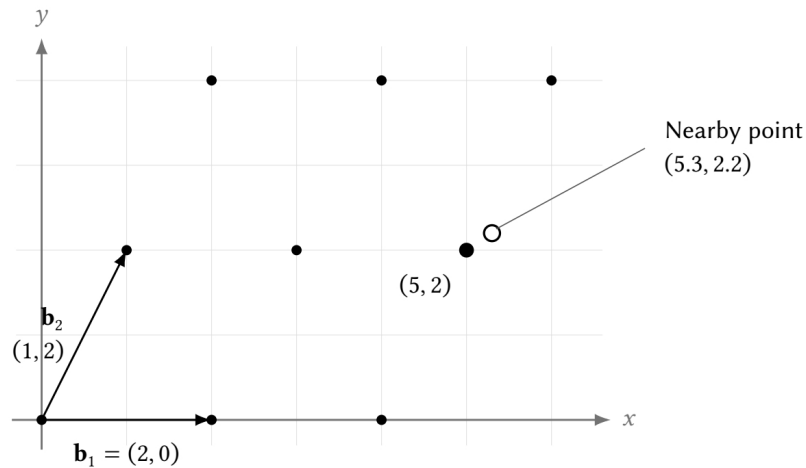
$$v = 1$$

gives

$$b_1 + b_2 = (3, 2)$$

.

In a small picture, finding a nearby lattice point can be easy. Higher-dimensional problems with carefully selected parameters can have very different computational behavior. The drawing is useful for introducing structure and distance, but it does not measure the difficulty of an actual post-quantum instance. Modern constructions also use algebraic structure and specific distributions; selecting a complicated-looking grid is not a cryptographic design method.



Only a finite window is drawn.
The lattice continues through integer combinations in both directions.

Figure 19.3. An original two-dimensional lattice illustration. Black points are integer combinations of the two basis vectors; the open marker is a nearby point outside the lattice. The small geometry teaches the vocabulary, not the hardness of a cryptographic parameter.

The point

$(5, 2)$

has a short description in this basis, while

$(5.3, 2.2)$

is displaced from it. A geometric problem might ask for a nearby lattice point given a different representation or much higher-dimensional data. Which problem is posed and how instances are generated are part of a hardness claim. A picture that lets us identify the nearest point by eye should not be described as a secure lattice encryption scheme.

Learning with errors, abbreviated LWE, begins from linear relations with deliberately added small errors. A **linear relation** combines unknown values using known coefficients and addition. Without errors, enough suitable independent equations modulo a prime can often be solved efficiently by elimination. Adding noise changes the inference task: the observer must find a secret consistent with many approximately satisfied modular equations, under the specified distributions and dimensions. [\[118\]](#)

For a small teaching example, let the modulus be

$q = 17$

and the secret be the pair

$$s=(3,5)$$

. Each published row contains two coefficients

$$a_1, a_2$$

and a result

$$b$$

. Its generation rule is

$$(19.15)$$

Here

$$s_1$$

and

$$s_2$$

are the secret coordinates,

$$a_1$$

and

$$a_2$$

are public coefficients, and

$$e$$

is a small signed error. The remainder modulo

$$q$$

is represented between 0 and

$$q-1$$

. In this example only, the possible errors are

$$-1$$

, 0 and 1. Real schemes specify their own distributions and much larger parameters.

Table 19.2. Four original teaching rows generated with secret

modulo 17. The error column is disclosed here to explain generation; an LWE observer is not ordinarily given those errors or the secret.

—

Teaching error	Published result
1	14
Teaching error	Published result
4	16
Teaching error	Published result
3	5
Teaching error	Published result
5	2

The first row gives

$$3 \times 5 + 2 \times 4 + 1 \times 14 = 39$$

. The second gives

$$8 \times 5 + 3 \times 4 + 4 \times 16 = 108$$

. The third produces 39, whose remainder modulo 17 is 5, since

$$39 - 2 \times 17 = 5$$

. The fourth produces 36, whose remainder is 2. These steps show how one secret creates several noisy relations without claiming that four tiny equations provide useful security.

Ignoring the errors can give the wrong secret. If the first two published rows are treated as exact equations, they become

$$a_1 + 2a_2 = 14$$

and

$$4a_1 + a_2 = 16$$

modulo 17. Multiply the first equation by 4 and subtract the second. This gives

$$7a_1 = 40 \pmod{17}$$

. The inverse of 7 is 5, because

$$7 \times 5 = 35 \pmod{17}$$

. Therefore

$$a_1 \equiv 5 \times 8 \equiv 13$$

, and the first equation gives

. The resulting pair

(5, 13)

differs from the actual secret

(3, 5)

. It predicts

for the third row, while the published result is 5. The error is not ordinary measurement imprecision to be removed from one isolated row; it is deliberately part of the problem's generation rule.

Testing a candidate secret also requires interpreting small errors across the modular boundary. A remainder of 16 modulo 17 can represent the small signed value

−1

. For this example, define a centered representative

(19.16)

Here

x

is the integer whose signed representative we want, and

center_{17}

names the centering operation for modulus 17. The result lies between

−8

and 8. Add 8, take the remainder from 0 through 16, then subtract 8. For

$x = 16$

, the expression gives

$$24 \bmod 17 - 8 = 7 - 8 \\ = -1$$

The centered residual

can therefore be checked against the allowed signed errors.

The following optional calculation tries all

$17^2 = 289$

possible secret pairs. It retains those for which every residual is between

-1

and 1. The output is the single pair

$(3, 5)$

. The example is intentionally small enough to solve completely and discloses all generation data for learning. It is not an LWE security demonstration or an implementation of ML-KEM.

Listing 19.2. Recovering the deliberately tiny noisy-equation example by exhaustive search.

```
q = 17
rows = [(1, 2, 14), (4, 1, 16),
        (3, 6, 5), (5, 4, 2)]

def centered(value):
    return (value + q // 2) % q - q // 2

possible = []
for s1 in range(q):
    for s2 in range(q):
        errors = [
            centered(b - a1 * s1 - a2 * s2)
            for a1, a2, b in rows
        ]
        if all(abs(error) <= 1 for error in errors):
            possible.append((s1, s2))

if possible != [(3, 5)]:
    raise AssertionError("Unexpected teaching candidates")
print("Teaching candidates:", possible)
```

The matrix form collects many rows into one statement,

Regev et al. 2002
. The matrix

A

stores public coefficients,

s

is the secret vector,

e

contains the errors and

b

contains the published results. Multiplication by

A

performs the row calculations just shown. The notation saves space; it does not change the roles of the inputs or supply a hardness argument by itself.

Regev's foundational work relates suitable LWE settings to worst-case lattice problems through a reduction whose original form uses quantum computation. A **reduction** connects the ability to solve one problem to the ability to solve another under stated parameters and conditions. It is evidence about a relationship between problems, not a proof that no efficient algorithm can ever solve either one. The quantum nature of the reduction is also not a requirement for ordinary users of the associated classical cryptosystem.

Module-LWE adds structured algebra that can make representations and operations more efficient. Polynomial coefficients and modules replace the plain scalar objects of the first example. That structure has to be included in the security analysis; the theorem for an arbitrary unstructured LWE instance cannot simply be declared a proof for every module-based scheme. The choice of dimension, modulus, errors, encoding and transformations is essential. Noise alone is not a universal source of security.

A polynomial is an expression built from coefficients and powers of a formal symbol. In a small invented example,

$$3 + 2X + X^2$$

stores the coefficient list

$(3, 2, 1)$

; the capital

X

here is a formal placeholder, unrelated to the earlier *X* quantum gate. Specified polynomial arithmetic combines such coefficient lists, usually reducing both coefficients and powers according to stated rules. It can package many linked arithmetic operations compactly. This is a classical representation: the word quantum in post-quantum cryptography does not imply that the coefficients are stored as qubit amplitudes.

There are also distinct LWE questions. In a search formulation, the task is to recover the secret from the sampled relations. In a decision formulation, the task is to distinguish the specified noisy distribution from another specified distribution, often an appropriate uniform one. A reduction can connect such problems under conditions, but the tasks should not be treated as identical by definition. Our tiny program addresses a disclosed bounded-error search example. Its success says nothing about a distinguisher's advantage at a standardized parameter.

19.7 ML-KEM, ML-DSA and hash-based signatures

NIST's 2024 standards distinguish three main roles in this discussion. FIPS 203 specifies ML-KEM, a module-lattice-based key-encapsulation mechanism. FIPS 204 specifies ML-DSA, a module-lattice-based digital signature scheme. FIPS 205 specifies SLH-DSA, a stateless hash-based signature scheme. They do not form three interchangeable versions of one encryption function. Their interfaces answer different questions, as the key-agreement and signature chapters already established. [\[119\]](#)[\[120\]](#)[\[121\]](#)

ML-KEM descends from the CRYSTALS-Kyber research line. The 2018 IEEE paper is useful for understanding the design motivation and its chosen-ciphertext-secure KEM construction. The standardized mechanism is named ML-KEM and has its own finalized definitions. An older Kyber implementation or paper should not silently replace FIPS 203's parameters, input checks and encodings. Historical development explains the lineage; the applicable standard defines the mechanism being used. [\[122\]](#)

The KEM interface can be written in the familiar form

(19.17)

The encapsulation key
 ek
is public, while
 dk
is the corresponding private decapsulation key. Encapsulation produces a
shared secret

K
and a ciphertext

C
that can travel over the public channel. The holder of
 dk
processes

C
to obtain
 $\widehat{K}$

. For ML-KEM, these secrets agree with overwhelming probability for honestly generated matching inputs under the specified conditions. A very small permitted probability of different results remains even without corruption or interference. This is a correctness-failure probability, not a claim that every honest input gives exact agreement. FIPS 203 distinguishes that event from the algorithm returning an explicit failure indication. The arrow describes the output of an operation; randomness and the scheme's required checks are part of that operation. [\[119\]](#)

The KEM does not take an arbitrary book chapter as its plaintext and return that chapter on decapsulation. It establishes key material to be used within a specified construction, commonly with a KDF and authenticated encryption. It also does not authenticate the identity associated with an untrusted supplied public key merely because a shared secret was obtained. The protocol needs the identity and context bindings described in the earlier chapters. [\[54\]](#)

ML-KEM includes **implicit rejection** for particular decapsulation failures. For a ciphertext of the required size that fails the internal consistency condition, the mechanism derives replacement secret material

rather than returning an ordinary success/failure flag for that condition. This does not authorize accepting an application message under a mismatched key. Subsequent authenticated processing and the protocol's checks still determine acceptance. It also does not remove required input validation: an incorrectly sized input is a different interface case.

FIPS 203 defines ML-KEM-512, ML-KEM-768 and ML-KEM-1024. The names identify parameter sets; the suffix is not a declaration of that many security bits or the public-key byte length. For example, ML-KEM-768 uses a public encapsulation key of 1,184 bytes, a ciphertext of 1,088 bytes and a shared secret of 32 bytes. The last quantity is 256 bits because

32 × 8 = 256

, but the length alone is not a proof of 256-bit security for the entire mechanism.

The noisy-equation idea helps explain why a real construction needs several transformations. Intended parties must obtain compatible secret material even though the underlying arithmetic introduces controlled error. Encodings and decoding rules have to preserve that agreement for the construction's valid inputs. At the same time, public data must not make the secret or an equivalent attack capability efficiently recoverable. The final KEM adds the specified hashing, re-encryption checks and rejection behavior around its mathematical core. A secure argument concerns that complete construction, rather than one noisy equation in isolation.

NIST's security categories provide comparison targets against defined reference attacks; they are not simple measured lifetimes in years. Classical and quantum costs, time and memory, success probabilities and the underlying models need interpretation. Likewise, a long public key can reflect the construction's representation without implying a proportionally larger security level. The standard's parameter names, serialized lengths and security claims describe different properties and should be read separately.

ML-DSA descends from CRYSTALS-Dilithium. Its signing key authorizes generation of a signature, while its public key allows verification of that evidence against the message and required context. The 2018 Dilithium paper explains the earlier research design; FIPS 204 governs the standardized ML-DSA functions. Its computational assumptions include module learning with errors and a specified variant of the module short-

integer-solution problem. Those are not interchangeable with factoring an RSA modulus. [\[123\]](#)[\[120\]](#)

The short-integer-solution idea asks for a nonzero short vector satisfying specified modular linear relations. The zero vector would trivially satisfy a homogeneous equation, so requiring a nonzero answer matters. Requiring short coordinates matters too: removing that restriction changes the problem. This is only an introduction to the kind of assumption involved. The full ML-DSA construction, its distributions and its precise assumption variants cannot be inferred from this one verbal description.

A signature still provides evidence relative to a public key. Replacing ECDSA with ML-DSA does not independently establish who controls that key, whether a software update is authorized for the device or whether a valid signed object is an obsolete version. Post-quantum signature deployment must carry forward the certificate, trust-anchor, context and policy decisions from Chapter [13](#). A new primitive does not erase those boundaries.

SLH-DSA follows the SPHINCS+ research line and builds signatures using hash-based structures. A **Merkle tree** combines hashes upward through a tree so that a root can commit to many underlying values. A verification path supplies the neighboring information needed to reconstruct the relevant root. SLH-DSA combines such structures with other hash-based components; it is not just the rule hash the message and publish the digest. Its security assumptions and parameter choices concern the full signature construction.

The word **stateless** distinguishes its signing interface from schemes that require the signer to maintain a changing per-signature usage state to avoid reusing a one-time signing capability. It does not mean that private keys need no protection, that randomness rules disappear or that an unlimited number of signatures has an unconditional guarantee. FIPS 205 states parameter and usage conditions. Hash-based signatures offer a different design foundation, with trade-offs in signature size and computational work.

Table 19.3. The standardized mechanisms have different tasks. The output column identifies what travels or is checked, rather than ranking the algorithms by one common byte count.

Mechanis	Main role	Important output and next step
----------	-----------	--------------------------------

m ML-KEM	Establish shared secret material with the intended recipient	A KEM ciphertext is sent; the resulting secret feeds specified key derivation and protection.
Mechanis m ML-DSA	Main role Authenticate a message relative to a signing key	Important output and next step A signature is checked with a trusted public-key association and required context.
Mechanis m SLH-DSA	Main role Provide hash-based signature evidence	Important output and next step A signature is checked under the selected hash-based parameter set and trusted key.

For scale, FIPS 204 lists a 1,952-byte public key and a 3,309-byte signature for ML-DSA-65. These are different objects from ML-KEM-768’s 1,088-byte ciphertext and 32-byte secret. Choosing the smallest number in isolation would compare unlike operations. A deployment must measure the full exchange, including credentials, protocol framing, verification cost and the required security profile.

19.8 Standards, experiments and crypto-agility

A **standardized algorithm** has an identified specification. An implementation experiment explores behavior in a particular environment. A validated module has undergone a specific validation process and configuration assessment. A deployed protocol combines primitives with identity, negotiation and application rules. These descriptions can overlap, but none can substitute automatically for the others. A program returning matching ML-KEM secrets is not, by that observation alone, a validated module or a complete authenticated connection.

The standards also have maintenance histories. The official FIPS 203 and FIPS 204 publication pages identify potential updates and errata, while retaining their final-publication identities. Implementers need the applicable specification and official corrections, together with their protocol profile. An old draft, a research parameter set and a final standard can share ancestry without having identical bytes. This chapter’s examples explain concepts and roles; they are not replacement specifications for an implementation. [\[124\]](#)

The collected liboqs source makes one useful distinction concrete. Its selected ML-KEM-768 wrapper identifies FIPS 203 and supplies separate key-generation, encapsulation and decapsulation functions, with explicit

length constants. The interface does not treat every post-quantum construction as a generic operation called encrypt. This is selected source inspection, not a new execution, validation or security audit of the library. The research project also documents limitations on what its experimental integration work establishes. [\[125\]](#)

Crypto-agility is the ability to change cryptographic choices through a controlled design and migration process. It includes knowing where algorithms and keys are used, representing algorithm identities unambiguously, changing credentials and coordinating endpoints. A function argument that accepts several names is only one small part. If an old device cannot process the new credential size or a stored record omits its algorithm identifier, changing a library setting may leave the system unusable or ambiguous.

Negotiation also needs protection. Suppose two peers support a stronger profile, but an active intermediary changes their advertised capabilities and causes an unacceptable fallback. The presence of a post-quantum implementation somewhere in the software does not prevent that downgrade. The protocol has to bind the relevant choices and transcript to its authentication and key schedule, enforce policy and reject unsupported or inconsistent selections. Compatibility behavior must be assessed alongside the new primitive.

A **post-quantum/traditional hybrid** combines contributions from a post-quantum mechanism and a traditional mechanism within a specified construction. The intended robustness can be that the combined secret remains protected when at least one component retains the required security, but that result needs a suitable combiner and protocol argument. It does not follow from placing two ciphertexts beside one another. NIST SP 800-227 discusses multi-algorithm KEMs, combiner constructions and their security conditions. [\[54\]](#)

The earlier use of hybrid encryption meant combining public-key establishment with symmetric data protection. A post-quantum/traditional hybrid refers instead to combining establishment contributions from different algorithm families. Both expressions are useful, but the context must identify which combination is meant. A design might use both: first

combine traditional and post-quantum establishment, then derive keys for an AEAD channel.

For an illustrative two-contribution construction, write

(19.18)

The output

K_{traffic}

is the derived traffic-key material, and KDF names the specified key-derivation procedure. The two

Z

values name the component shared secrets, and the context represents the specified protocol information binding their use. This notation states the roles, not an approved generic combiner recipe. Exact input encoding, extraction, component validation, failure behavior and authentication must come from the selected construction. One cannot infer a one-component-survives theorem for every function named KDF.

Replacing only establishment is not the same as replacing signatures. A connection might derive a combined secret while still authenticating the peer with a traditional certificate and signature. The security question for a recorded session can therefore differ from the question for an active attacker at the time of a future handshake. The credential chain, the verifier's trust anchors and the agreement mechanism need separate inventory and migration decisions.

Algorithm diversity also develops through a public process. NIST selected the code-based HQC mechanism for standardization in March 2025 to diversify its key-establishment portfolio. A selection announcement is a different status from the final FIPS publications described above. Broader selection and transition documents should be checked for their stated stage rather than converted into a claim that every named candidate is already a final, interchangeable standard.

19.9 Migration and the limits of forecasts

Harvest now, decrypt later describes an attacker retaining encrypted information or public exchange data in the hope of exploiting a later capability. The data's required confidentiality lifetime matters. A message that loses all sensitivity tomorrow has a different exposure from a medical archive or research record that must remain confidential for decades. The relevant question is not only whether an attacker can read it at the time of interception.

Migration also takes time. An organization may have to discover hidden dependencies, replace long-lived devices, issue new credentials, change protocols and coordinate several independent suppliers. A small library demonstration does not measure that entire process. Inventory from Chapter [14](#) becomes useful again: it connects particular data, services, keys and algorithm choices to a concrete replacement plan.

A simple planning relation helps name the quantities:

(19.19)

Here

S

is the required confidentiality lifetime of newly protected information,

M

is the time needed to complete migration and

Q

is an assumed time until a relevant attack becomes practical. If protection continues through a migration period of length

M

, the latest sensitive data protected using the old mechanism may need secrecy until roughly

$M + S$

from the starting point. The inequality flags a problematic scenario when the assumed attack time lies inside that combined window. It is a planning model, not a prediction or security proof.

For example, suppose the information needs fifteen years of confidentiality and a transition could take five years. The combined window is twenty years. If a planning scenario places the relevant attack within that window, waiting for it before beginning would leave a gap. The calculation does not tell us when such an attacker will exist, whether a particular recording contains enough information or how every attack resource will develop. The uncertain quantity remains uncertain after being given a letter.

Nor does migration erase already collected evidence. Changing today's key or re-encrypting a retained local file cannot remove a ciphertext that an attacker copied earlier. If the old transcript supports a future attack, a new protected transcript does not retroactively replace it. Migration can reduce new exposure and protect retained copies under new controls, while the confidentiality history of information already disclosed or recorded remains part of the assessment.

Signatures raise another timeline. A future attacker able to forge an old signature scheme might create an apparently historical document. Preserving confidence in old evidence can involve trusted timestamps, archival validation records and renewal under suitable mechanisms before the earlier evidence becomes unreliable. Merely writing an old date into signed data is not a trustworthy timestamp. The appropriate archival policy depends on what the verifier must establish and which earlier evidence can still be trusted.

Implementation work should examine the complete path: message sizes, certificate chains, parser limits, fragmentation, computation cost and failure behavior. A successful laboratory exchange does not establish that every middlebox, embedded device or recovery service can handle the same profile. Measurements should identify actual parameters, platforms and protocol versions. Their practical value is lost when a result for one machine is presented as a universal latency guarantee.

Official transition guidance must be read within its jurisdiction and publication stage. At the source check for this chapter, NIST IR 8547 was listed as an initial public draft, rather than a final report. Its proposed transition approach is relevant research and planning context, but it should not be represented as a final worldwide deadline applying to every reader.

Concrete requirements come from the applicable final policy, standards and system obligations. [\[124\]](#)

Hardware progress does not reduce to one number either. Useful estimates identify the algorithm, target key, logical operations, error-correction assumptions, physical error rates, storage and permitted runtime. Different assumptions can produce different resource estimates without changing Shor's mathematical result. A carefully bounded forecast can help planning; a confident calendar date without those assumptions cannot supply the same evidence.

Post-quantum cryptography therefore changes the computational foundation while preserving familiar responsibilities. We still need appropriate randomness, exact encodings, authenticated identities, correct state transitions and trustworthy implementations. The standards supply important mechanisms, and research continues to examine their assumptions and deployment. The aim is a controlled response to a defined threat, rather than replacing every uncertainty with a new label. The next chapter turns to a different question: how to prove or compute something useful while keeping specified information private.

Questions to check your understanding

- What do amplitudes and their squared magnitudes represent?
 Calculate the squared magnitude of $\frac{(3 + 4i)}{5}$, defining i , and explain why one measurement does not reveal every amplitude.
- Define every symbol in the Hadamard calculation. Starting from amplitudes $(1, 0)$, calculate the outputs after one and after two Hadamard gates.
- Why do H
 $—$
 H
 and
 H
 $—$
 Z
 $—$
 H
 , both starting at $|0\rangle$, give different ideal outcomes? Distinguish a relative sign from a global sign.
- Derive the Bell state from $|00\rangle$. Why do equal standard-basis measurement records alone not prove entanglement or allow faster-than-light communication?
- What does the optional Bell-state program calculate? Explain its index order, its use of squared magnitudes and the difference from

executing on Qiskit-controlled hardware.

6. Identify the classical host, control, cold processor and readout roles in the IBM example. Why do processor-family names or a physical-qubit count not establish a cryptographic attack?

7. Explain

$$P_{\text{none}}$$

,

$$p$$

and

$$G$$

in the no-event model. Calculate the stated example and explain why it is not a processor-fidelity prediction.

8. Define a shot, expectation value, bias and sampling variation. Calculate the mean from 5,500 zero outcomes and 4,500 one outcomes. Why does more sampling not remove every systematic error?
9. Explain each term in the two-component weighted-variance formula. What do negative weights and an unbiased estimate mean? Why does halving the statistical standard deviation require four times the shots under the stated model?
10. Explain how the repetition encoding detects a middle-qubit X error without reading the logical amplitudes. Why does it not correct every phase error? Distinguish this task from mitigation and fault tolerance.
11. For $a = 2$ modulo 15, find the order and gcd factors. Explain the information carried by the ideal Fourier fractions $1/4$ and $1/2$, and why classical checking remains necessary.
12. Work through the four-candidate Grover example from initial amplitudes to final probabilities. Explain how its exact small result

relates to the general

$$2^{k/2}$$

query scale and differs from Shor's result.

13. Construct the first two noisy teaching rows for secret

$$(3, 5)$$

. Why can a residual of 16 modulo 17 represent the small error

$$-1$$

?

14. Explain every object in

$$h = \Delta \epsilon \times \text{load}()$$

. Why do small errors and a complicated-looking lattice picture alone not establish security?

15. Why does the optional 289-candidate program not demonstrate the hardness of LWE or break a standardized algorithm? What conditions limit the scope of Regev's foundational reduction?

16. Distinguish ML-KEM, ML-DSA and SLH-DSA by their roles. Why is a 32-byte shared secret not a signature or automatically a proof of 256-bit scheme security?

17. What is implicit rejection, and why must an application not treat replacement secret material as proof of an authenticated peer or message?

18. Distinguish post-quantum/traditional establishment from public-key-plus-symmetric hybrid encryption. Why does an unspecified KDF call not prove the desired robustness?

19. Explain

$$S$$

,

$$M$$

and

$$Q$$

in the planning relation. What can changing present keys or re-encrypting a local file not remove from an attacker's earlier recording?

20. Why do a published standard, a draft, a mitigation benchmark, a source-code inspection and a hardware forecast establish different things? How should Qedma's and the named researchers' contributions be assessed?

20. Proving and Computing with Private Data

Part VI: The Next Questions

The earlier chapters mostly protected messages while they travelled, remained in storage or established trust between endpoints. A different difficulty appears when someone needs to use protected information. A service might need evidence that a customer meets a condition, without receiving the customer's complete record. Several organizations might want a joint calculation without pooling their original datasets. A device might need several independent approvals to sign, without placing its complete signing key on one machine. Decrypting everything in one place would change the trust arrangement that these applications are trying to preserve.

This chapter develops several ways to separate information from the work it supports. Zero-knowledge methods concern evidence about a specified claim. Secret sharing distributes information across participants, while threshold signing lets a qualified group authorize a signature. Secure multiparty computation coordinates work on private inputs. Homomorphic encryption permits specified processing while the data remain encrypted. These methods can be combined, but each begins with a different interface and a different question.

We will introduce the vocabulary through concrete examples, then work through a small proof transcript, a complete secret-sharing calculation and elementary shared arithmetic. An approximate-number example will explain why encrypted computation also needs a numerical error analysis. The numbers are deliberately small enough to inspect. They explain the operations and their limits; they are not cryptographic parameters or complete implementations. Throughout, the central question is what a participant is allowed to learn, and which evidence establishes that the agreed task was performed correctly.

20.1 Proving a claim without revealing the witness

A **statement** is a claim expressed in a form that the proof system can check. A **witness** is additional information that makes that claim demonstrable. For example, a public value may be the digest of a private document. A person could know a document that hashes to that value without wanting to publish it. The claim must identify the exact hash function, encoding and public digest. It should not silently expand into the unrelated claim that the document is true, legally valid or written by a particular person.

One useful notation makes the distinction explicit:

(20.1)

Here

x

denotes the public statement data,

w

the witness and

R

a specified checking relation. The result 1 means that the relation accepts those inputs. For a digest-preimage claim,

x

could contain a digest,

w

a candidate document and

R

the operation that encodes and hashes the document and compares the result. The notation does not require that every relation be a hash comparison. It says which information is public and which information the prover needs to keep private.

The **prover** tries to convince a **verifier** that the required claim holds. In an ordinary demonstration, the prover might simply send the witness and let the verifier run

R

. A **zero-knowledge** construction instead aims to provide the specified evidence without giving the verifier additional knowledge about the witness. The foundational work of Shafi Goldwasser, Silvio Micali and Charles Rackoff formalized interactive proof systems and knowledge complexity. Its importance is the precise treatment of what a verifier can learn from interaction, rather than a promise that any persuasive-looking conversation is private. [\[137\]](#)

Three properties should be read separately. **Completeness** concerns acceptance of a valid claim when the prescribed honest process is followed. **Soundness** limits the ability to obtain acceptance for an invalid claim. **Zero knowledge** concerns the verifier's information, usually expressed through a suitable simulation argument. Their exact formulations depend on the construction and adversary model. A protocol may be convincing but reveal its witness, or conceal a witness while failing to establish the intended claim.

A **proof of knowledge** adds a particular kind of requirement: a successful prover must be related to an algorithm that can extract a valid witness under the definition's conditions. An extractor is a tool in the security argument, not a button that an ordinary verifier can press to recover every honest user's secret. For a public group element, a discrete logarithm may exist as a mathematical fact even if it is hard to calculate. Demonstrating knowledge of that logarithm is a different claim from merely observing that it exists.

The simulation idea can initially seem paradoxical. If a transcript can be generated without the witness, why should the interaction convince anyone? The answer concerns which choices a simulator can make and which choices a real prover faces. A simulator may generate a complete transcript in a different order, while a real prover has to answer an unpredictable challenge after sending an earlier message. The distributions can match without giving the real prover control of the verifier's next random choice.

This also limits the everyday interpretation of zero knowledge. The verifier learns the statement being proved, the public inputs and that the specified acceptance condition was met. Those facts can be informative. A proof that a private age exceeds a public threshold still reveals the threshold condition. Repeated proofs with chosen thresholds can reveal more. Zero

knowledge is defined relative to the permitted public statement and auxiliary information; it does not make every possible inference about a person disappear.

An application therefore has to choose its statement carefully. If a public commitment represents a private account balance, a range proof might establish that the committed number lies in an allowed interval. It must also bind that commitment to the intended account, asset, time and authorization policy. Otherwise a correct proof about one number could be presented where evidence about another was needed. The cryptographic proof establishes its encoded relation, while the application determines whether that relation answers the actual question.

20.2 Interactive proofs and succinct arguments

A three-message example

Chapter [13](#) introduced Schnorr-style signature arithmetic. We can reuse its group vocabulary to explain an interactive knowledge claim. Let

p

be a prime modulus, and let

g

generate a group of prime order

q

modulo

p

. The public value is

$Y = g^r \bmod p$

, and the prover's witness is the exponent

w

, represented modulo

q

. The claim is knowledge of an exponent consistent with that particular public value and group.

The prover first chooses a fresh uniform secret

r

modulo

q

and sends

$T = g^r \bmod p$

. The verifier then chooses an independent uniform challenge

c

modulo

q

. Finally, the prover sends the response

(20.2)

The symbol

z

is the response,

r

is the fresh secret used for this interaction,

c

is the verifier's challenge and

w

is the witness. The expression

cw

means multiply the challenge by the witness before adding

r

. Arithmetic on exponents uses modulus

q

, while the group check on the right uses modulus

p

. The verifier accepts only after the required parameter, membership and encoding checks and the displayed group equation.

The equation follows from the exponent rules:

. Reducing the exponent modulo

q

preserves the group element because

$$g^q = 1$$

in this group. This establishes completeness of the algebraic check, not every security property of a complete protocol. RFC 8235 describes Schnorr identification and its non-interactive transformation using a subtraction convention for the response. Our plus-response convention rearranges the corresponding verification equation; its symbols and arithmetic are defined here rather than copied across incompatible conventions. [\[138\]](#)

For the original teaching transcript in Figure [20.1](#), take

$$p = 23$$

,

$$q = 11$$

,

$$g = 2$$

and

$$w = 7$$

. The public value is

$$Y = 2^7 \bmod 23 \\ = 128 \bmod 23 = 13$$

Choose the disclosed teaching value

, giving

. If the challenge is

, the response is

$$z = (4 + 3 \times 7) \bmod 11 \\ = 25 \bmod 11 = 3$$

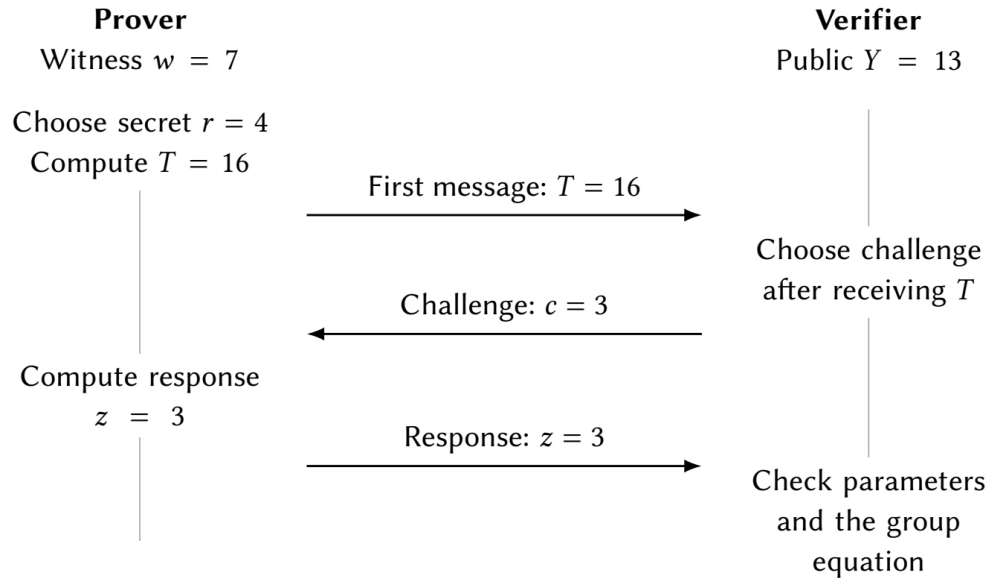


Figure 20.1. An original small interactive transcript, using the plus-response convention. The verifier chooses its challenge after receiving

. The secret values shown beside the prover are disclosed only for teaching.

The verifier's left side is

$$2^2 \bmod 23 = 8$$

. On the right,

$$13^2 \bmod 23 = 8$$

, so

$$13^3 \bmod 23 = 8 \times 13 \bmod 23 = 12$$

The product

leaves remainder 8 modulo 23. Both sides agree. A real group must be large enough for its computational assumptions, and the fresh secret must remain protected; this tiny example provides neither deployment security nor a reason to reuse its published values.

What simulation and extraction explain

For an honest verifier that selects

C

independently and uniformly, a simulator can first choose

C

and

z

uniformly modulo

q

, then set

(20.3)

The superscript

-1

denotes a multiplicative group inverse. It does not mean negate the value or perform ordinary real-number division. The simulator knows the public values

g, Y, p, q

and chooses the transcript values

C, z

; it does not need

w

. Multiplying

g^z

by the inverse of

Y^C

makes the verification equation hold by construction.

In our disclosed example, choosing

$c = 3$

and

$$z = 3$$

gives

$$g^z = 8$$

and

$$Y^c = 12$$

modulo 23. The inverse of 12 is 2, because

$$12 \cdot 2 = 24 \equiv 1 \pmod{23}$$

. The simulator therefore obtains

$$Y^c = 12 \text{ and } Z = 12$$

, matching the earlier transcript. For each challenge, the mapping from a uniform

r

to

$$Z = r + c \pmod{q}$$

modulo

q

is a permutation. Consequently the honest transcript and the simulator have the same distribution in this simple model.

That is an **honest-verifier zero-knowledge** argument for the interaction described. It should not be silently promoted to security against every malicious verifier, concurrent session schedule or altered protocol. Stronger definitions require the corresponding constructions and arguments. In particular, generating an accepting transcript offline is different from successfully answering a new unpredictable challenge after a fixed first message has already been sent.

The knowledge side can be seen from two accepting responses to the same first message with distinct challenges:

$$(20.4)$$

The subscripts 1 and 2 label the two transcripts. Both use the same

T

and public

Y

. Dividing the corresponding group equations gives

$$g^{y_1} = g^{y_2}$$

. Since the challenges differ modulo the prime

q

, their difference has an inverse. Multiplying by that inverse recovers the exponent

w

modulo

q

.

With the same teaching

$$r = 4$$

and witness 7, challenges

$$c_1 = 2$$

and

$$c_2 = 3$$

give responses

$$z_1 = 7$$

and

$$z_2 = 3$$

. Their response difference is 4 and challenge difference is

$$-1 \equiv 10 \pmod{11}$$

. The inverse of 10 modulo 11 is 10, giving

$$w = 7 \pmod{11}$$

. A security argument can use a suitable extractor that obtains such transcripts under its defined conditions. An honest implementation must not answer unrelated challenges using the same secret

r

; that behavior would expose the witness by the same calculation.

A prover without the witness can prepare an accepting transcript for one guessed challenge using simulation. If the verifier later selects uniformly from all

q

challenges after receiving that fixed first message, matching the guess has probability

$1/q$

. Here that is

$1/11$

. Repetition can reduce such a simple guessing probability when fresh choices, independence and the intended protocol conditions hold. This intuition is not a complete security proof for arbitrary adversarial behavior, and the small challenge space is only a teaching device.

From interaction to compact evidence

In a **Fiat–Shamir transformation**, a specified cryptographic hash calculation supplies a challenge derived from the public statement, first message and required context. The prover can then send a non-interactive object that the verifier checks later. Exact encoding, domain separation, challenge derivation and the security model are essential. A prover that tries many first messages gets many possible hash outputs, so replacing a random challenge with a hash is not justified by saying that the hash looks random.

This transformation also changes the simulation argument. A simulator that freely selected a challenge in the interactive example cannot simply ignore a fixed hash result. Appropriate proofs often use a random-oracle model or other stated assumptions. The earlier honest-verifier explanation makes the information roles visible, but it does not establish every non-interactive or quantum-adversary guarantee. The construction's actual statement and model must accompany the claim.

Interactive proofs and **arguments** are often distinguished by the adversary against which soundness is required. In the usual terminology, a proof's soundness holds even against computationally unbounded cheating provers, while an argument relies on a computational restriction or assumption. Both can be valuable. The everyday practice of calling every object a proof should not hide that distinction in a technical assessment.

A **succinct** argument reduces the size of the evidence and the verifier's work relative to an appropriate representation of the computation. A **SNARK** is a succinct non-interactive argument of knowledge. The additional description zero knowledge concerns privacy of the witness; succinctness alone does not supply it. Nor does a small output imply that generating it was cheap. Preprocessing, proving time, memory and public-input processing still contribute to the complete cost.

Jens Groth's 2016 pairing-based construction provides a concrete research example: its argument for arithmetic-circuit satisfiability contains three group elements. A **pairing** is a specified mathematical map between groups with an algebraic compatibility property used in verification. The construction uses a common reference string and a particular setup/security model. Three elements is a construction-specific size statement, not a universal three-byte proof, a guarantee of constant work for reading arbitrary public data or the size of every later proof system. [\[139\]](#)

An **arithmetic circuit** represents a computation through operations such as field addition and multiplication. Compiling a program into constraints must preserve the intended meaning. A missing range constraint can admit values that the application would reject, even if the resulting proof is valid for the incomplete circuit. Public inputs also have to bind the claimed output to the correct dataset, function and context. A correct proof of the wrong encoded program is still the wrong evidence for the application.

Setup choices matter independently. A common reference string is public parameter information used by participants. Some constructions generate it using secret setup material whose retention or misuse can undermine required guarantees. A carefully specified multi-party ceremony can distribute that trust, but a ceremony's name does not establish its execution. Other constructions have different setup arrangements. It is more useful to identify the actual assumptions than to divide all systems into a casually trusted or trustless category.

20.3 Secret sharing and threshold signatures

Distributing a secret

Copying a complete secret onto several machines improves availability but increases the number of places where that secret can be stolen. Cutting an encoded key into ordinary substrings has a different problem: each piece exposes part of the original representation. **Secret sharing** uses a mathematical construction so that qualified groups can reconstruct the secret while smaller groups receive no additional information under the scheme's privacy conditions.

In a
 k
 -of-
 n
 threshold scheme,
 n
 counts the participants and
 k

is the number of shares required for reconstruction. A share is associated with a distinct participant identifier; repeating one share does not create another independent share. Adi Shamir's construction represents the secret as the constant term of a random polynomial over a finite field. The field supplies exact modular arithmetic and inverses for nonzero elements, rather than an approximate fit to a real-number curve. [\[140\]](#)

For a complete example, let the field modulus be the prime
 $p = 17$
 , choose threshold
 $k = 3$
 and issue
 $n = 5$
 shares. Put the secret
 $s = 9$
 into

The capital
 X
 is a formal polynomial input. The values
 a
 and
 b
 are independently uniform field coefficients;
 s
 is the secret and
 p

determines the arithmetic. The polynomial has degree at most two, so three suitable points determine it. Uniform choices include zero coefficients. Excluding them without analysis changes the distribution and can affect the claimed privacy.

We disclose
 $a = 4$
 and
 $b = 6$
 for teaching. A participant with nonzero identifier
 x_i
 receives

$y_i = f(x_i)$
 , so its share is the pair

(x_i, y_i)
 . The subscript
 i

labels that participant. The identifier is not secret, while the share value needs the scheme's required protection. Identifier zero is excluded because

$f(0) = s$
 would directly reveal the secret.

Table 20.1. Original three-of-five shares for the disclosed polynomial

modulo 17. The unreduced calculation is shown to make the generation steps visible.

Identifier	Ordinary calculation before reduction	Share value
1		
2		
3		
4		
5		

For identifier 3, the square is

$$3^2 = 9$$

, so

$$6X^2$$

contributes 54 and the unreduced sum is 75. Subtracting

$$4 \times 17 = 68$$

gives remainder 7. Equal share values at different identifiers are allowed: the two points

$$(2, 7)$$

and

$$(3, 7)$$

remain distinct. Their presence does not turn one participant into two or remove the need for a third qualified share.

Reconstructing the constant term

Choose participants with identifiers 1, 3 and 5. **Lagrange interpolation** lets us calculate the constant term directly:

(20.6)

The set

I

contains the selected participant labels. The summation sign means add one weighted share for every selected label. The coefficient

λ_i

is that share's interpolation weight at input zero. The product sign means multiply the displayed fraction for every other selected label

j

. A fraction means multiply by the denominator's modular inverse. Distinct field identifiers make each difference

$x_i - x_j$

nonzero and therefore invertible.

For identifier 1, the two numerator factors are

-3

and

-5

, whose product is 15. The denominator is

$(1-2)(1-3)=8$

. Its inverse modulo 17 is 15, since

$$8 \times 15 = 120$$

$$\equiv 1 \pmod{17}$$

Thus

. For identifier 3, the numerator is

and the denominator is

. The inverse of 13 is 4, giving

.

For identifier 5, the numerator is

$$(-1)(-3) = 3$$

and the denominator is

$$(3 - 1)(3 - 5) = 4$$

. Using its inverse 15 gives

$$3 \cdot 15 = 45 \bmod 13 = 6$$

. Substitute the corresponding share values:

The three terms correspond to identifiers 1, 3 and 5 in that order.
Subtracting

$$7 = 17 - 10$$

from 128 gives the original secret 9. This is exact field arithmetic; the shares are not averaged, and the numbers 4, 3 and 11 are modular interpolation weights rather than percentages.

Why do two shares not determine the secret? Fix any two distinct nonzero identifiers and their observed values. For every candidate secret

$$s'$$

in the field, there is exactly one coefficient pair

$$a', b'$$

making a degree-at-most-two polynomial pass through those points and

$$(0, s')$$

. Since the original coefficients were independently uniform, the observation has the same probability for every possible secret. It does not change the observer's prior distribution over secrets. This is stronger than saying that an attacker merely needs a faster computer.

That privacy statement has conditions. The shares must be sampled and protected as specified, and the attacker must remain below the reconstruction threshold. It does not authenticate share values, identify a dishonest dealer or prevent a qualified coalition from reconstructing. Ordinary Shamir sharing also does not show that a supplied collection is consistent with the intended original secret. **Verifiable secret sharing** adds mechanisms for checking required consistency under a defined model.

The optional program reconstructs the disclosed secret from every three-share subset. It checks identifiers and field ranges before taking inverses. The code is fixed public teaching arithmetic, not a key-backup service or a source of secure random shares.

Listing 20.1. Checking all three-share subsets of the public teaching example.

```
from itertools import combinations

P = 17
shares = [(1, 2), (2, 7), (3, 7), (4, 2), (5, 9)]

def recover_three(points):
    if len(points) != 3:
        raise ValueError("Choose exactly three shares")
    if any(type(x) is not int or type(y) is not int
           or not 0 < x < P or not 0 <= y < P
           for x, y in points):
        raise ValueError("Invalid field point")
    if len({x for x, _ in points}) != 3:
        raise ValueError("Repeated identifier")
    result = 0
    for x, y in points:
        weight = 1
        for other, _ in points:
            if other != x:
                weight *= -other * pow(x - other, -1, P)
                weight %= P
        result = (result + weight * y) % P
    return result

groups = list(combinations(shares, 3))
if not all(recover_three(group) == 9 for group in groups):
    raise AssertionError("Unexpected reconstruction")
print("All", len(groups), "subsets recover:", 9)
```

There are ten such subsets, and the printed result is All 10 subsets recover: 9. The negative exponent in Python's three-argument `pow` requests a modular inverse, matching the fractions in Equation [20.6](#). The check does not detect every altered share: a different set of three field points

still defines a polynomial and may reconstruct another secret. Authentication, consistency and authorization need their own mechanisms.

Using shares without assembling a signing key

Secret sharing for backup can reconstruct a complete key when it is needed. A **threshold signature** instead lets a qualified group compute a signature while its private signing material remains distributed under the protocol's assumptions. The verifier checks the resulting signature against the appropriate group public key. The threshold, participant identities and institutional approval rules are separate information that a single ordinary signature may not reveal.

FROST, developed by Chelsea Komlo and Ian Goldberg, provides a Schnorr threshold-signature construction. Its original research addresses coordination cost and security with concurrent signing. RFC 9591 later specifies two-round FROST signing with identified ciphersuites and required helper operations. The RFC is informational CFRG/IRTF work, not an Internet Standards Track specification. A historical paper's round description should not silently replace the published interface being implemented. [\[141\]](#)[\[142\]](#)

At a high level, participants first prepare fresh nonce material and publish commitments. They then compute signature shares tied to the selected participants, commitment list and message. A coordinator gathers these contributions, checks the required relations and assembles the signature. FROST's hiding and binding nonce roles and binding factors address threats that a naive sum of ordinary signatures would miss. This overview explains the workflow; it is not a substitute for its exact equations and validation rules.

The fresh nonce requirements from Chapter [13](#) remain essential, now across several machines and sessions. Reusing a participant's nonce material can expose its signing share. Recovery after a crash must not unknowingly restore already-used state. Participants must agree on the message and relevant signing data, and a malicious coordinator must not be allowed to turn inconsistent requests into unauthorized signatures.

Distributing the long-term key does not make these state and context requirements disappear.

The source of the shares matters too. A trusted dealer can generate a key and distribute shares, but a retained dealer copy preserves a single complete-key exposure path. **Distributed key generation** uses a specified collaborative protocol to create the required key relationship without that single initial holder. RFC 9591 leaves general key generation outside its main signing specification and supplies a trusted-dealer appendix. A deployment must choose and assess its key-generation process rather than assuming that threshold signing automatically provides a distributed ceremony.

Finally, a threshold is also an availability policy. Too few reachable cooperating participants can prevent signing, while a qualified malicious coalition may authorize an unwanted signature. Share refresh, participant replacement and backup require protocols that preserve the intended key and security conditions. Chapter [14](#)'s inventory and recovery questions still apply, with distributed state and trust boundaries added.

20.4 Computing together with private inputs

Defining the permitted result

Secure multiparty computation, abbreviated MPC, lets participants carry out a specified computation over their private inputs with defined security guarantees. Those guarantees identify the corrupted participants, the information they may learn and the permitted outcome when someone misbehaves. A useful application begins with a function and an output policy, rather than the general request to keep everything private.

For example, several organizations might hold local counts and want their combined total. Write the intended task as

(20.7)

The number

m

counts the input holders,

x_i

is holder

i

's private input,

F

is the agreed function and

y

is its result. The dots indicate the intervening inputs in order. An output policy specifies who receives

y

and which other information is released. The same mathematical function can have very different privacy consequences depending on its inputs and recipients.

A **semi-honest** participant follows the protocol but tries to learn from its observations. A **malicious** participant may change messages, choose inconsistent shares, withhold responses or otherwise deviate. A privacy argument against the first behavior is not automatically a defense against the second. The number and timing of corruptions, possible collusion, communication assumptions and treatment of aborts belong in the security statement.

It is also necessary to distinguish correctness from **fairness** and **guaranteed output delivery**. A malicious participant may be unable to make honest parties accept a wrong result yet still be able to stop the computation. Fairness concerns which parties learn outputs when others do; guaranteed delivery concerns whether honest parties obtain their outputs despite disruption. These properties cannot be inferred merely from the word secure.

Shared arithmetic and a multiplication step

An **additive sharing** of a field value splits it into parts whose sum equals that value modulo the field modulus. For two computing participants A and B, a sharing of

$$x$$

is denoted

$$[x]$$

and consists of local parts

$$x_A, x_B$$

satisfying

$$(20.8)$$

The brackets name a distributed representation; neither participant is supposed to receive both parts simply because the notation displays them. The subscripts identify local holders. The congruence sign says that the sum and

$$x$$

have the same remainder modulo

$$p$$

, not necessarily that their ordinary integer representatives are equal. Choosing one part uniformly and setting the other to make the relation hold hides

$$x$$

from a holder of only one part, provided the stated randomness and collusion conditions hold.

Take

$$p = 17$$

and the disclosed values

$$x = 4$$

,

$$y = 7$$

. One possible sharing has

$$x_A = 6, x_B = 10$$

and

$$3x + 9y = 15$$

. The sums reduce to

$$21 \bmod 17 = 4$$

and

$$24 \bmod 17 = 7$$

. Each participant can add its own parts: A obtains

$$6 + 9 = 15$$

, while B obtains

$$15 + 15 = 30$$

. Together these form a sharing of

$$x + y = 11$$

, because

$$15 + 13 = 28 \\ \equiv 11 \pmod{17}$$

.

Multiplication requires more. Multiplying the local parts gives

$$6 \times 9 + 15 \times 15 = 279 \\ \equiv 7 \pmod{17}$$

,

while the actual product

is 11 modulo 17. The missing terms are the cross products in

. Revealing every part would make multiplication easy, but would remove the intended input protection.

A **multiplication triple** supplies correlated shared values

$$[a], [b], [c]$$

satisfying

$$c = ab$$

in the field. For a fresh suitable triple, participants open the masked differences

$$d = x - a$$

and

$$e = y - b$$

, then use

$$(20.9)$$

Here

$$a, b$$

are secret independent masks,

c

is their shared product, and

$$d, e$$

are the differences made public within the protocol. Substitute

$$x = a + d$$

and

$$y = b + e$$

into their product:

$$(a + d)(b + e) = ab + db + ea + de$$

.

Replacing

with

gives the formula. Multiplication by public

or

can be applied locally to shares. The public term

must be inserted once in the agreed sharing representation, not independently added in full by every participant.

For a disclosed teaching triple, take

$$a = 13, b_1 = 10$$

. The differences are

$$d = 4 - 3 = 1$$

and

$$e = 7 - 5 = 2$$

. Substitution gives

$$15 + 1 \times 5 + 2 \times 3 + 1 \times 2 \\ = 28 \equiv 11 \pmod{17}$$

,

the correct product. The masking explanation assumes that the actual masks are protected and generated as required. Publishing the masks as this teaching example does removes that privacy.

A triple must not be reused for unrelated secret inputs. If the same mask

a

protected two opened differences

$$d_1 = x_1 - a$$

and

$$d_2 = x_2 - a$$

, subtraction would reveal

. The subscripts identify separate uses. This relation can leak information even without revealing either input directly. Correlated preprocessing is therefore consumable protocol state, with inventory, erasure and recovery requirements.

The SPDZ research of Ivan Damgård, Valerio Pastro, Nigel Smart and Sarah Zakarias provides a concrete architecture for arithmetic MPC with an active adversary. It combines preprocessing with authenticated shared values and checks that address malicious deviations. Its model can tolerate a dishonest majority with security conditions and abort behavior. The elementary triple calculation above explains one algebraic mechanism; it omits the authentication and consistency machinery and is not an implementation of SPDZ. [\[143\]](#)

In particular, giving a protocol more triples does not by itself prove that they were generated correctly. A corrupt multiplication correlation or altered opened value can make later arithmetic wrong. Secure preprocessing and checking are part of the construction. The offline/online distinction also does not make offline costs vanish: an application needs to account for their computation, communication, storage and possible regeneration after failure.

What the output still tells its recipients

Suppose two input holders compute their sum and both receive it. A holder that knows its own input can subtract it from the result to recover the other input. This inference is compatible with a protocol that reveals nothing beyond the permitted output and the holder's own input. The problem lies in the chosen disclosure policy, not necessarily in a failed encryption or sharing primitive.

Repeated queries can create similar effects. If an authorized total changes only because one known person was added, subtracting the old result from the new one reveals that person's contribution. Query controls, aggregation policies and, in some applications, a separately designed statistical-privacy

mechanism address such release questions. The cryptographic protection of intermediate work cannot make an overly revealing output harmless.

MPC also cannot make a false supplied value true by arithmetic alone. A participant may enter an invented count unless the application establishes a trustworthy input source or proves the required relation to authenticated data. A malicious-security argument can constrain protocol behavior while still allowing an adversary to choose its own permitted input. Correct execution and factual accuracy of the underlying record are different requirements.

20.5 Computing on encrypted data

An encryption interface that permits evaluation

Homomorphic encryption, abbreviated HE, allows specified computations on ciphertexts without supplying the decryption key to the evaluator. A data owner can encrypt inputs, ask another machine to evaluate an allowed function and decrypt the encrypted result. The evaluator's permitted view, key material and output access depend on the construction. Multiple independent data owners and keys require an appropriate multi-key or collaborative design; they do not automatically fit the simplest single-owner interface.

The intended route can be described as

(20.10)

The subscript

i

labels an input message

m_i

and its ciphertext

C_i

;

m

counts the inputs. The public key

pk

supports encryption, while

sk

is the protected secret decryption key. The function

F

is the chosen computation. The evaluation key material

evk

enables the construction's required evaluation operations, and

c_F

is the resulting ciphertext. The tilde marks the decoded output

$\tilde{y}$

, whose relation to the desired

$F(m_1, \dots, m_n)$

depends on whether the scheme provides exact or approximate arithmetic. Encryption's required randomness and all parameter checks are part of the operations, though not displayed as extra arguments.

For an exact arithmetic scheme, correctness means recovering the specified function value in the defined plaintext domain, within the construction's validity and failure conditions. A modular domain can wrap: if the plaintext modulus is 17, the sum of 12 and 9 is represented as 4. That is a correct modular result and a wrong ordinary-integer answer if the application expected 21. Bounds, encodings and domain choice must therefore be fixed before the protected evaluation.

Partially homomorphic schemes support a restricted kind of operation. **Somewhat homomorphic** schemes support a limited amount of suitable combined computation. A **leveled** design chooses parameters for a specified computation depth. **Fully homomorphic encryption**, or FHE, supports general efficient computation through the construction's methods and assumptions. These labels describe capabilities; they do not imply equal speed, unlimited memory or a free translation of every ordinary program.

Craig Gentry's 2009 work gave a foundational construction of FHE. A central idea is **bootstrapping**: homomorphically evaluating an appropriate

decryption-related computation to refresh a ciphertext under the scheme's conditions. The evaluator does not simply receive the plaintext secret key and decrypt all records. Required encrypted key information, supported circuits and security assumptions belong to the construction. A refresh has a cost and need not remove every approximation or restore the exact original data. [\[144\]](#)

Why approximate arithmetic needs its own explanation

CKKS, named for Jung Hee Cheon, Andrey Kim, Miran Kim and Yongsoo Song, develops homomorphic arithmetic on approximate numbers. Its encoding, encrypted operations, rescaling and decoding control representations and error through a specified construction. It serves applications where a quantified approximation is acceptable. It should not be described as arbitrary exact real-number computation merely because its inputs are written as decimals. [\[145\]](#)

An original unencrypted scaling model helps explain one source of approximation before introducing encrypted noise. Let

x

be a real number and choose a positive scale

Δ

. Form the integer

$x = \text{round}(x\Delta)$

, then interpret

x / Δ

as the stored approximation:

(20.11)

The hat identifies the approximation, and

round

means rounding to the nearest integer, choosing the even one at an exact tie. The vertical bars give absolute error, its nonnegative magnitude. Rounding changes

$$\Delta x$$

by at most one half, so dividing by positive

$$\Delta$$

gives the displayed bound. This bound belongs to this single rounding model; it is not the complete error bound of CKKS.

Take

$$\Delta = 1000$$

,

$$x = 1.2344$$

and a second input

$$y = 2.0013$$

. Multiplying by the scale gives 1234.4 and 2001.3. Rounding yields integer encodings

$$X = 1234$$

and

$$Y = 2001$$

, interpreted as

$$\hat{x} = 1.234$$

and

$$\hat{y} = 2.001$$

. Their signed errors are

$$-0.0004$$

and

$$-0.0003$$

. Each lies within the single-encoding bound

$$\Delta/2 = 500$$

.

The integer product is

$$XY = 1234 \times 2001 = 2,469,234$$

.

Its natural scale is now

, so its interpreted value is 2.469234. The exact product of the original decimal inputs is 2.47040472. Their difference is

, showing how small input errors can combine and be multiplied by the other input's size.

Write the input errors as

$$\epsilon_x = \tilde{x} - x$$

and

$$\epsilon_y = \tilde{y} - y$$

. Expanding the product gives

$$(20.12)$$

The Greek symbol

ϵ

names an error, with its subscript identifying the input. Substitute

$$\tilde{x} = x + \epsilon_x$$

and

$$\tilde{y} = y + \epsilon_y$$

, multiply the two brackets, and cancel the original xy

term. In our example the three remaining terms are

$$-0.00037032$$

,

$$-0.00000052$$

and

$$0.00000012$$

. Adding them gives the earlier

$$-0.00117072$$

. A useful accuracy analysis must track the magnitudes of the inputs as well as their initial precision.

To illustrate a return to scale 1000, divide the product integer by 1000 and round:

$$\text{round}(2469234 / 1000) = 2469$$

Interpreting that integer at scale 1000 gives 2.469, introducing a further difference of

from the already approximate product. This decimal example explains why reducing a representation’s scale can introduce more rounding. Actual CKKS rescaling operates on its ciphertext/modulus structure and includes additional sources of error; it is not implemented by these plain integer steps alone.

Table 20.2. An original plain-number model of encoding and rescaling. This table explains approximation; none of its rows is a ciphertext or a complete CKKS operation.

Stage	Value	Meaning
Original product	e	Exact product of the chosen decimals.
Stage	Value	Meaning
Encoded-input product	e	Integer product interpreted at scale
Stage	Value	Meaning
Rounded lower-scale result	e	One additional rounding in this model.

The final absolute error relative to the original product is

$$|2.469 - 2.47040472|$$

$$= 0.00140472$$

Whether that is acceptable depends on the application's required units, tolerance and use of the result. A calculation used near a decision boundary may need a much tighter bound than one used only to draw a rough trend. The presence of encryption does not remove the need to specify that numerical requirement.

Depth, representation and implementation

Multiplicative depth counts the longest chain of dependent multiplication steps in the relevant arithmetic circuit. For

$$F(x) = x^4$$

, compute

$$u = x \times x$$

and then

$$u \times u$$

. The temporary value

$$u$$

is

$$x^2$$

, and these two dependent squarings have multiplicative depth two. The polynomial's degree is four; degree and circuit depth are different quantities. Rearranging a computation can change its resource needs even when the intended mathematical result stays the same.

The recent approximate-HE survey by Weinan Liu and colleagues reviews CKKS developments, bootstrapping and applications with their accuracy and efficiency trade-offs. Its 2025 volume date and November 2024 online publication are different bibliographic events. Historical

measurements in such a survey describe their particular hardware, parameters and implementations; they are not a universal runtime for the reader's future computation. [\[147\]](#)

Microsoft SEAL's public CKKS example makes the representation issues concrete. It tracks scales and ciphertext levels while evaluating a polynomial, rather than treating encrypted values as ordinary floating-point variables. Its selected source comments explain how multiplication increases scales and rescaling changes the modulus level. This is useful source evidence about the interface and intended bookkeeping. We have not built or benchmarked SEAL for this chapter, and its example parameter choices are not presented as universal recommendations. [\[148\]](#)

Packing several values into a ciphertext can improve throughput for suitable workloads, but operations must still match the encoding and slot arrangement. Additional operations such as rotations can need their own evaluation keys and computation. A branch depending on a secret comparison cannot ordinarily be treated like a free visible branch in an unencrypted program. Comparisons, approximated functions, data movement and decoding can dominate work that looks simple in a short arithmetic formula.

There is a further distinction called **circuit privacy**: hiding appropriate information about the evaluated computation from the party receiving its result ciphertext. Protecting a data owner's plaintext from the evaluator does not automatically provide this other direction of protection. SEAL's security documentation explicitly states that it does not supply circuit privacy. An evaluator using its own sensitive constants therefore needs a suitable protocol and implementation analysis, rather than assuming that the data owner's encryption also protects every evaluator input.

Confidentiality also differs from proof of correct evaluation. An evaluator may return an encryption of the wrong result or an unrelated ciphertext. The fact that the owner can decrypt it does not establish that the intended function was executed. Conversely, an evaluator's ability to transform ciphertexts is deliberate: homomorphic encryption is designed to permit specified changes. The integrity goals of Chapter [7](#)'s authenticated messaging cannot simply be assumed for that different interface.

20.6 Privacy, correctness and practical limits

The methods in Figure [20.2](#) answer related but distinct questions. Zero knowledge can establish a relation while protecting a witness. Threshold signing distributes control of a signing operation. MPC carries out a joint computation under a participant/adversary model. HE lets an evaluator process ciphertexts under a defined key arrangement. None is a synonym for a complete private, correct and available application.

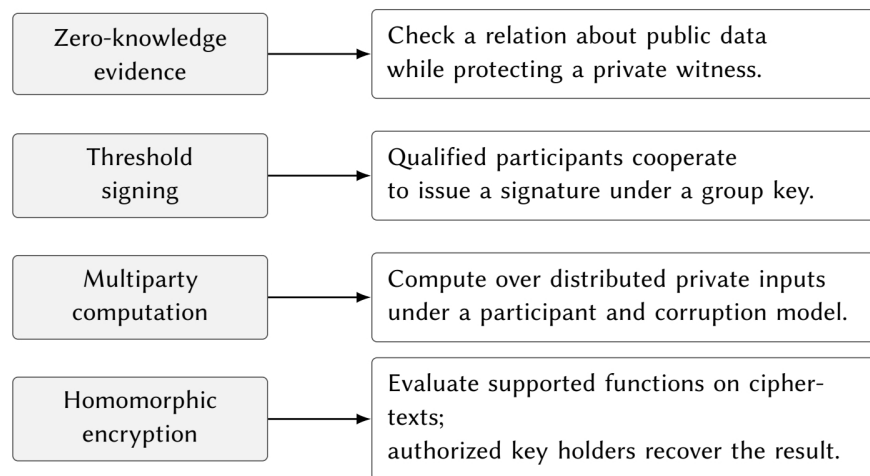


Figure 20.2. An original comparison of information roles. Each route needs its own assumptions, key or participant model, and output policy. The boxes identify tasks rather than interchangeable products.

A workflow may use more than one route. An MPC protocol can protect intermediate inputs, while a proof system supplies evidence about an agreed relation. An HE evaluator can return encrypted work together with separately specified verification evidence. Combining components requires the evidence to refer to the same inputs, function, output and context; placing two secure-looking objects together does not establish that binding.

The survey by Tariq Bontekoe, Dimka Karastoyanova and Fatih Turkmen examines verifiability for privacy-preserving computation over distributed data. It distinguishes protecting data from establishing correct execution and compares schemes with different assumptions and costs. That separation is useful when assessing a proposed service: the privacy claim, verifier role and result guarantee should each be identifiable. A summary saying encrypted and verified leaves too many questions unanswered. [\[146\]](#)

A proof can be mathematically valid yet refer to an untrustworthy input. A private calculation can be correct yet release a revealing output. A threshold signature can be valid yet express a decision made by a compromised qualified coalition. These are different failure paths. Addressing them needs clear data provenance, authorization, output policy and participant assumptions alongside the cryptographic construction.

Finite domains create another common mismatch. If a circuit proves a modular relation, it does not automatically prove an intended statement about unrestricted integers. If an approximate computation returns a value near a threshold, it does not automatically justify the same decision as the ideal real-number function. Range checks, overflow reasoning, precision bounds and explicit encodings connect the mathematical domain to the application's meaning.

Consider an original planning example with three organizations, each supplying a nonnegative count no greater than 1,000. Their desired ordinary-integer total is therefore between 0 and 3,000. A field representation for that total must be large enough to avoid confusing it with a wrapped residue; a proof also needs to enforce the input bounds it relies on. This size requirement concerns the data representation, not the separate cryptographic security parameters. A large enough field does not establish the correctness of an authentication or encryption scheme.

The application can then assign separate jobs to its protections. Authenticated submissions identify the intended contributors and reporting period. An MPC construction can protect intermediate inputs under its corruption model. A suitable verification mechanism can tie the accepted result to the agreed submissions and sum. The recipients still need an output policy: if two organizations already know both of their own counts and receive the total, they can calculate the third count. Each guarantee remains meaningful, but its scope must match what that recipient is allowed to infer.

The evaluator's knowledge can also extend beyond plaintext values. Message sizes, participant identities, query timing, access patterns and success or abort behavior may be visible. A construction can protect its defined inputs while leaving those channels outside its model. An application that needs stronger metadata protection must identify that

requirement explicitly, rather than inferring it from secret sharing or ciphertext processing.

Output access requires similar care. In a single-owner HE design, the owner may be the only party able to decrypt results. Returning detailed decryption feedback to an untrusted evaluator can create a new interface that the original security argument did not cover. In MPC, a party learning its output can combine it with other information. In zero knowledge, a public statement can expose an identifier or condition. The permitted release has to be assessed as part of the system.

Implementation evidence remains the subject of Chapter [18](#). Exact arithmetic checks can expose a wrong interpolation weight, a missing inverse or an incorrect scaling calculation. They cannot certify the security of an unimplemented protocol, validate an entire trusted setup or measure leakage from compiled software. A meaningful evaluation records the construction, version, parameters, adversary, workload, correctness conditions and resources actually assessed.

The quantum discussion also carries forward. Pairing-based arguments and Schnorr threshold signatures rely on mathematical assumptions threatened by the relevant quantum algorithms. A lattice-based HE component has a different foundation, but that alone does not make every surrounding proof, authentication method, key ceremony and implementation post-quantum secure. Claims about a composed system must account for its complete dependency chain.

For a reader examining a new claim, the useful starting point is a precise task. Identify what remains secret, who holds it, what is computed or proved, who learns the output and what happens under corruption or failure. The constructions in this chapter show several ways to change those information boundaries. Their value lies in the particular guarantees they support under clear conditions, with numerical and implementation evidence that matches the actual task.

Questions to check your understanding

1. Define the statement, witness, prover and verifier in

$$R(x, w) = 1$$

. Why can a valid digest-preimage proof fail to establish that the document is true or belongs to the intended account?

2. Distinguish completeness, soundness, zero knowledge and knowledge extraction. What does the verifier still learn from the public statement?
3. Calculate every step of the teaching transcript with

$$p = 23, q = 11, g = 2, w = 7, \\ r = 4, c = 3$$

.

Why do exponent arithmetic and group arithmetic use different moduli?

4. Reconstruct the simulated first message from

$$c = 3, z = 3$$

. Why does an honest-verifier transcript simulation not automatically establish security against every malicious verifier?

5. Use the two response/challenge pairs to recover the witness. Why must an implementation not reuse the same secret

$$r$$

across different challenges?

6. Distinguish a proof, an argument, a succinct argument and a zero-knowledge SNARK. What do public inputs, setup and circuit constraints still need to establish?
7. Generate all five Shamir shares from the displayed polynomial. Why must identifiers be distinct and nonzero, and why can two different shares have the same value?

8. Calculate the three interpolation weights for identifiers 1, 3 and 5, including each modular inverse. Show that their weighted reconstruction gives 9.
9. Why do two shares leave every candidate secret possible under the stated sampling? What does ordinary secret sharing fail to authenticate or enforce?
10. What does the optional program verify, and what would an altered but well-formed share demonstrate about its limits?
11. Distinguish reconstructing a signing key from threshold signing. What do FROST nonce state, message binding, key generation and the participant threshold contribute?
12. Define the function and output policy in MPC. How do semi-honest, malicious, abort, fairness and guaranteed-delivery questions differ?
13. Check the additive share sums and show why multiplying local parts gives the wrong product.
14. Derive the multiplication-triple equation and work through its numbers. What leaks if a mask is reused, and which SPDZ protections are absent from this elementary calculation?
15. Define every key, message, ciphertext, function and output in the HE interface. Why can exact modular correctness disagree with an intended ordinary-integer result?
16. Work through the decimal scaling example, input-error expansion and final error. Why is this not the complete CKKS noise calculation?
17. Explain multiplicative depth, rescaling, bootstrapping and packing without treating them as free operations. What does selected SEAL source reading establish?
18. Give separate examples of valid execution with false inputs, private execution with a revealing output and confidentiality without verifiable correctness. What additional evidence would the application need?

Conclusion: Understanding Digital Trust

Cryptography begins with a human problem: we need to communicate, store information and make decisions when other people may observe or change what reaches us. The journey from secret writing to modern protocols shows why a convincing appearance is not enough. A large key space can hide a weak construction. A correct calculation can belong to a system that offers no useful secrecy. A signature can verify even when the signed claim is false. We have therefore asked the same practical questions throughout this book: what is being protected, against which actions, under which assumptions, and with what evidence? Bits, remainders, groups and probability gave us a language for answering those questions. Their purpose was to make the reasoning visible, including the steps that an unfamiliar formula might otherwise conceal. Understanding a method means knowing both what its operations do and why its security claim needs more than a successful example.

Modern protection depends on how several methods work together. Encryption, authentication, derivation and signatures each have a defined role. Nonces, encodings, identities and protocol state connect those roles to particular messages and participants. TLS and secure messaging showed how such connections develop over an exchange, while key management extended the discussion to storage, access, replacement and recovery. Failures then revealed the importance of the surrounding implementation: a reused value, a revealing error or an exposed endpoint can change the problem without defeating the underlying mathematics. Testing and verified software provide valuable evidence when their inputs, properties and remaining dependencies are clear. That evidence supports careful decisions; it does not remove the need to examine the application that uses a component. A reader can now follow a protection claim from its intended goal through the operations and checks on which it depends.

The final chapters broadened that view. Quantum algorithms change particular computational assumptions, while post-quantum constructions seek protection against the resulting attacks. Private proofs and private computation change what participants must reveal to achieve a task. These

developments build on precise models, rather than making every earlier distinction disappear. New methods still need correct inputs, protected keys, suitable parameters and a clear account of the information their outputs disclose. Continue learning by returning to an explanation, checking a small calculation and reading the identified source when a claim matters. For a real system, follow maintained specifications and examine the complete setting instead of turning a teaching example into a deployment. The most useful result of this book is a lasting habit of asking clear questions about digital trust, and following the evidence far enough to understand the answer.

Mathematical Reference

This appendix collects the operations that recur in the book. Use it to recover a meaning or check a step, then follow the chapter reference for the full argument. Each letter is local to its calculation. The letter used for a prime in one example need not name a probability in another. A calculation with small public numbers explains an operation; choosing those numbers for a real key would be a different and usually unsafe decision.

Bytes, powers and XOR

A byte contains eight bits. As an unsigned integer, it has a value from 0 through 255. For example, the binary byte 10100111 has value

The eight multipliers are the bits, read from left to right. The numbers in parentheses are their place values: successive powers of two. A zero bit contributes nothing. Hexadecimal writes the same byte as A7: the first digit has value 10 and the second has value 7, giving

10 × 16 + 7 = 167

. This numeric reading is separate from interpreting a byte sequence as encoded text. Chapter [3](#) explains the distinction and the choice of byte order for larger integers.

XOR compares equal-length bit strings one position at a time. It gives 1 for different bits and 0 for equal bits. For a byte example,

Here



names XOR, and each two-digit item is one byte written in hex. The first operation applies a mask; the second applies the same mask again and restores the original input. This identity does not require the mask to be

secret or random. Secrecy needs additional conditions, as the one-time-pad discussion in Chapter [1](#) explains. XOR, ordinary addition and joining two byte strings are different operations.

Remainders and congruence

For an integer

a

and a positive integer modulus

n

, division with a nonnegative remainder gives

The integer

q

is the quotient, and

r

is the remainder. The expression

$$a \bmod n$$

names that remainder. For

$$a = -3$$

and

$$n = 12$$

, choose

$$q = -1$$

: then

$$-3 = (-1) \cdot 12 + 9$$

, so

$$r = 9$$

. The remainder is not negative under this convention. The expression

$$a \equiv b \pmod{n}$$

says that

a

and

b

have the same remainder on division by

n

, or equivalently that their difference is an integer multiple of

n

.

For example,

$38 \equiv 5 \pmod{11}$

because

$38 - 5 = 33 = 3 \cdot 11$

. Congruence does not say that the ordinary integers 38 and 5 are equal. We can reduce intermediate additions and multiplications without changing the final remainder. Thus

$$(8 \times 9) \bmod 11 \\ = 72 \bmod 11 = 6$$

,

and replacing either input by a congruent value preserves the result. See Section [3.5](#).

Inverses and division

The greatest common divisor of two integers is the largest positive integer that divides both. We write it as

$\gcd(a, n)$

. An integer

a

has a multiplicative inverse modulo

n

exactly when

$$\gcd(a, n) = 1$$

. An inverse is an integer

u

satisfying

Here

a

is the value we want to undo by multiplication,

n

is the modulus and

u

is its inverse. For

$$a = 7$$

and

$$n = 26$$

, the value

$$u = 15$$

works because

$$7 \cdot 15 = 105 \equiv 1 \pmod{26}$$

. Modular division by 7 therefore means multiplication by 15 in this setting. It does not mean ordinary division followed by rounding.

For a small equation

$$7x \equiv 3 \pmod{26}$$

, multiply both sides by 15. This gives

$$105x \equiv 45 \pmod{26}$$

. The letter

x

is the unknown residue; checking gives

$$7 \times 19 = 133$$

$$\equiv 3 \pmod{26}$$

In contrast, 2 has no inverse modulo 26 because their greatest common divisor is 2. A prime modulus makes every nonzero residue invertible. These conditions matter in the public-key and sharing calculations. Section [4.2](#) develops the method for finding the inverse.

Powers and group operations

A nonnegative integer exponent counts repeated multiplication, with exponent zero giving the multiplicative identity. Modular powers reduce the result modulo a stated modulus. Repeated squaring avoids constructing every intermediate power separately. For example,

Every congruence here uses modulus 23. The values 3, 9 and 12 are the successive remainders after squaring. Since

$$13 \equiv 8 \times 4 \equiv 1$$

, the final power uses those three factors. The product

$$12 \times 9 \times 7$$

is 756, whose remainder is 20. Exponents describe the operation; they are not labels identifying different keys. Section [4.3](#) explains the procedure.

A group supplies an operation, an identity and inverses, with the closure and associativity properties defined in Section [4.4](#). Multiplicative notation

uses a product and powers. The elliptic-curve chapters use additive notation:

kP

means adding a point

P

to itself

k

times for a positive integer

k

, with zero copies giving the identity point. It does not mean multiplying each coordinate by

k

. The order of

P

is the least positive number of additions that returns to the identity.

Coordinate arithmetic uses the field modulus, while scalar reductions use the appropriate point or subgroup order. Chapter [12](#) keeps these different numbers explicit.

Probabilities and repeated trials

A probability lies between 0 and 1. For independent trials with a fixed success probability

p

, the probability of at least one success in

t

trials is

Here

p

is the success probability for one trial, and

t

is a nonnegative integer counting the trials. The value

$$1 - p$$

is the chance of one failure. Independence lets us multiply those failure probabilities, giving

$$(1 - p)^t$$

for all trials failing. Subtracting from 1 leaves the chance of at least one success. With

$$p = 1/4$$

and

$$t = 3$$

, the result is

$$1 - (3/4)^3 = 37/64$$

. Adaptive guesses without replacement do not generally follow this independent-trial model.

For

t

independent uniform selections from a set of

N

values, where

$$1 \leq t \leq N$$

, the probability of at least one repeated value is

The integer

N

counts the allowed values, and

t

counts the selections. The product sign

$\prod$

means to multiply one factor for each integer

i

from 0 to

$t - 1$

. After

i

different earlier selections, exactly

i

values would cause a repeat, so

$1 - i/N$

is the conditional chance of avoiding one. Multiplying gives the chance of no repeat; subtraction gives its complement. With

$N = 4$

and

$t = 3$

, this is

$1 - (1/4)(3/4)(2/4) = 7/8$

. Repeated values are certain if

$t > N$

. The calculation assumes the stated uniform, independent sampling; it does not estimate every real nonce generator. See Chapter [5](#).

Reading a cryptographic interface

An expression such as

$\text{Enc}(K, M)$

means to run the named encryption operation on key

K

and message

M

in the specified construction. Its notation alone does not say how fresh randomness or a nonce is supplied. Read the definition of the interface before using it. Likewise, decryption can produce an accepted message or a failure result; the application must respect that distinction. Hashing an input, authenticating it with a secret key and signing it with a private key are different interfaces even when each produces a short byte string.

Before following any longer formula, identify its permitted inputs and result. Check whether a number counts bits, bytes, trials or group elements. Distinguish a claimed identity from an approximation, and a correctness relation from an attack probability. The detailed calculations in the chapters provide those conditions locally. This appendix is a way back to them, rather than a new set of assumptions.

Running the Optional Examples

The main explanations do not require programming. The optional Python listings let you inspect selected inputs, reproduce calculations and observe the stated rejection cases. They use public teaching data or keys generated temporarily for that run. They do not contact a remote service, send a payment or operate a production account. Read the explanation before executing a listing so that you know which result it is intended to demonstrate. A completed run checks the assertions included in that program; it does not establish every security property of the surrounding construction.

A recorded environment

The listings were checked using CPython 3.12.14 and the Python `cryptography` package, version 50.0.2, on Windows. The package reported OpenSSL 4.0.3 as its cryptographic backend. This is the backend used by that package, which need not be the same library as a separate `openssl` command installed on a computer. These versions record a reproducible teaching environment. They do not promise support on every operating system or replace the maintenance policy needed by a real application.

Chapters [3–6](#), Chapter [8](#) and the public arithmetic examples in the later chapters use Python’s standard library or built-in operations. The AEAD, derivation, public-key and library-vector examples also use the `cryptography` package. In particular, the HPKE example requires the interface provided by the recorded version. A missing import can indicate a different package version; it does not show that a mathematical claim is false. The small quantum calculations in Chapter [19](#) use ordinary Python arithmetic. They require neither a quantum computer nor a paid hardware account.

Prepare a separate Python environment

Install Python from its official distribution or an appropriately maintained package source, then create an empty directory for the examples. Open a terminal in that directory. The following commands create a local environment and install the recorded package version. They assume that `python` starts the intended Python interpreter.

```
python -m venv .venv
```

The environment places its interpreter in an operating-system-specific directory. On Windows, install and inspect the package with:

```
.venv\Scripts\python -m pip install cryptography==50.0.2  
.venv\Scripts\python -m pip show cryptography
```

On macOS or Linux, the corresponding interpreter is in the `bin` directory:

```
.venv/bin/python -m pip install cryptography==50.0.2  
.venv/bin/python -m pip show cryptography
```

Creating an environment keeps this exercise’s installed packages separate from another project’s packages. Installing a package requires a network connection to the package source; running the book’s printed examples does not. The commands do not activate the environment in the current shell, so each later command should name the local interpreter explicitly. If installation reports an unsupported platform or Python version, inspect that error and the package’s official documentation before changing the example. Do not disable a verification check merely to make a program finish.

Copy, run and inspect a listing

Copy the program text into a plain UTF-8 text file ending in `.py`, preserving indentation and ordinary straight quotation marks. Printed line wrapping is a display choice, not always a new program line. In particular, preserve byte-string literals and the text inside quotation marks. Save the file with a descriptive name, such as `example.py`, and run it with the local interpreter:

```
.venv\Scripts\python example.py
```

On macOS or Linux, replace the interpreter path with `.venv/bin/python`. A Python assertion checks that a stated condition is true; if it is false, execution stops with an error. Run these examples normally, without the `-O` option that removes assertions. Some listings deliberately alter a message, tag, label or other input and expect an exception. The program catches that expected failure and reports it. A caught rejection is part of the successful test, not a reason to remove the altered-input check.

Most listings contain their own imports and inputs. The second TLS listing deliberately continues the first: place both listings from Chapter [15](#) in one file in their printed order, or run them in the same Python session in that order. It uses helper functions and values established in the first listing. A missing name after running only the second part is an incomplete setup. The chapter identifies this dependency before the program.

Interpret the evidence

Known-answer examples compare an implementation with a published result. Round-trip examples check recovery for particular generated inputs. Negative cases check selected failures. These are useful but different kinds of evidence, as Chapter [18](#) explains. A randomized encryption can produce different bytes on repeated runs while still satisfying the checked behavior. A program's success message is therefore sometimes the stable result, rather than one fixed ciphertext.

The deliberately small modular, curve and quantum-search examples expose arithmetic that would be hidden by realistic parameter sizes. Their short loops, exhaustive searches and ordinary integer operations make no constant-time or production-security claim. Public test keys and fixed nonces belong only to their stated demonstration. Generated temporary secrets should not be printed, copied into an application or treated as credentials. When adapting an application, use the appropriate maintained library and protocol specification, including their input checks, state rules and usage limits.

Answers to the Review Questions

Chapter 1: From Secret Writing to Modern Cryptography

1. **An attack can exploit structure.** A fixed substitution preserves repeated letters and equality patterns. Frequency information, likely words and known plaintext can narrow the choices without trying every alphabet. The size of the full key set does not establish the cost of the best relevant attack.
2. **The stated protection should survive knowledge of the method.** The attacker still lacks the appropriately protected secret material, such as the key. Publication does not establish that a design meets a security goal; it permits the design and its assumptions to be examined without relying on hiding the procedure.
3. **The recovered message is 1011.** Position by position, $1101 \text{ XOR } 0110$ gives that result. For the fixed-length pad construction, secrecy requires a uniformly random pad independent of the message, kept secret and used only once for those positions. Correctly recovering the displayed example alone does not establish these conditions.
4. **The probability remains**
 $3/4$
. Perfect secrecy says that the ciphertext observation does not change the message probabilities. It does not make different messages equally likely if they had unequal probabilities beforehand.
5. **The observer learns**
 $m_1 \oplus m_2$
. The two copies of the pad cancel when the ciphertexts are combined. This may not identify both messages immediately. Other knowledge can make the relation useful, and knowledge of one complete message exposes the reused pad and the other message in those positions.
6. **Correct arithmetic does not identify the peer.** If the exchange has no appropriate authentication, an active intermediary may substitute public values and establish a different secret with each participant. The intended participants need evidence tying the exchange to the right parties and context.

7. **The third position would require E to encrypt to E.** That contradicts the reflector-based Enigma property discussed in the chapter. An alignment with no matching position passes only this necessary test; its wiring, settings and other consequences may still be inconsistent.
8. **The result depended on several kinds of work.** Polish mathematics and intelligence cooperation established essential early results; Turing developed a crib-based search and Welchman strengthened it with the diagonal board. Engineers made working machines, operators ran them, and analysts checked candidate stops. A stop identifies a possibility for further checking, not automatically the correct setting or a complete readable message.
9. **They addressed different cipher systems.** The Bombe helped eliminate Enigma setting candidates. Colossus supported the separate attack on Lorenz teleprinter encryption, with a central engineering contribution from Tommy Flowers and colleagues. Calling both machines one Turing invention confuses their mechanisms, purposes and contributors.

Chapter 2: What Cryptography Protects

1. **Integrity is essential.** The requirement is to detect an unauthorized change in the quantity. The fact that the order is public means that confidentiality is not needed for that particular requirement. The receiver may also need source authentication and an authorization rule before acting on an order. Those are additional questions about the application.
2. **The connection between the key and Nora is unestablished.** Verification gives evidence about the signed data relative to the supplied public key. An attacker could have supplied their own key under a false label. Sam needs an appropriate basis for trusting the key-to-source association, as well as the assumptions of the signature scheme.
3. **Authenticity alone does not establish freshness.** A replay can contain unchanged, previously authentic data. The application needs a way to identify the order and recognize that it has already been processed. A repeated tag or signature check need not fail when the original bytes are sent again.
4. **Correct recovery does not establish confidentiality.** A procedure that leaves its input unchanged can also recover the same message every time. The missing question concerns what an attacker can learn from the observations and operations allowed in the model. A round-trip test checks one kind of behavior, not all the ways to defeat a security goal.
5. **The signature can support the data-to-key relationship.** Under the relevant assumptions, it provides evidence that the signed data were accepted by verification under the corresponding public key. Connecting that key to the intended sensor or operator needs a trusted binding. Establishing that the measurement was accurate requires evidence about the sensor, its calibration and the measurement process.

Chapter 3: Bits, Bytes and Remainders

1. **Visible letters and encoded bytes are different units.** With a precomposed final letter, the UTF-8 encoding of *Café* is 43 61 66 C3 A9: five bytes. With an e followed by a combining acute accent, it is 43 61 66 65 CC 81: six bytes. The usual display can look the same even though the inputs differ. A protocol must specify any normalization it requires before the relevant cryptographic operation.

2. **The byte is hex A7, with unsigned decimal value 167.** The two four-bit groups 1010 and 0111 give the hex digits A and 7. Equivalently, the nonzero binary weights add to

0010101111

. The prefix 0x identifies hexadecimal integer notation; it is not part of the represented byte.

3. **Hex decoding produces one byte; literal text produces two.** Decoding 41 as hex gives the byte 41, whose decimal value is 65. The literal byte string b"41" instead contains the ASCII bytes for 4 and 1, written in hex as 34 31. Confusing these inputs changes both the data and their length.

4. **The results are 55, 69 and 5A 0F, respectively.** XOR compares corresponding bits without carries. Integer addition gives

90 + 15 = 105

, which is hex 69. Concatenation joins the two original one-byte inputs and is the only result here with two bytes. The fact that the same printed inputs appear in all three calculations does not make the operations interchangeable.

5. **The remainders are 1 and 9.** We have

107 = 1 × 100 + 7

and

107 = 0 × 100 + 107

. A logical left shift that retains a fixed width discards bits leaving the high end and inserts zeros at the low end. A rotation brings the departing bits back at the other end. For the chapter's eight-bit example, the results are 10001000 and 10001101.

6. **Byte order assigns different weights.** Big-endian interpretation gives

$1 \times 100 + 2 \times 20$

; little-endian interpretation gives

$1 \times 2 + 100 \times 10$

. The byte sequence remains 01 02. Choosing its numeric interpretation does not reverse the individual bits in either byte.

7. **The length check prevents silent truncation.** Python's `zip` would otherwise stop when the shorter input ends. The successful example checks the specified transformation, expected output and recovery for these inputs. It does not establish confidentiality or authentication. Anyone who knows the fixed public mask can reverse the operation, and a round trip alone is not a proof of security.

Chapter 4: Primes, Groups and Hard Problems

1. **A composite number must have a factor no larger than its square root.** For 97, this requires testing prime divisors 2, 3, 5 and 7; none divides it. The base-2 congruence for 341 is insufficient:

$$341 = 11 \times 31$$

is composite, but

$$2^5 \equiv 3 \pmod{341}$$

implies

$$2^{340} \equiv 1 \pmod{341}$$

. A necessary condition of primality is not automatically a sufficient test.

2. **The last nonzero Euclidean remainder is 1.** The remainders in dividing 26 by 7, then 7 by 5, then 5 by 2 are 5, 2 and 1. Back-substitution gives

$$1 = 5 - 2 \times 2$$

. Thus

$$-11$$

, represented as 15 modulo 26, is the inverse of 7. The check is

$$7 \times 15 = 105 \equiv 1 \pmod{26}$$

.

3. **The common factor 2 prevents an inverse of 6 modulo 26.** The congruence to 1 would require an even integer combination of 6 and 26 to equal 1. For the second equation,

$$x = 5$$

and

$$x = 18$$

give

$$6x = 30$$

and 108, both with remainder 4. Without an inverse, multiplication can map different inputs to the same output.

4. **The result is 20.** The successive squared residues are

$$7^2 \equiv 3$$

,

$$7^4 \equiv 9$$

and

$$7^8 \equiv 12$$

modulo 23. Because

$$13 \equiv 8 + 4 + 1$$

, combine

$$12 \times 9 \equiv 16$$

and then

$$16 \times 7 \equiv 20$$

. Intermediate reduction preserves sums and products modulo the same modulus, so the full ordinary power need not be stored.

5. **The element 3 has order 5.** Its powers generate

$$\{1, 3, 9, 5, 4\}$$

, returning to 1 at the fifth positive power. Exponents of 3 can therefore be reduced modulo 5, while the represented residues are multiplied modulo 11. The containing group's order is 10; it is a different quantity from both the subgroup order and the arithmetic modulus.

6. **The group is**

$$\{1, 5, 7, 11\}$$

and has no generator of order 4. Each nonidentity element squares to 1 modulo 12 and thus has order 2. No one element visits all four values by taking powers. Zero is excluded because it cannot have a multiplicative inverse modulo 12.

7. $A = 9$

,

$$B = 5$$

and the combined element is 3. We have

$$9^5 \equiv 9^2 \equiv 3 \pmod{11}$$

, agreeing with

$$3^{2 \times 3}$$

. Multiplying the public values instead gives

$$9 \times 5 \equiv 1$$

, which represents

$$g^{a+b}$$

, not

$$g^{ab}$$

. The tiny cycle also lets an observer recover the hidden exponents easily.

8. **The requested outputs differ.** DLP recovers an exponent; CDH calculates the combined element; DDH distinguishes a related candidate from an independent one under the stated sampling model. A DLP solver supplies a CDH attack, and a reliable CDH solver supplies a DDH distinguisher. Hardness implications go in the opposite direction, so DLP hardness alone does not establish DDH hardness. None of these computational assumptions by itself identifies or authenticates a peer.

Chapter 5: Probability, Randomness and Security

1. **The probabilities are**

$$1/4$$

and

$$1/2$$

, respectively. Initially the four ordered outcomes are equally likely. Given first coin H, only HH and HT remain. Copying one fair bit instead allows only 00 and 11, each with probability

$$1/2$$

. Both positions are balanced individually, but their joint probability of 11 is not the product

$$1/2 \times 1/2$$

, so they are not independent.

2. **The probabilities are**

$$1/4$$

,

$$14011/65536$$

and

$$1/16$$

. Four distinct guesses cover four of sixteen candidates. With independent uniform replacement, the complement of four misses gives

$$1 - (15/16)^4$$

, about 22.75 percent. Repeating one fixed guess against an unchanged secret covers only one candidate. These results rely on the stated uniform-key and information model.

3. **There are 24 no-repeat triples.** The choices are

$$4 \times 3 \times 2$$

, out of

$$4^3 = 64$$

equally likely ordered triples. Hence 40 triples contain a repeat, giving probability

$$40/64 = 5/8$$

. A collision allows any pair to match, whereas guessing a fixed target requires matching the particular value selected as the target. The

events and their probabilities must be specified separately.

4. **Shannon entropy is 1.75 bits, min-entropy is 1 bit and the best guess succeeds with probability**

$1/2$

. The weighted information sum is

$$(1/2)1 + (1/4)2 + (1/8)3 + (1/8)3 = 1.75$$

The largest probability is

, whose negative base-two logarithm is 1. An eight-bit field can contain only two possible strings, or another biased distribution, so its length alone does not establish eight bits of entropy.

5. **There are at most**

$2^8 = 256$

possible outputs. A deterministic function can give only one fixed-length result per seed. Its 64-bit output space has many more possible strings, but the function cannot reach them all from eight-bit seeds. This counting fact alone does not contradict computational pseudorandomness as a general concept. However, the actual eight-bit seed space is small enough to enumerate, so it gives no meaningful cryptographic resistance to a seed-search or output-membership attack.

6. **The roles are different.** The entropy source provides assessed uncertainty from a noise process and its associated mechanisms. A DRBG deterministically generates output and updates protected state. An RBG construction specifies how those components are combined and managed. Health tests address specified source behavior and failures; passing them is not a complete analysis of unpredictability, implementation security or application use.
7. **Future uncertainty requires an appropriate unknown input and update.** Restarting the same known state repeats a deterministic

process, and a predictable timestamp is not automatically fresh entropy. Recovery after compromise depends on the construction and on new input unavailable to the attacker. It cannot undo the exposure of keys or outputs that have already been learned.

8. **The distinguishing advantage is 0.10 and classification success is 0.55.** The probability gap is

$$0.10 - 0.50 = 0.10$$

. Under equal experiment probabilities and the better orientation, success is

$$1/2 + 0.10 = 0.55$$

. The alternative excess-over-one-half convention would report advantage 0.05, so the definition must accompany the number.

9. **The constructed attack's cost and the loss term's meaning must be specified.** A theorem must identify the scheme, attack model, underlying problem, advantage conventions, resources used by

$\mathcal{B}$

and the event or loss represented by

$\mathcal{E}$

. Without them the displayed bound cannot establish a concrete security level. The function

$$1/\lambda^2$$

is not negligible because, for example, it is larger than

$$1/\lambda^3$$

for every

$$\lambda > 1$$

.

Chapter 6: AES and ChaCha20

1. **AES-256 still uses a 128-bit block.** Its key has 256 bits, and it performs fourteen rounds with fifteen 128-bit round-key blocks, including the initial one. These blocks are deterministically derived from the original key. Their combined length is not a count of independent secret bits or independently generated keys.
2. **The ChaCha20 block function produces keystream from a key, nonce and counter.** It does not permute a 64-byte plaintext block. A 70-byte message needs two keystream blocks: all 64 bytes of the first and six bytes of the second. At the raw stream-cipher level, XOR produces 70 ciphertext bytes; unused keystream does not become additional ciphertext. Protocol framing and authentication can add separate overhead.
3. **The byte is 06.** The index rule gives

. AES fills columns downwards; filling across rows places many bytes in different positions. ShiftRows and MixColumns would then act on different groups, so using that layout without the required adjustments computes a different result.
4. **The reduced byte is 47.** The left shift of AE gives the nine-bit polynomial encoding 15C . XOR with the modulus encoding 11B gives 047 . Equivalently, 5C XOR 1B gives 47 . Ordinary doubling modulo 256 would give 5C , because it drops the high bit without the polynomial correction.
5. **The field product is FE.** The 10 , 02 and 01 terms of 13 give partial products 07 , AE and 57 . Their XOR is A9 XOR $57 = \text{FE}$. Zero has no field inverse, so the S-box definition explicitly sets its intermediate byte to zero before the affine step. The resulting S-box output for zero is 63 .
6. **ShiftRows brings bytes from different columns into the new first column.** Row zero stays fixed; the other rows rotate left by one, two and three bytes. The first column immediately after SubBytes was D4 27 11 AE ; after the row movements it is D4 BF 5D 30 . MixColumns

maps that column to 04 66 81 E5 through field products and XOR, as shown in the worked calculation.

7. **The carry leaving the 32-bit word is discarded.** The sum is reduced modulo

$$2^{32}$$

, so 14581472E becomes 4581472E. Carries inside the retained word still matter. XOR would give a different result, because it combines each pair of bits without any carry. Modular addition is one operation and XOR is another.

8. **There are eighty quarter rounds.** Ten double rounds each contain four column and four diagonal quarter rounds. The transformed state is then combined with the original state by wordwise addition modulo

$$2^{32}$$

, not XOR. Little-endian serialization of E4E7F110 gives bytes 10 F1 E7 E4. The number of secret key bits remains 256, rather than the 512-bit size of the complete working state.

9. **The vector checks the specified behavior on that input.**

Depending on what was compared, it can detect mistakes in rounds, feed-forward or byte order. It does not prove security for all inputs, constant-time execution, correct key generation, nonce and counter management, or message authentication. A complete construction and its implementation must address those additional requirements; the raw cipher has no message-acceptance tag check.

Chapter 7: Nonces and Authenticated Encryption

1. **The first and third ciphertext blocks are identical.** A fixed-key AES permutation gives the same output for the same input. The middle block differs when

$$P_1 \neq P_2$$

. A longer key does not remove this equality pattern; the problem is independent processing of repeated blocks in ECB.

2. **A predictable sequence can satisfy uniqueness if it is managed correctly.** The complete nonce must not repeat for an encryption under the same key. Shared-key senders must coordinate their spaces or use appropriate separate keys. Restoring an old key together with an old counter can repeat earlier pairs. A counter also requires safe allocation under concurrency and a rule for exhaustion; it is not automatically a persistent uniqueness guarantee.
3. **The sender field has 32 bits and the counter field 64 bits.** Four big-endian bytes 00 00 00 07 encode sender 7. Eight big-endian bytes 00 00 00 00 00 00 01 02 encode counter 258. Their concatenation has twelve bytes. Each integer must fit in its field, and sender assignments and counter histories must prevent repetition under the key.
4. **Associated data do not independently select the message keystream in these constructions.** Repeating the key, nonce and relevant internal counters repeats it even if the header changes. For the displayed positions,

$$M = H(IV, A, C)$$

. XOR of the ciphertexts is 00 00 00 00 08 00 00; combining that with PAY=100 gives PAY=900. Knowing only the XOR relation would not identify every possible message pair uniquely.
5. **The header is authenticated but remains unencrypted.** Its bytes must match during verification, but the receiver must still check that its conversation, direction or operation belongs to the expected context. A valid record carrying an old context label does not become appropriate for a new context just because it verifies under a shared

key. Authentication also does not conceal sensitive values placed in clear associated data.

6. **Both need thirteen and eleven padding bytes in this illustration.**

The padded associated-data and ciphertext parts each occupy sixteen bytes, followed by a sixteen-byte length block. GCM encodes bit lengths 24 and 40 as 64-bit big-endian integers. ChaCha20-Poly1305 encodes byte lengths 3 and 5 as 64-bit little-endian integers. Thus the formatted input sizes agree here, while the length-field bytes differ. This padding is for authentication and does not lengthen the ciphertext body.

7. **The three AES uses have separate roles within GCM.**

$$E_K(0^{128})$$

derives the GHASH subkey

$$H$$

. For the 96-bit nonce case, applying AES to the first incremented message counter block produces the first plaintext-masking stream block.

$$E_K(J_0)$$

masks the GHASH result for the tag. GHASH uses a field of

$$2^{128}$$

elements with its own representation and modulus; the AES byte-field rule operates in a different field of

$$2^8$$

elements.

8. **Counter zero derives the one-time key; counter one begins the message stream.** The first 32 bytes from block zero provide the two Poly1305 key parts, including the specified clamping of

$$r$$

. Accumulation is integer arithmetic modulo

$$2^{130} - 5$$

. Adding

$$s$$

and extracting the final tag uses reduction modulo

$$2^{128}$$

, followed by little-endian serialization. Neither step is the GHASH field calculation.

9. **Authenticity checking and replay policy are separate.** The unchanged inputs remain valid for AEAD, so the program accepts them twice. Its receiver has no stored rule marking the record number as already delivered. A changed tag fails the authentication check. On any verification failure, candidate plaintext must not be released or acted upon as authenticated data, and failure must not be confused with an empty valid message.

Chapter 8: Hashes and Message Authentication

1. **The digest has 32 bytes and 64 hexadecimal characters.** A byte has eight bits, so

$$256 / 8 = 32$$

; each byte takes two hex characters. The displayed one-bit change from `abc` to `abb` changes 124 digest bits. That is one checked pair, not a requirement that every pair change exactly 128 bits. Good diffusion and the three security goals are different questions.

2. **The attacker is given different starting information.** A preimage search starts with a target digest and seeks any matching input. A second-preimage search starts with a particular message and seeks a distinct message with its digest. A collision search can choose both distinct inputs. Under the stated ideal classical model, the many possible pairs lead to the square-root scale

$$\sqrt{2^{160}} = 2^{80}$$

for collision search. This is an order-of-work comparison with assumptions, not a prediction of a practical attack on every hash or every message length.

3. **Both toy results are decimal 131, hex 83.** ASCII `A` and `B` have values 65 and 66. Addition gives

$$65 + 66 = 131$$

, so the sum loses their order. Appending a zero byte also leaves the result unchanged. Determinism and a fixed output size do not prevent easy collisions or make this sum suitable for cryptographic use.

4. **Padding reserves a 64-bit length field at the end.** The original 24 bits plus one appended bit plus 423 zero bits total 448. The final eight bytes encode 24 as `00 00 00 00 00 00 00 18`, producing a 512-bit block. SHA3-256 instead uses a sponge with rate 1088 bits and capacity 512 bits. The permutation mixes all 1600 bits, while input and output steps directly use only the rate portion; its digest exposes just 256 output bits.
5. **The attacker can substitute both objects.** Public hashing lets them calculate the digest of their replacement file without a secret or collision. The secret-prefix example is a different attack: the known

digest exposes SHA-256's chaining state after its original padding. Continuation produces a tag for a message that includes that padding,

$$M||P||X$$

, under a correct length guess. It does not directly give a tag for

$$M||X$$

, and a particular application may reject the embedded padding bytes.

6. **The twenty-byte key is extended by forty-four zero bytes.** Thus

$$K_0$$

is 64 bytes; appending the eight-byte message to the inner keyed block gives 72 bytes. SHA-256 returns

$$I$$

as 32 bytes, so the outer keyed block followed by

$$I$$

gives 96 bytes. Its hexadecimal display would instead add 64 ASCII bytes, giving a different 128-byte outer input. The layers use raw digest bytes.

7. **The public collision and keyed forgery are distinct tasks.** A collision from the hash's public initial state need not persist after the unknown HMAC keyed block. Research analyzes the specific keyed construction under stated assumptions, including conditions needed for HMAC's related key-derived inputs. The displayed

$$v/2^t$$

bound concerns blind guesses of ideal uniform

$$t$$

-bit tags; it omits actual construction-specific security terms, key weakness, implementation leakage and protocol errors.

8. **Both concatenations are the identical three bytes abc.** This is an encoding ambiguity, not two different hash inputs with equal digests. Separate update calls still process the same concatenation. The illustrated four-byte length prefixes distinguish the pairs, provided that the receiver follows the fixed field order, checks lengths and rejects incomplete or trailing data as required by that format.

9. **A stateless tag check can accept an exact replay.** HMAC itself does not require a random nonce for each tag. The application must authenticate and enforce its intended context and freshness mechanism, for example a sequence number bound to the correct sender and conversation with coordinated replay state. Merely adding a random value without defining its check and acceptance rules does not ensure that an old command is processed only once.

Chapter 9: Passwords and Key Derivation

1. **The trials cover probability 0.35, or 35 percent.** Add

0.35 + 0.00 + 0.00

for the three distinct candidate events. A deterministic 256-bit result does not change the probabilities of the original choices. With a fixed public salt and parameters, at most

D

possible password inputs give at most

D

possible derived outputs, regardless of the output field's larger representable space.

2. **The attacker can calculate candidates without contacting the website.** Its rate limit controls requests to that service, not the attacker's separate computer. A distinct salt makes the expensive calculation record-specific and frustrates a common precomputed table; it does not stop fresh trials for that known record. Verification must use the recorded salt and parameters to reproduce the original calculation. A new salt is appropriate when producing a new record.

3. **The hypothetical trials take 1000 seconds, or 16 minutes and 40 seconds.** Divide

10^6

candidates by

10^3

candidates per second. At most sixteen 64-MiB work areas fit into 1024 MiB before overhead under that stated allocation model. This gives neither a measured throughput nor a universal attack limit: bandwidth, computing resources, implementations and time-memory trade-offs remain unspecified.

4. **The total is 64 MiB, shared across the four lanes.** With

$4p = 16$

, the requested 65,536 one-KiB blocks already form the required multiple. Each lane therefore has

65,536 / 4 = 16,384

blocks, in four segments of 4096 blocks. Three passes means traversing and updating the prescribed work area three times, not allocating a separate 64 MiB for each lane or multiplying the salt length by three.

5. **It stores the algorithm, version, costs, salt and calculated verifier.** The salt and verifier are Base64-encoded in the illustrated record. If its publicly stored result were also the file-decryption key, someone who copied the record would obtain that key without guessing the password. Decimal version 19 corresponds to hex 13 for Argon2 version 1.3. The pass count is the separate $t=3$ field.
6. **Salt is the HMAC key argument; IKM is the message argument.** Extraction uses HMAC for processing suitable source material under its extraction assumptions, rather than for authenticating a sender with a secret salt. Its public parameters do not create secret entropy. HKDF has no deliberate memory-hard password cost, so an attacker can repeat it cheaply over likely passwords. An appropriate password-based KDF addresses that separate cost problem.
7. **Two 32-byte expansion blocks are needed.** The ceiling of
$$\frac{42}{32}$$
is 2. The first HMAC message has ten info bytes and counter byte 01, with an empty previous block. The second has all thirty-two raw bytes of
$$T(1)$$
, ten info bytes and counter byte 02, totaling forty-three. Keep all thirty-two bytes of
$$T(1)$$
and the first ten of
$$T(2)$$
to obtain the forty-two-byte published result.
8. **Labels bind the expansions to their specified uses.** Encoding the intended protocol, direction, purpose and, where required, length avoids deliberately deriving the same material for distinct roles. This relies on a sound schedule and security assumptions, not an absolute impossibility of finite-output coincidences. With otherwise identical inputs, the sixteen-byte output is the prefix of the thirty-two-byte

output; the requested length is not automatically included as a domain separator.

9. **Recompute from the successfully supplied password with the new settings and a fresh salt.** A coordinated record replacement follows successful old-record verification and policy checks. The old verifier alone is not generally convertible into the desired new verifier for the original password. Resetting an account establishes a new access credential; it does not reconstruct an encryption key derived from the forgotten password. Recovering encrypted data requires a separately designed and protected recovery capability, if one exists.

Chapter 10: Agreeing on a Secret

1. **There is one key for each unordered pair.** Counting

$$1000 \times 999$$

ordered pairs counts each relationship twice, giving

$$499,500$$

distinct pairwise keys. Other architectures have different counts and trust assumptions. A public-key directory changes which material can be distributed openly, but its key-to-owner associations still need integrity. Substituting an attacker's public key under the intended recipient's name can make correct encryption protect the wrong relationship.

2. **The modulus is 23, the subgroup order is 11, and the public shares are 18 and 6.** Group products are represented modulo 23; powers of the generator repeat modulo exponent period 11. Nora obtains

$$6^6 \bmod 23 = 12$$

, while Sam obtains

$$18^6 \bmod 23 = 12$$

. Their exponent product is

$$36 \equiv 3 \bmod 11$$

, and

$$2^3 \bmod 23 = 12$$

gives another check.

3. **Multiplication of shares adds their exponents.** Thus

$$18 \times 6 \bmod 23 = 12$$

represents

$$g^{a+b}$$

rather than

$$g^{ab}$$

. The exponent identity proves agreement for the intended correctly formed exchange, not passive secrecy or authentication. Enumerating the ten nonzero exponents in this group readily recovers the displayed secrets; its arithmetic is teaching material.

4. **Mallory computes 13 with Nora and 8 with Sam.** She obtains

and

, matching
 U^6

and

V^9

. Both replacement shares are valid subgroup elements. A MAC under either substituted new secret is also computable by Mallory, so neither membership nor such a MAC creates the missing trusted identity binding.

5. **Confirmation concerns matching keying material in the relevant exchange.** Its identity meaning depends on the authentication and context already established. Faithful relaying lets the intended parties carry out their authenticated conversation without necessarily giving the relay their secrets. Substituting shares to establish separate attacker-known secrets changes that relationship. A security model must distinguish these behaviors and account for multiple sessions and compromise.
6. **Encaps also returns the KEM shared secret.** The recipient derives a matching secret from its private key and the transmitted encapsulation under the construction's correctness conditions. Agreement alone does not establish secrecy: a deliberately insecure KEM could expose its secret in public. The basic interface generates secret material rather than encrypting a supplied application plaintext, and base encapsulation alone does not authenticate a particular sender.
7. **The encapsulation is public, while the KEM secret and AEAD key remain local.** The twelve-byte base nonce helps derive each record nonce and is not a separately transmitted field in the illustrated combined representation. The 29-byte plaintext gives 29 encrypted body bytes plus sixteen tag bytes, and adding the 32-byte encapsulation gives 77. Equal lengths for the encapsulation and KEM secret do not make them the same bytes or give them the same role.
8. **Base encryption needs only the recipient public key, and the single-shot decryption call keeps no replay history.** Another caller

can make a fresh valid base-mode packet, and reconstructing the matching context lets the same packet decrypt again. Sender authentication requires an appropriate credential-bound mode or enclosing protocol; one-time processing requires defined freshness and coordinated acceptance state. A valid tag does not supply those policies on its own.

9. **The example is public-key/symmetric hybrid encryption, not a post-quantum/traditional combination.** The second meaning uses multiple key-establishment mechanisms and an appropriate combiner, for example ML-KEM with X25519. Later compromise of the long-term HPKE recipient key permits use of recorded encapsulations to derive the old keys. A fresh and erased sender ephemeral key alone does not prevent that recipient-compromise attack.

Chapter 11: RSA and ElGamal

1. RSA is named after Rivest, Shamir and Adleman. A sender uses the recipient's public key to encrypt; the recipient uses the related private key and the specified decoding procedure to recover the message. The modulus is

$$61 = 53 \times 8233$$

. The least common multiple of 60 and 52 is

$$60 = 52 / 4 = 780$$

. Since

$$17 \times 413 = 7021 = 9 \times 780 + 1$$

,

413 is the modular inverse of 17. It is not the real-number reciprocal

.

2. The public powers give

$$61^2 = 3721 \pmod{8233}$$

, so

$$61^2 = 3721 \pmod{8233}$$

. Writing the recovered result as

$$m = 12 + 53h$$

and requiring residue 4 modulo 61 gives

$$h = 1$$

, hence

$$m = 65$$

. The proof also covers a multiple of a prime factor: both that input and its positive powers are zero modulo that prime.

3. The factors give

$$\lambda(n)$$

and then the inverse exponent. This proves a route from factoring to inversion, not a general proof of the converse. The leaked totient gives

sum

$$400 - 320 + 1 = 81$$

and product 3233. Solving the resulting quadratic gives factors 61 and 53.

4. Raw encryption is public and deterministic, so the observer encrypts both possible representatives and compares the results. Multiplying a ciphertext by

$$2^e$$

produces an encryption of

$$2m \bmod n$$

. For this example, ciphertext 3017 therefore recovers 130. Making that modification does not require learning

$$m$$

first.

5. The capacity is

$$|B| - 1 = 8 - 1 = 7$$

bytes. The seed is fresh random input to the encoding; the label is associated context whose hash is checked. The masked data block conceals the structured block under the seed-derived mask. The final ciphertext is the

$$256$$

-byte representation of the RSA result, not the seed or unmasked message.

6. The argument assumes RSA inversion difficulty for the specified keys and treats the hash-based masking through the random-oracle model. Its reduction is not tight. It establishes a conditional chosen-ciphertext secrecy result, not sender identity, replay prevention or forward secrecy under later compromise of the recipient's private key.

7. The public value is

$$p = 2^8 \text{ and } q = 19$$

. With

$$r = 3$$

,

$$C_1 = 8$$

, the shared factor is

$$16^8 \bmod 23 = 16$$

, and

$$C_2 = 16 \bmod 23 = 16$$

. Sam obtains the same factor from

$$8^8 \bmod 23$$

. The inverse is 16 because

$$16 \cdot 8 = 128 \bmod 23 = 1$$

, so

$$9 \cdot 16 \bmod 23 = 9$$

. Real-number division is a different operation.

8. The explanation requires a specified prime-order group, messages in that group, suitable independent uniform exponent sampling and the DDH assumption. Zero is not a member of the multiplicative group, and its second component would always be zero. Reusing the same mask exposes

$$C_2/C_1 = M_2/M_1$$

. For the example, the ciphertext ratio is 9 modulo 23, and knowing message 9 reveals the other message as

$$9 \cdot 9 \bmod 23 = 12$$

.

9. Changing the second component from 2 to 4 changes the recovered message from 9 to 18. Rerandomizing both components to

$$(16, 13)$$

preserves message 9 while changing the ciphertext. A distinct rerandomized challenge could be submitted to a decryption oracle to learn the original challenge message, so the bare scheme does not meet adaptive chosen-ciphertext secrecy.

Chapter 12: Elliptic Curves

1. **Coordinates identify a candidate position; the equation tests membership.** Substituting

$$(0, 1)$$

or

$$(1, 1)$$

gives equal sides on the real curve. Their line is

$$y = 1$$

, with third intersection

$$(-1, 1)$$

; reflection gives

$$(-1, -1)$$

. Coordinate-by-coordinate addition instead gives

$$(1, 2)$$

, which fails the curve equation.

2. **There are 17 field elements, 18 affine points and 19 group elements.** The last count includes the identity



, which has no affine coordinate pair. The inverse of

$$(5, 1)$$

is

$$(5, 16)$$

because

$$1 + 16$$

has remainder zero modulo 17. Here

$$(0, 0)$$

is not even on the curve, since its two sides would be 0 and 2.

3. **The sum is**

$$(10, 6)$$

and the double is

$$(6, 3)$$

. Distinct-point slope 2 gives remainders

$$-7 \equiv 10$$

and

$$-11 \equiv 6$$

. For doubling,

$$2^{-1} \equiv 9$$

because

$$2 \times 9 = 18 \equiv 1$$

. The tangent numerator is 77, so the slope is

$$77 \times 9 \equiv 13$$

; the coordinate remainders are 6 and 3. Handle identity inputs, inverse points and vertical-coordinate-zero doubling before attempting division.

4. Doubling builds

$$2P$$

,

$$4P$$

and

$$8P$$

; addition combines the selected terms. Here

$$8P + 4P \equiv (0, 10)$$

and adding

$$P$$

gives

$$(16, 4)$$

. Coordinate operations use the field modulus 17. Scalar actions repeat with the point's order 19. The full group also has 19 elements, so its cofactor is

$$19 / 19 = 1$$

.

5. The public points are

$$(9, 16)$$

and

$$(0, 6)$$

; the shared point is

$$(10, 11)$$

. Both parties obtain

$$35P = 16P$$

because 35 has remainder 16 modulo 19. Recovering a scalar from a public multiple is the elliptic-curve discrete logarithm problem.

Computing

abP

from

aP

and

bP

is computational Diffie–Hellman. Solving the former supplies a way to solve the latter; no general equivalence is claimed.

6. **The inputs are scalar bytes and an encoded**

$\mathcal{U}$

-coordinate; the output is an encoded

$\mathcal{U}$

-coordinate. Each has 32 bytes. Public-value generation uses base input 09 followed by 31 zero bytes, with the least significant byte first. Scalar decoding clears three low bits, clears the highest bit and sets the next-highest bit, giving

$$2^{254} + 8t$$

with

$$0 \leq t < 2^{25}$$

. Encoding length does not determine attack cost; the RFC describes a classical level slightly below the conventional 128-bit level.

7. **SEC 1 compression identifies a full point; X25519 uses its specified coordinate interface.** For the odd-prime example, a SEC 1 prefix distinguishes the vertical coordinate's parity and the horizontal coordinate uses the specified big-endian width. X25519 uses 32 little-endian coordinate bytes without that prefix or parity bit. Its input reduction, bit masking and twist handling must not be replaced by a generic full-point acceptance checklist.

8. **An all-zero raw result can remove the expected secret contribution.** Its predictability is not repaired by HKDF. RFC 7748 describes the output check as optional; a protocol may require rejection, and the installed library rejects the exchange. A 32-byte zero input can pass the constructor's length check and fail at exchange; a 31-byte public input fails construction. These are distinct interface stages.

9. **Agreement does not establish the participant's identity.** An intermediary can substitute their own public values unless the protocol supplies the required authentication. Binding public values and roles in a derivation does not by itself provide that authentication. Serialization maps an existing point to a specified byte representation; hash-to-curve maps arbitrary bytes to a point under a specified construction, without being reversible file encryption.

Chapter 13: Signatures, Certificates and Identity

1. **Signing uses the private key and message to produce a signature; verification uses the public key, message and signature to accept or reject.** A signature does not conceal the message. Unlike a shared-key MAC, it separates private signing power from public checking power. Trust in the public key remains necessary.
2. **Ordinary chosen-message unforgeability concerns an accepted signature on a message not previously requested from the signer.** A changed accepted signature for a previously signed message concerns a stronger security definition or a representation policy. Exact replay is neither of those events and normally passes mathematical verification again.
3. **The public value is 18, the commitment is 8 and the response is 2.** Calculate

$$2^2 \bmod 23 = 4$$

,

$$2^2 \bmod 23 = 4$$

and

$$(8 \times 18) \bmod 23 = 4$$

. Verification gives

$$2^2 \bmod 23 = 4$$

and

$$\begin{aligned} &8 \times 18^9 \bmod 23 \\ &= 8 \times 12 \bmod 23 = 4 \end{aligned}$$

.

Group products use modulus 23, while scalar responses use subgroup order 11.

4. **The challenge binds the public inputs to the exact message under the specified encoding.** Accepting a challenge chosen

independently by an attacker would omit that required relationship. The correctness equality explains why honest calculations agree; it is not an unforgeability proof. Our tiny group and eleven-value challenge permit easy searches.

5. **PSS prepares the hashed message and salt before the RSA private power; verification checks its specified encoding after the public power.** OAEP and PSS have different purposes and formats. The 190-byte OAEP encryption capacity does not limit the message length for PSS signing, although the PSS block still has its own capacity conditions.

6. **The inverse is**

$$w = 10$$

modulo subgroup order 19. The coefficients are

$$17 \equiv 10 \text{ and } 10 \equiv 18$$

and

$$18 \equiv 18 \text{ and } 10 \equiv 5$$

. The resulting point is

$$16G + 5Q = 41G = 3G = (10, 6)$$

,

with horizontal coordinate matching

. Modulus 17 is for coordinate-field operations; the response and its inverse belong to scalar arithmetic modulo 19.

7. **The seed is the private input; hashing and bit processing derive a scalar and secret prefix; the public point is a scalar multiple of the base point.** The signature contains an encoded point and response scalar, not the seed or prefix. Plain Ed25519 applied to digest bytes does not implement Ed25519ph, whose prehash and variant processing are specified separately.

8. **The recovered scalar is 6.** The response difference is

−3

and the challenge difference is 5. Its inverse modulo 11 is 9, giving

$-3 \bmod 11 = 9$

. A specified deterministic procedure derives the per-message value from private material and the message. A fixed constant used across different messages omits that derivation and causes the demonstrated cancellation.

9. **A trust anchor is trusted information configured as an input to validation.** A self-signature does not establish the authority of an otherwise untrusted key. Path validation checks certificate relationships and constraints against the selected anchor; service-name matching checks whether the leaf represents the intended service. A valid path for `files.example` does not establish the requested `shop.example` identity.

10. **A signature alone does not establish truth, authorization, freshness, human intent or independent creation time.** A correctly signed older release can contain known vulnerabilities. Preventing rollback needs an enforced release policy or trusted state in addition to the signature check.

Chapter 14: Managing Keys

1. **A cryptographic generation interface uses the selected system's protected random-generation facilities and required algorithm processing.** A timestamp can be predictable even after hashing. Output length alone does not establish the uncertainty of its source. A generation failure must not trigger an undisclosed fixed-key fallback.
2. **The inventory records identity, purpose, algorithm, owner, state, permissions, usage periods and protected dependencies.** An ordinary inventory should refer to protected key storage. Collecting raw secret keys in it would create another highly sensitive store rather than a simple management record.
3. **A public key needs a trustworthy identity and purpose association; a secret data key also requires confidentiality.** Both can require integrity and availability protection. An attacker can replace an untrusted key and its adjacent fingerprint together. An independent trusted comparison or another defined trust process is needed.
4. **Extraction and use are different capabilities.** A compromised caller might request unauthorized signing or decryption even if the module never exports the raw key. Operation permissions, caller authentication, application context and administrative controls must protect that capability.
5. K_D
protects the file, and
 K_W
protects
 K_D
.
 C
is the ciphertext/tag output,
 W
the wrapped key,
 N
the file nonce and

A

its associated data. Rewrapping changes

W

and the wrapping-key version while retaining the same

K_D

. The file can remain unchanged only when its authenticated inputs remain unchanged.

6. **The originator period permits applying new protection; the recipient period permits processing earlier protection.** Retained data may need an old decryption key. Earlier signatures are verified using public material, so that need does not justify retaining a retired private signing key. The allowed treatment depends on role and policy.
7. **Revocation changes authorization and trust decisions, not the cryptographic function on copied inputs.** A copied key can still decrypt a matching copied ciphertext. A compromised credential or host that can obtain the replacement material can also defeat a transition unless that access route is addressed.
8. **Recovery needs the file ciphertext/tag, nonce, exact associated data, wrapped data key, correct wrapping-key version and authorized access to it.** Algorithm and metadata interpretation must also be available. Verification of earlier signatures uses the public verification key and required evidence, not the private signing key.
9. **Split knowledge limits what one participant knows; dual control limits what one participant can authorize alone.** They are different properties and require defined mechanisms. Cryptographic erase removes required secret material to make encrypted data infeasible to recover under stated conditions. Copies, backups, runtime memory and derivation paths can survive deletion of one file or variable.
10. **The exported wrapping key can open every matching wrapped data key available in the copied database.** Offline use may leave no later central-service log entry. Rewrapping the same data keys preserves values the attacker may already know. Appropriate fresh data keys can protect future material, but cannot erase past disclosure.

Chapter 15: Securing a Connection with TLS 1.3

1. **The handshake negotiates parameters, establishes secret material and performs authentication; the record protocol protects successive communication units.** Handshake keys become available before all server authentication checks finish. An active intermediary could establish its own agreement with the client, so successful decryption alone is insufficient.
2. **The suite selects AES-128-GCM and SHA-256 for the specified record and handshake functions.** It does not select the certificate type or agreement group. TLS 1.3 does not use the old RSA key-transport construction; an RSA certificate can instead support a handshake signature.

3. M_j
is the encoded handshake message at position

j

,

||

joins byte strings,

H

is the selected hash and

T_j

is its output through that stage. The transcript covers handshake encodings, independently of record fragmentation and framing. HelloRetryRequest replaces the first ClientHello's direct transcript representation with the specified synthetic hash-containing message.

4. **Path validation checks the certificate relationships against trust anchors; name matching checks the intended service.**

CertificateVerify signs the structured transcript-related input with the certificate's private key. Finished checks the handshake transcript using derived secret material. SNI tells a server which name is requested; sending it does not verify that identity.

5. P

is the selected PSK or defined zero input,

Z

the agreement output,

E

the early secret,

G

the handshake secret and

M

the main secret. The intermediate salts and extraction stages remain present without a PSK.

ϵ

contains zero bytes;

$H(\epsilon)$

is the hash output of that empty string and normally contains nonzero byte values.

6. **Both initial application traffic secrets use the transcript through the server Finished.** The resumption-secret derivation uses the later transcript through the client Finished. Availability of a derived secret does not override state-machine, authentication or sending rules.
7. **The first padded counter is zero, so its nonce equals the fixed IV.** For the next counter, XOR changes the corresponding low-order byte. Resetting a counter under the same key and IV would repeat nonces. Algorithm-specific usage limits can require changing keys or closing before a 64-bit counter would wrap.
8. **The programs reproduce selected published derivations and open one public application record.** Altering the tested ciphertext/tag produces an authentication failure. The programs do not implement a TLS state machine, negotiate a connection, validate a certificate or service identity, parse all allowed records or prove conformance to the current specification.
9. **PSK-only uses existing secret material without a fresh asymmetric contribution; PSK with ephemeral agreement adds one.** 0-RTT protection is derived before the new server contribution and does not gain its forward secrecy. A captured early flight can be

replayed without forging its tag. Application duplicate handling and deployment-wide acceptance state determine what repeated processing can mean.

10. **A known current traffic secret allows its deterministic future update secrets to be calculated.** KeyUpdate alone introduces no independent secret. A proof establishes a property of its modeled protocol under assumptions; a trace checks selected inputs and outputs. Neither establishes every implementation path, endpoint configuration or application decision.

Chapter 16: Secure Messages and Changing Keys

1. **TLS to a delivery server protects a channel ending at that server.** The server can normally read the plaintext at its endpoint unless a separate message layer protects it for the recipient. Public prekeys let a sender calculate the setup while the recipient is offline; the matching private values remain at the recipient.
2. **Identity keys support an endpoint's longer-lived identity association; signed prekeys provide checked setup values; one-time prekeys add single-exchange material; ephemeral keys are generated for the current exchange.** Removing a public prekey from the server's available pool is not deletion of its private counterpart. The recipient still needs that private value to process the selected delayed setup.
3. **Nora calculates the left side of each line with her private input and a received public value; Sam calculates the matching right side with his own private input.**

Z_1

combines Nora's identity and Sam's signed prekey;

Z_2

her ephemeral and his identity;

Z_3

her ephemeral and his signed prekey.

Z_4

combines her ephemeral with his optional one-time prekey and is omitted when that prekey is absent.

4. **A signature verifies an association under the identity key used for verification, not that the supplied identity key belongs to the intended person.** An attacker can sign substituted prekeys under a substituted key. On the classical path without a one-time prekey, an initial message can be replayed and reproduce the same initial secret. The following protocol must handle replay and fresh reply-key requirements.

5. C_j

is the current chain key and

K_j

the derived message key. The constants are distinct single bytes separating output roles. A copied

C_j

allows the same deterministic future steps. Recovering earlier erased values from a later state should remain infeasible under the construction's assumptions; simply printing distinct outputs is not a proof.

6. **The root chain incorporates new agreement outputs; sending and receiving chains derive keys for their respective message directions.** On a new peer ratchet value, the old local private value first produces the matching receiving chain. A newly generated local private value then produces the next sending chain, with its public value advertised to the peer.
7. **Receiving position 2 first requires retaining the skipped keys for positions 0 and 1 in the appropriate chain.** They are consumed when their messages successfully arrive. Chain identity and number distinguish lookup entries, and bounds limit retained state. Failed authentication must discard candidate changes, including attempted skipped-key consumption and chain advancement.

8. **The copied**

a_1

and

R_1

let the attacker calculate the

a_1, B_1

output and follow the next root step. Later

a_2

and

b_1

are unknown in the specified passive scenario. Correct agreement and derivation can then restore protection for the new chain. Continued endpoint access, active substitution, weak fresh randomness, other exposed private inputs or incomplete erasure can defeat the intended recovery conditions.

9. **PQXDH starts the session with a conventional and post-quantum contribution; SCKA supplies a sequence of continuing shared secrets.** ML-KEM Braid is one SCKA construction; the Triple Ratchet combines classical and sparse post-quantum message-key contributions through a specified KDF. One initial post-quantum secret does not alone provide later post-compromise recovery. A message can advance an existing chain while a sparse agreement is still incomplete.
10. **These are additional state and observation paths.** Retained skipped keys can expose their delayed messages; local history can retain plaintext after old keys are erased. Backups can preserve secrets or restore old key state. Additional devices receive authorized copies, and metadata can remain observable outside the protected body. None is resolved solely by authenticating one ciphertext.

Chapter 17: How Good Cryptography Fails

1. **Length is only one input condition.** A valid-length key can be predictable, and a nonce can repeat under the same key. The collision approximation assumes independently uniform choices and a small collision probability. Cloning a machine can duplicate generator or counter state and invalidate that model.
2. **An oracle supplies a useful answer about an input.** A chosen-ciphertext attack is adaptive when later queries depend on earlier replies. A one-bit formatting distinction can progressively constrain a hidden value when an attacker chooses related queries; it need not disclose an entire secret in one response to be useful.
3. **Multiplying by a public RSA power needs no private key.** The attacker can form the related ciphertext using the public modulus and exponent. A conforming PKCS #1 v1.5 block starts with the required bytes and therefore lies in the stated interval, but an interval member can still fail the nonzero padding or separator conditions.
4. **The inclusive bounds are 250 and 259.** The lower calculation is

$$\lceil (200 + 2 \times 1147) / 10 \rceil \\ = \lceil 249.4 \rceil = 250$$

;

the upper is

$$\lfloor (299 + 2 \times 1147) / 10 \rfloor \\ = \lfloor 259.3 \rfloor = 259$$

.

Ten teaching candidates remain. The query has not identified which one is the hidden value or demonstrated a complete real-format attack.

5. **CBC combines bytes by XOR; the RSA example multiplies residues.** Changing a preceding CBC byte changes the corresponding plaintext byte through the displayed relation. The one-byte 01 example uses a PKCS #7-style convention. Historical TLS CBC uses its own padding-length convention, so the teaching byte cannot be imported unchanged.

6. **The model separates baseline, signal and noise.**

T_0

is the baseline,

δ

the conditional timing contribution,

b

the hidden 0/1 condition and

ϵ_i

the noise. Under the given independent equal-variance assumptions, averaging reduces the uncertainty in a mean. The example's five-standard-deviation gap is model arithmetic, not an observed success probability or a key-recovery experiment.

7. **Blinding introduces a removable random factor.** The private operation on

$c r^e$

yields the original plaintext multiplied by

r

modulo

n

; multiplying by

r^{-1}

recovers the original plaintext. The required fresh invertible choice and implementation conditions still matter. Added delays may be averaged

out, while branch-free source can still contain secret-indexed accesses or compile to variable-time instructions.

8. **Both pairs produce exactly the same unframed bytes.** There are no two distinct hash inputs in that example, so it is not a hash collision. Framing separates fields, canonicalization selects a required representation, domain separation distinguishes uses, and authorization determines which accepted operation the caller may perform.

9. **The failure involved implementation timing.** For

$t = 833$

, the numerator is 3330, its quotient by 3329 rounded down is 1, and the output bit is 1. Correct arithmetic does not imply that executed instructions take secret-independent time. The studied Kyber versions and platforms must be distinguished from all present ML-KEM implementations and from the lattice assumption itself.

10. **Each result has a limited scope.** A round trip checks selected functional behavior; a detector checks its modeled misuse patterns; a correction addresses a particular historical path. The actual binary, settings, reachable interfaces, state exposure and leakage model still need appropriate examination. None of these pieces alone certifies every application path.

Chapter 18: Testing and Verified Software

1. **The two mistaken operations can undo each other.** For the teaching example,

$S(10) = 11$

,

$F(10) = 12$

and

$S(F(10)) = 12$

. Recovery succeeds even though the forward result is not the intended one. A known-answer comparison with 11 exposes that disagreement.

2. **The expectation needs a distinct basis.** A published vector or independently checked specification supplies something other than the tested program's own earlier output. Two wrappers can call one implementation or share a mistaken representation, so agreement between their results need not provide independent confirmation.

3. **Each uniform draw misses the single revealing input with probability**

$255/256$

. Under independent draws with replacement, multiplying that probability 256 times gives about

0.367

. The fraction and sampling rule belong to the stated teaching model. They do not estimate the likelihood of an unknown attack on a real library, and an attacker may choose the revealing boundary deliberately.

4. **The fields have different roles and encodings.** The nonce selects the invocation, the associated data are authenticated clear inputs, and the protected result contains ciphertext bytes plus a tag. A harness must match the chosen interface's packaging. Ninety-six bits are twelve bytes, not ninety-six bytes; a hexadecimal display has two characters per byte.
5. **The denominator and expected policy must remain visible.** Invalid inputs require the relevant rejection, acceptable cases need their documented interpretation, unsupported operations cannot be

called successful, and excluded profiles are outside the run. The 513 outcomes establish behavior for the selected data and installed interfaces: 372 valid cases matched and 141 invalid cases were rejected. They do not cover every Wycheproof case or platform.

6. **Every permitted input satisfying the precondition must have the specified result.** The domain identifies which objects the claim covers, and the precondition identifies the required initial conditions. If the specification encodes the wrong algorithm or omits a desired requirement, agreement with it does not supply the missing property. Stateful claims must also model state and effects.
7. **The last valid index is 15.** Allowing index 16 goes beyond a sixteen-element array. Memory safety also depends on valid object lifetime, access permissions and related references; a numerical bound is one obligation within that larger model.
8. **The properties answer separate questions.** Functional correctness concerns the specified computation; memory safety concerns modeled memory rules; secret independence concerns defined extra observations; cryptographic security concerns protection under an attack model and assumptions. Repeating a nonce or ignoring an authentication failure can invalidate application security while a primitive computes its specified function correctly.
9. **Agility changes algorithms; multiplexing changes implementations of one algorithm.** A scalar and a vectorized implementation may calculate the same specified function. The remaining trusted chain can include the specification, checking tools, compiler behavior, assembly/linking models, hardware and caller contracts. The exact dependencies must be identified for the actual result.
10. **Their conclusions have different scopes.** The AEAD program compares one complete published result and five rejected inputs. REA inspection identifies selected digest/key/signature decisions and test definitions, without executing its complete suite. A binary-identification result proposes which function is present under the analysis model. None alone establishes safe application state,

configuration, authorization, all execution paths or the absence of unmodeled leakage.

Chapter 19: Quantum Computers and Post-Quantum Cryptography

1. **Amplitudes describe the state; squared magnitudes give probabilities in the selected measurement basis.** The number

$$i$$

satisfies

$$i^2 = -1$$

. The squared magnitude of

$$(3 + 4i) / 5$$

is

$$(3^2 + 4^2) / 5 = 1$$

. A measurement returns an outcome according to the distribution, not the complex coefficients or every basis value.

2. **The primed amplitudes are the outputs of the Hadamard matrix.**

$$H$$

names the gate,

$$\alpha, \beta$$

are input amplitudes, and

$$\alpha', \beta'$$

are outputs in the same basis order. With

$$(\alpha, \beta) = (1, 0)$$

, both outputs are

$$1 / \sqrt{2}$$

. Applying the rule again gives

$$(1, 0)$$

because the first row adds those equal amplitudes and the second subtracts them.

3. **The inserted**

$$Z$$

changes plus to minus before the final interference. Two Hadamard gates return

$$|0\rangle$$

. Inserting

Z

reverses the sign of the

$|1\rangle$

component, and the final Hadamard returns

$|1\rangle$

. A global sign reverses all amplitudes and leaves subsequent squared-magnitude predictions unchanged; a relative sign can alter those predictions.

4. Hadamard on the first qubit gives

$(00+110)/\sqrt{2}$

, then CNOT maps the second term to

$|11\rangle$

. The resulting Bell state gives equal-string outcomes. A shared classical random bit can imitate those counts in one basis; other settings distinguish the states. Local outcomes remain random, so an observer cannot choose and transmit an outcome faster than light by measuring one side.

5. It evolves a four-entry classical amplitude list. The order is

$00, 01, 10, 11$

, and the index

$2c + t$

places control bit

c

on the left and target bit

t

on the right. The output probabilities are the squared magnitudes,

$[0.5, 0, 0, 0.5]$

. It is an ideal local calculation with no shots, device noise, network or account. Qiskit's displayed qubit-index order must be checked separately.

6. The host describes the task, control electronics drive operations, the cold processor carries physical quantum information, and

readout produces records. A family name can span different revisions. Physical counts omit connectivity, operation quality, error correction, logical resources and required runtime. Several processors in one system do not automatically form one freely connected logical register.

7. The expression counts the chance of no specified independent error event.

p

is the event probability per operation,

G

the number of operations and

P_{none}

the no-event probability. The example gives

$$0.999^{1000} \approx 0.368$$

. A real observable can be affected differently by different errors, which may also be correlated or systematic; the model is not a measured success probability for a named processor.

8. A shot is one execution and measurement; an expectation value is a probability-weighted mean. With values

$+1$

for zero and

-1

for one, the sample mean is

$$(0.001 - 0.999) / (0.001 + 0.999) \approx -0.999$$

. Sampling variation is the fluctuation of finite estimates; bias is a systematic difference between the expected estimate and the intended quantity. Repetition cannot by itself repair a persistent model or calibration error.

9. Each independent component contributes its mean's variance multiplied by the square of its weight. The symbols

w_1, w_2

are weights,

μ_1, μ_2

true means and

N_1, N_2

$$a_j$$

, giving

$$(0, 0, 0, 1)$$

and the same final probability list. That exact one-step outcome is specific to four candidates with one marked solution. The general ideal single-solution search has square-root query scaling,

$$2^{k/2}$$

for

$$2^k$$

keys, with additional circuit costs. Shor instead exploits factoring/discrete-logarithm structure with polynomial-time quantum methods.

13. **The published results are 14 and 16.** The calculations are

$$1+3+2+5+1=12$$

and

$$4+3+1+5+1=14$$

. In the second row the noiseless sum is 17, so the signed residual is

$$16 - 17 = -1$$

. Its modular representative is 16; centering gives

$$(16 - 17) \bmod 17 = -1$$

.

14. **The matrix contains public coefficients, the secret vector unknown coordinates, the error vector perturbations and the result vector published residues.** The modulus

$$q$$

specifies the arithmetic. Hardness requires defined dimensions, distributions, parameters and an attack model. Neither noise alone nor a difficult-looking drawing establishes those conditions.

15. **The program exhausts a disclosed two-coordinate space of**

$$17^2 = 289$$

pairs. Exactly one pair satisfies the four chosen residual conditions. This is a deliberately solvable bounded-error search example, not a standardized parameter. Regev's reduction connects suitable LWE settings to specified lattice problems under conditions; it does not

prove unconditional hardness or automatically cover every structured module scheme.

16. **ML-KEM establishes secret material; ML-DSA and SLH-DSA provide signatures with different design foundations.** A shared secret is not public signature evidence. Its 32-byte length is 256 bits, but security also depends on the underlying problem, parameter set, transformations and adversary model. The length alone does not establish the entire mechanism's security level.
17. **A specified internal consistency failure produces replacement secret material instead of an ordinary flag for that condition.** This neither authenticates a peer nor accepts an application message. Input validation and subsequent authenticated processing remain necessary. A wrong-length ciphertext is a separate interface case.
18. **The first combines post-quantum and traditional establishment contributions; the second combines public-key establishment with symmetric data protection.** A system can use both. Robustness needs an exact combiner, encodings, component handling, context and protocol argument. An arbitrary function named KDF does not establish the required guarantee when a component fails.

19. S
is secrecy lifetime,
 M
migration duration and
 Q
an assumed time to a relevant attack. Continuing old protection through migration can leave its latest data requiring secrecy until roughly
 $M + S$
. The relation flags a scenario, not a forecast. New keys or local re-encryption cannot remove ciphertexts and transcripts already copied by an attacker.

20. **Their evidence has different scope.** A standard defines a mechanism, a draft proposes material, a benchmark reports selected experiments, source inspection examines specified code paths and a

forecast estimates resources under assumptions. Qedma's mitigation work and the researchers' architecture and fault-tolerance contributions should be assessed through their specific methods, results and conditions. They are not interchangeable proof of every product claim or a completed cryptographic attack.

Chapter 20: Proving and Computing with Private Data

1. The public data are

$\mathcal{X}$

, the private witness is

w

, and

R

checks their specified relation. The prover supplies evidence and the verifier checks it. A digest-preimage relation establishes that the encoded witness hashes to the given digest. It does not establish factual truth, ownership or the intended account/time association unless those claims are included and supported by the relation and its trusted inputs.

2. **Completeness concerns honest acceptance; soundness limits acceptance of invalid claims; zero knowledge concerns the verifier's information.** Knowledge extraction gives a witness-recovery property under the definition's conditions. It is not an ordinary verifier capability against every honest user. Public inputs and the proved condition remain available for inference.

3. **The public value is 13, the first message 16 and the response 3.** Calculate

$$g^2 \bmod 31 = 10$$

,

$$g^2 \bmod 31 = 10$$

and

$$(g^2 \cdot 3) \bmod 31 = 8$$

. The check has left side 8 and right side

$$16 \cdot 3 \bmod 31 = 8$$

. Exponents wrap modulo group order 11, while group residues are represented modulo prime 23.

4. **The simulator obtains**

$$T = 16$$

. With

$$c = 3, z = 3$$

, the response power is 8 and

$$Y^c$$

is 12 modulo 23. The inverse of 12 is 2, so

$$7 \cdot 6 = 12 \text{ and } 12 \cdot 2 = 24$$

. Choosing the whole transcript in a different order explains its distribution for the stated honest verifier. A malicious verifier's choices and concurrent interaction require additional analysis.

5. **The extracted witness is 7.** The response difference is

$$7 - 3 = 4$$

, and the challenge difference is

$$4 \cdot 10 = 40 \text{ and } 40 \cdot 10 = 400$$

. The inverse of 10 is 10, giving

$$4 \cdot 10 \text{ and } 11 \cdot 7$$

. Reusing the same secret first-message randomness across those challenges exposes the witness through this relation.

6. **Their labels concern different properties and models.** Proofs and arguments conventionally differ in whether soundness is information-theoretic or computational; succinctness concerns evidence and verification costs; zero knowledge protects the witness. A SNARK is an argument of knowledge with the specified compact non-interactive properties. Setup, public-input binding and correct circuit constraints still determine what is actually established.

7. **The share values are**

$$2, 7, 7, 2, 9$$

at identifiers 1 through 5. Substitute each identifier into

$$9 = 4X + 6Y$$

and reduce modulo 17. Zero exposes

$$f(0) = 9$$

, while repeated identifiers do not provide distinct interpolation points. Equal values at different identifiers are possible and remain different points.

8. **The weights are 4, 3 and 11.** For identifier 1 use numerator 15, denominator 8 and inverse 15. For identifier 3 use numerator 5, denominator 13 and inverse 4. For identifier 5 use numerator 3, denominator 8 and inverse 15. The weighted sum is

$$4 \times 2 + 3 \times 7 + 11 \times 9 = 128 \\ \equiv 9 \pmod{17}$$

9. **For each candidate secret, a unique coefficient pair fits the two observed points.** Independent uniform coefficients make the observation equally likely for every candidate secret, preserving the prior secret distribution. This does not authenticate a share, compel truthful dealing, prevent a qualified coalition or guarantee availability.
10. **It checks the ten three-share subsets of one disclosed example and specified input-shape errors.** Replacing

(1, 2)

with

(1, 3)

while using identifiers 1, 3 and 5 gives

$$4 \times 3 + 3 \times 7 + 11 \times 9 = 132 \\ \equiv 13 \pmod{17}$$

The altered points remain well formed, so ordinary interpolation returns a different secret rather than detecting dishonesty.

11. **Reconstruction assembles a complete key; threshold signing computes a signature with distributed private material.** FROST needs fresh protected nonce state, agreed message/commitment binding and the specified participant checks. A dealer and a distributed key-generation protocol have different trust arrangements. The threshold limits both compromise and availability; a single signature does not by itself display every institutional approval rule.

12. F
is the agreed function, the

$$x_i$$

are private inputs and

$$y$$

is the permitted result. The output policy names recipients. Semi-honest parties follow steps; malicious parties may deviate. Correctness with abort does not imply fairness or guaranteed delivery: a party may be unable to forge an accepted output but still disrupt its release.

13. **The input sharings reconstruct 4 and 7, and their local sums reconstruct 11.** The residues are

$$6 + 15 \equiv 4$$

,

$$9 + 15 \equiv 7$$

and

$$(6 + 9) + (15 + 15) \equiv 11 \pmod{17}$$

. Local multiplication gives

$$(6 + 9)(15 + 15) \equiv 7$$

, omitting cross terms, whereas the desired product is

$$28 \equiv 11$$

.

14. **Expand**

$$(a + d)(b + c)$$

and substitute

$$c = ab$$

. The result is

$$c + ab + ac + bc$$

. The displayed values give

$$15 + 5 + 6 + 2 = 28 \\ \equiv 11 \pmod{17}$$

.

Reusing a mask reveals differences between private inputs. Protected valid triples, authenticated sharing and malicious-consistency checks are absent from this elementary arithmetic; they are essential parts of the full SPDZ construction and its abort model.

15. pk
is public encryption material,
 sk
the protected decryption key, and
 evk
the specified evaluation material. Each
 m_i
 is an input and
 C_i
 its ciphertext;
 F
 is the computation,
 C_F
 its encrypted result and
 $\tilde{y}$
 the decoded output. Correct arithmetic modulo 17 maps
 $12 + 9$
 to 4, while the intended unrestricted integer sum is 21. Domain bounds and encodings decide which result is meaningful.

16. **The encodings are 1234 and 2001 at scale 1000.** Their decoded inputs are 1.234 and 2.001, with errors

. The product model gives 2.469234, then 2.469 after the displayed scale reduction. The original product is 2.47040472, so the final absolute error is 0.00140472. Actual CKKS includes ciphertext arithmetic, encoding and additional error sources beyond this plain-number model.

Key Terms

These short explanations help you return to the main discussion. They cover the material developed in Chapters 1–20. A term’s precise conditions remain with its construction; a reminder here does not replace those conditions. Follow the section reference for the worked explanation.

0-RTT.

Early application data sent before receiving the current connection’s server flight. Its PSK-based protection has weaker replay and forward-secrecy guarantees than ordinary fresh-agreement traffic. See Section [15.6](#).

AEAD.

Authenticated encryption with associated data. A construction that protects message secrecy and checks protected data, including specified clear inputs. The receiver obtains accepted plaintext or a failure result. See Section [7.4](#).

AES.

Advanced Encryption Standard. A secret-key block cipher that transforms sixteen-byte blocks using a selected key. AES-128, AES-192 and AES-256 name different key lengths, not different block lengths. See Section [6.3](#).

Agility.

Support for choosing between algorithms that serve a functionality. It differs from choosing several implementations of one algorithm. See Section [18.4](#).

Algorithm.

A set of exact steps for a task. A cryptographic algorithm’s description can be public while the key or other required secret inputs remain protected. See Section [2.2](#).

Amplitude.

A coefficient describing a quantum state in a selected basis. Its squared magnitude determines a measurement probability in that basis.

An amplitude is not itself an ordinary probability. See Section [19.1](#).

Authentication.

Evidence about a message's source, a party or a key relationship, as specified by the scheme or protocol. It is different from authorization, which determines whether an action is permitted. See Section [2.3](#).

Bell state.

One of a family of two-qubit entangled states. The plus Bell state in this chapter gives equal 00/11 outcomes in the standard basis, but those counts alone do not prove experimental entanglement. See Section [19.2](#).

Bias.

A systematic difference between an estimator's expected value and its intended target. Increasing the sample size does not by itself remove an incorrect model or calibration bias. See Section [19.3](#).

Bit and byte.

A bit has value 0 or 1. A byte, in this book, is a sequence of eight bits. Its numeric or text meaning depends on how those bits are interpreted. See Section [3.2](#).

Blinding.

Randomizing an internal operand in a way that can be removed from the result. RSA blinding reduces the direct relation between chosen external inputs and private-operation timing; it does not certify every possible side channel. See Section [17.3](#).

Block cipher.

A keyed, reversible transformation of blocks with a fixed length. For one fixed key, different input blocks have different output blocks. Protecting a whole message requires a suitable surrounding construction. See Section [6.1](#).

Bombe.

Electromechanical equipment used to eliminate inconsistent Enigma setting candidates from crib-based relationships. A candidate stop requires further checking. See Section [1.6](#).

Canonical encoding.

A required agreed representation of structured data. It is distinct from separating fields and from deciding whether an authenticated operation

is authorized. See Section [17.4](#).

Certificate and CA.

A certificate is a signed statement connecting a subject and public key under specified rules. A certification authority (CA) is an issuer authorized within the relevant trust arrangement to make such statements. See Section [13.5](#).

ChaCha20.

A secret-key method that produces a keystream from a key and specified position inputs. XOR combines that stream with the message. The AEAD construction ChaCha20-Poly1305 adds a defined authentication calculation. See Sections [6.2](#) and [7.5](#).

Chain key.

Secret ratchet state used to derive one message-key output and the next chain key. A copied current chain key predicts its unchanged future chain. See Section [16.3](#).

Cipher suite.

In TLS 1.3, a selection of the record AEAD algorithm and handshake/key-schedule hash. Certificate and agreement choices are negotiated separately. See Section [15.2](#).

Ciphertext.

The encrypted representation of a message at the layer being discussed. The original input is the plaintext. Plaintext can be a program, picture or record, rather than readable prose. See Section [2.2](#).

Circuit privacy.

Protection of specified information about an evaluated computation from the result recipient. Input confidentiality from the evaluator does not automatically provide it. See Section [20.5](#).

Cofactor.

The ratio of the full curve group's order to the chosen base point's order. It distinguishes the whole group from the subgroup used by a construction. See Section [12.4](#).

Collision.

A matching pair of selected values. For a hash collision, the inputs must be different while their digests are equal. Hashing an identical

input twice does not find a collision. See Sections [5.3](#) and [8.2](#).

Completeness.

Acceptance of a valid claim when the prescribed honest proof process is followed. It is different from excluding invalid claims and from hiding a witness. See Section [20.1](#).

Constant-time programming.

Avoiding secret-dependent control flow, memory access and instruction timing under the relevant leakage model. It does not mean that all public input lengths and network requests take identical time. See Section [17.3](#).

Correctness.

The requirement that the intended operations give the specified result for allowed inputs. Correct recovery of a message does not by itself establish that an attacker cannot learn it. See Section [2.2](#).

Crib.

A proposed plaintext fragment aligned with ciphertext. Machine rules can reject inconsistent hypotheses; passing one test does not prove the guess correct. See Section [1.5](#).

Crypto-agility.

The controlled ability to change cryptographic choices across an actual system, including credentials, stored formats, endpoint support and negotiation policy. Supporting several library function names is only part of the task. See Section [19.8](#).

Cryptoperiod.

The period during which a key is authorized for a specified use. Applying new protection and processing earlier protection can have different permitted periods. See Section [14.4](#).

Deterministic.

Fixing all inputs fixes the output. An algorithm using fresh randomness can still consist of deterministic steps once all those random choices have been included among its inputs. See Sections [1.4](#) and [5.5](#).

Diffie–Hellman.

A key-agreement method that lets parties calculate matching secret material from their private choices and exchanged public values.

Authentication of the exchange and protection of subsequent messages require additional specified steps. See Section [10.2](#).

Digital signature.

Verification data produced using a private signing key and a message. The corresponding public key checks it. Verification does not by itself establish identity, authorization, freshness or truth. See Sections [13.1](#) and [13.6](#).

Domain separation.

Specified labels or context that distinguish cryptographic uses. The encoding and receiver checks must preserve those distinctions. See Section [17.4](#).

Double Ratchet.

A two-party stateful messaging construction combining per-message symmetric chain steps with DH-based root updates. It keeps separate sending and receiving chains. See Section [16.3](#).

ECDSA and EdDSA.

Different elliptic-curve signature families with distinct formulas, encodings and processing rules. Ed25519 is a specific EdDSA instance, not ECDSA or X25519. See Section [13.3](#).

ElGamal.

A public-key encryption method using a fresh sender exponent and a shared group value. The multiplicative version studied here hides message elements under a decision Diffie–Hellman (DDH) assumption, but allows algebraic changes to ciphertexts. See Section [11.6](#).

Elliptic curve.

In the forms studied here, coordinate pairs satisfying a specified cubic equation, together with an identity element and a defined point-addition operation. Cryptographic examples use a finite field. See Sections [12.1](#) and [12.2](#).

End-to-end encryption.

Content protection between the intended participating endpoints rather than only a channel to an intermediary. It does not automatically hide metadata or protect displayed plaintext from its endpoint. See Section [16.1](#).

Enigma.

A family of electromechanical cipher machines using changing substitutions. The history distinguishes earlier Polish breakthroughs, Turing and Welchman's Bombe work, engineering and operational contributions. See Sections [1.5](#) and [1.6](#).

Entanglement.

A joint-state relationship that cannot be represented by independent pure states for the parts in the example considered. Local random outcomes do not provide controllable faster-than-light communication. See Section [19.2](#).

Entropy.

A measure of a stated feature of uncertainty in a probability model. Shannon entropy and min-entropy answer different questions. Neither can be inferred from how irregular a printed string looks. See Section [5.4](#).

Envelope encryption.

A design in which a data key protects a file or record, while a wrapping key protects that data key. The hierarchy creates recovery and access dependencies that must also be managed. See Section [14.3](#).

Epoch.

In sparse continuous agreement, an ordered identifier for an emitted shared-secret generation. It is not necessarily a timestamp or one application-message number. See Section [16.5](#).

Error mitigation.

Additional experiments and processing that improve estimates of specified ideal quantities under assumptions. It differs from encoding and protecting logical quantum information through error correction. See Section [19.3](#).

Error syndrome.

Information about specified error relations in an encoded quantum state, obtained without directly measuring the protected logical value. A decoder interprets the syndrome under the code's assumptions. See Section [19.3](#).

Expectation value.

The probability-weighted average of a measurement's assigned numerical outcomes. A finite sample estimates it with statistical variation and potentially bias. See Section [19.3](#).

Field.

A mathematical system with compatible addition, subtraction, multiplication and division by nonzero elements. A finite field has finitely many values. AES uses one with 256 elements and specified byte arithmetic. See Section [6.4](#).

Finished.

A TLS handshake message carrying a keyed check of the preceding handshake transcript. It is different from the certificate signature and service-name check. See Section [15.3](#).

Forward secrecy.

Protection of earlier session material against later disclosure of specified long-term secrets, under the construction's assumptions and appropriate erasure. It does not imply protection after exposure of a live traffic or ephemeral secret. See Section [15.6](#).

Functional correctness.

Agreement of an implementation with its formal specification under the stated contract. It does not establish that the specification supplies every desired protection. See Section [18.3](#).

Fuzzing.

Exercising software with generated or mutated inputs while monitoring specified failures or properties. Its findings depend on reached paths and observed outcomes. See Section [18.2](#).

Group.

A set with a specified operation obeying closure and associativity, with an identity and inverses. Its order counts its elements. A generator of a cyclic group reaches every element by repeated application of the operation. See Section [4.4](#).

Hadamard gate.

A one-qubit operation that adds and subtracts input amplitudes with normalization by the square root of two. Its interference can turn a

relative-sign difference into a different measurement outcome. See Section [19.2](#).

Handshake.

The messages and processing that establish a connection's parameters, key material and specified authentication. Not every intermediate stage has the final stage's guarantees. See Section [15.2](#).

Hardware security module.

An HSM performs cryptographic operations within a specified protected device boundary. Preventing raw-key export does not by itself prevent an authorized interface from being misused. See Section [14.3](#).

Harvest now, decrypt later.

Retaining protected data or an exchange transcript to exploit a possible future attack. Changing today's keys does not erase an attacker's earlier copies. See Section [19.9](#).

Hash and digest.

A hash function calculates an output, called a digest, from an input using a public rule. The fixed-output hashes studied here give a fixed digest length for many permitted input lengths. The digest does not identify a sender on its own. See Section [8.1](#).

HKDF.

HMAC-based Extract-and-Expand Key Derivation Function. A specified method for preparing and expanding suitable existing keying material. It is not a password-stretching function. See Section [9.5](#).

HMAC.

Hash-based message authentication code. A particular two-layer calculation using a hash function and a secret key to produce a tag. It is different from simply hashing a message publicly. See Section [8.5](#).

Homomorphic encryption.

Encryption permitting specified operations on ciphertexts, with a corresponding defined plaintext result after authorized decryption. Evaluation does not independently establish authenticity or correct execution. See Section [20.5](#).

Honest-verifier zero knowledge.

A privacy property whose simulation concerns a verifier following the specified process. It does not by itself establish the same guarantee for every malicious or concurrent verifier. See Section [20.2](#).

Implicit rejection.

Producing replacement secret material for a specified consistency failure during decapsulation, instead of exposing that failure through an ordinary flag. Input-validation and application-authentication requirements still apply. See Section [19.7](#).

Inverse.

An operation or value that undoes another under the stated rules. A multiplicative inverse modulo

gives a product with remainder 1. Not every value has such an inverse. See Section [4.2](#).

IV.

Initialization vector. An input that starts a mode's processing. Requirements depend on the mode: uniqueness and unpredictability are different properties. The fixed initial value in a hash calculation has a different role. See Sections [7.2](#) and [8.3](#).

Key derivation function.

A KDF turns suitable input material into keying material under a specified construction. Some KDFs process passwords with deliberate resource costs; HKDF serves a different purpose. See Sections [9.3](#) and [9.5](#).

Key inventory.

A record of key identities, purposes, owners, states and dependencies, normally referencing protected key storage rather than collecting raw secrets. A key identifier names a key; it does not itself authorize use. See Section [14.1](#).

Key management.

The arrangements for creating, establishing, protecting, using, replacing, recovering and destroying key material and its security-relevant associations. See Chapter [14](#).

Key-encapsulation mechanism.

A KEM generates a shared secret and public encapsulation using a recipient's public key. The recipient obtains the matching secret using the encapsulation and private key. The basic interface does not encrypt an application-chosen message. See Section [10.4](#).

Key.

Data that controls a cryptographic operation. Shared secret keys and public/private key pairs support different arrangements. Key length alone does not establish a system's security. See Section [2.3](#).

Keystream.

Bytes generated for combination with message bytes, usually by XOR. An observer should not be able to predict the protected portion under the construction's assumptions. Reusing the same portion for different messages reveals a relation between them. See Sections [6.2](#) and [7.3](#).

KeyUpdate.

A TLS 1.3 message advancing a sending direction to the next application traffic-secret generation. It introduces no new independent agreement secret. See Section [15.5](#).

Known-answer test.

A test comparing a calculated result with exact expected data from an identified independent basis, such as a published vector. See Section [18.1](#).

Lattice.

Points formed by integer combinations of independent basis vectors. A two-dimensional teaching picture does not measure the difficulty of a high-dimensional cryptographic problem. See Section [19.6](#).

Learning with errors.

LWE, an inference problem involving a secret vector and deliberately noisy modular linear relations under specified distributions. The dimensions, error distribution and other parameters are essential to its analysis. See Section [19.6](#).

Logical qubit.

An encoded unit of quantum information protected through an error-correction scheme. A physical-qubit count alone does not establish a

cryptographic attack capability. See Section [19.3](#).

MAC and tag.

A message authentication code uses shared secret material to produce verification data called a tag. Anyone given the same key can generally produce tags too, which limits what they establish about an individual sender. See Section [8.5](#).

Memory safety.

Preservation of the specified rules for memory access, object lifetime and related permissions. It is distinct from algorithmic correctness or a complete privacy guarantee. See Section [18.3](#).

Message key.

Per-message secret material produced by a ratchet's chain interface. A concrete scheme can further expand or combine it before encryption. See Section [16.3](#).

Modulus and remainder.

The modulus is the positive number used in a modular calculation. A remainder is what remains after removing complete multiples of it, chosen here from zero up to one less than the modulus. See Section [3.5](#).

Multiparty computation.

A protocol for an agreed computation on private inputs under a defined participant and adversary model. Its allowed output can still reveal information. See Section [20.4](#).

Multiplexing.

Selecting between implementations of the same specified algorithm, for example scalar and vectorized paths. See Section [18.4](#).

Multiplication triple.

Fresh correlated shared values whose third value is the product of the first two. Their use enables specified shared multiplications; correctness, authentication and single use are separate requirements. See Section [20.4](#).

Negative test.

A selected check that an operation refuses a case that the relevant rule requires it to refuse, with the required observable effects. See Section [18.2](#).

No-cloning theorem.

The impossibility of a universal perfect operation that independently copies every unknown quantum state while preserving the original.

Encoding information for error correction does not require such copies.

See Section [19.3](#).

Nonce.

An additional input that distinguishes uses of a construction. For the AEAD examples here, encryption must not repeat a nonce with the same key. Its purpose is different from keeping a key secret or recording whether a message has already been accepted. See Section [7.2](#).

OAEP.

Optimal Asymmetric Encryption Padding. A randomized encoding used with RSA encryption. Its masks and decoding checks belong to the scheme; appending arbitrary random bytes is not equivalent. See Section [11.5](#).

Oracle.

An interface that supplies an answer useful in cryptanalysis, even if it never returns a secret directly. An adaptive attacker can select a later query from earlier answers. See Section [17.2](#).

Padding oracle.

An observable distinction about whether decrypted data meet a padding or formatting condition. Errors, connection behavior or timing can provide the distinction. See Section [17.2](#).

Parameter.

A setting that specifies part of a calculation, such as memory use, number of passes or requested output length. The setting's meaning and allowed range depend on the construction. See Section [9.4](#).

Per-signature secret.

A secret scalar required by a signing construction, such as Schnorr or ECDSA. Its disclosure or reuse across different signing equations can reveal a long-term signing key. Its requirements differ from those of a public AEAD nonce. See Section [13.4](#).

Post-compromise security.

Conditional protection for later communication after an earlier exposure. It requires a defined attacker model and suitable fresh inputs; it does not promise secrecy while an attacker keeps reading every new secret. See Section [16.4](#).

Post-quantum cryptography.

Classical cryptographic algorithms designed to resist both classical and quantum attacks under their stated assumptions. Ordinary users do not need quantum hardware to run them. See Section [19.1](#).

PQXDH.

Post-Quantum Extended Diffie–Hellman, an asynchronous setup combining curve agreement with a post-quantum KEM contribution. The cited revision’s authentication remains conventional. See Section [16.5](#).

Pre-shared key.

A PSK is secret material already shared before the current exchange. A resumption ticket identifies or carries server-side context; its visible bytes are not the PSK itself. See Section [15.6](#).

Precondition.

A condition required before an operation for its stated guarantee to apply. A postcondition describes the promised result or state afterward. See Section [18.3](#).

Prekey.

A setup public key published before a sender begins an exchange. Its matching private value is protected at the recipient; removing a public listing is different from deleting that private value. See Sections [16.1](#) and [16.2](#).

Prime.

An integer greater than 1 whose only positive divisors are 1 and itself. A product of two distinct primes supplies the modulus in the RSA construction discussed here. See Section [4.1](#).

Protocol.

Rules for what parties send, check and do next. Algorithms are components of a protocol; a correct component does not establish every property of the surrounding exchange. See Section [2.1](#).

Public and private keys.

Related values that support different operations. A public encryption key can be distributed, while the matching private decryption key must be protected. Signature keys have their own roles and interfaces. See Section [11.1](#).

Qubit.

A unit of quantum information whose state is described relative to basis states and amplitudes. Measuring in the chosen basis produces an outcome, not a readable list of every amplitude. See Section [19.1](#).

Record.

A TLS framing unit whose protected form contains encrypted inner content and a tag, with its specified outer header authenticated as associated data. Records, handshake messages and network packets have different boundaries. See Section [15.5](#).

Regression test.

A retained test with an expected outcome, often recording a previously discovered defect so that later changes do not reintroduce it unnoticed. See Section [18.1](#).

Relative phase.

A relationship between components of a quantum state that can affect later interference. A common global phase does not change the state's measurable predictions. See Section [19.2](#).

Replay.

Delivery of previously valid data again. An unchanged packet can pass a cryptographic check while representing an instruction that should not be processed twice. See Section [2.4](#).

Resumption.

Establishing a later TLS connection using a PSK associated with an earlier connection, with or without a fresh asymmetric agreement contribution. It does not automatically permit early application data. See Section [15.6](#).

Root key.

Evolving ratchet secret used when mixing a new agreement output to derive a new root and message chain. It is not the key directly used for

every message. See Section [16.3](#).

RSA.

A public-key method named after Rivest, Shamir and Adleman. Its basic transformation uses powers and remainders under a modulus formed from prime factors. A complete encryption scheme needs more than that raw transformation. See Sections [11.1](#) and [11.2](#).

Salt.

Additional input to a specified derivation or password calculation, usually public. Fresh per-record password salts prevent one common password-to-verifier table from serving all records; HKDF salts have a different role and reuse rules. See Sections [9.2](#) and [9.5](#).

Scalar multiplication.

Applying an integer number of curve-group additions to a point. The coordinates are not multiplied separately by that integer. See Section [12.4](#).

Secret sharing.

Distributing a secret so that qualified share sets reconstruct it while smaller sets receive the scheme's privacy guarantee. Ordinary shares do not automatically authenticate themselves. See Section [20.3](#).

Security assumption.

A stated condition, often a computational task believed difficult for an attacker under a specified model. A security proof can connect a construction's guarantee to that condition; it does not turn the condition into an established fact. See Section [5.6](#).

Shot.

One prepared quantum execution followed by its specified measurement. Repeated shots supply the records used to estimate probabilities or expectation values. See Section [19.3](#).

Side channel.

Information revealed through behavior beyond the intended outputs, such as timing or memory access. The attack and defense depend on which observations are available. See Section [17.3](#).

Skipped message key.

A retained per-message key allowing a delayed message to be processed after the receiving chain advances. Its retention preserves a corresponding exposure path. See Section [16.3](#).

Soundness.

A bound on obtaining acceptance for an invalid claim under a proof system's stated model. Knowledge extraction and witness privacy are distinct properties. See Section [20.1](#).

Sparse Continuous Key Agreement.

SCKA emits an ordered sequence of shared secrets as public protocol data is exchanged. A message can be processed without emitting a new agreement secret. See Section [16.5](#).

Specification.

A precise statement of intended behavior or properties. Its correctness as a description of the desired system must itself be examined. See Section [18.3](#).

Statevector.

An ordered amplitude representation in a chosen basis. A general full statevector for

qubits has

entries, although structured computations can admit more efficient classical descriptions. See Section [19.2](#).

Succinct argument.

Compact verification evidence relative to the defined task and representation. Small evidence does not make its generation or public-input processing free. See Section [20.2](#).

Threshold signature.

A signature jointly computed by a qualified group using distributed private signing material. Key generation, nonce state, authorization and availability still need their own treatment. See Section [20.3](#).

TLS.

Transport Layer Security. A protocol combining handshake establishment and protected records, with guarantees that depend on the

selected mode, credentials, checks and endpoints. See Chapter [15](#).

Transcript.

For a TLS calculation, the specified ordered handshake-message encodings covered at that stage, with special retry processing when required. Record headers are not part of the handshake transcript. See Section [15.3](#).

Transpilation.

Translating and optimizing a circuit for a specified gate set and device connectivity. Ideal equivalence does not make every resulting implementation equally costly or equally exposed to noise. See Section [19.3](#).

Triple Ratchet.

A specified composition combining classical and sparse post-quantum message-key contributions with a hybrid KDF. It does not mean three participants or three identical encryptions. See Section [16.5](#).

Trust anchor.

Trusted authority information supplied as an input to certificate-path validation. Receiving an unknown self-signed certificate does not automatically establish trust in its key. See Section [13.5](#).

Trusted computing base.

Components and assumptions relied on by an argument without being established within that argument. The exact boundary depends on the verification and execution chain. See Section [18.4](#).

Uniform.

Each allowed value has the same probability. A large range of possible values does not by itself establish that a generator selects uniformly from it. See Section [5.2](#).

Witness.

Private information satisfying a specified checking relation with public statement data. A proof about that relation does not automatically establish the truth of an underlying real-world record. See Section [20.1](#).

Wrapping key.

A key used through a specified construction to protect other key material. Rewrapping under a new wrapping key can leave the protected

data key unchanged. See Sections [14.3](#) and [14.4](#).

X25519.

A specified key-agreement function using Curve25519, scalar input bytes and encoded coordinate bytes. Its decoding and acceptance rules differ from a full-point SEC 1 interface. See Sections [12.5](#) and [12.6](#).

XOR.

Exclusive OR. At each bit position it gives 1 when the two input bits differ and 0 when they match. Applying the same mask twice restores the input, but that identity alone does not provide secrecy. See Section [3.3](#).

Zero knowledge.

A defined privacy property connecting a verifier's view to an appropriate simulation without the witness. The public statement and its permitted implications remain visible. See Section [20.1](#).

Notes and Bibliography

The numbered entries form an annotated bibliography arranged by chapter. Each entry identifies its source and the passages or claims for which it is used. A chapter can also refer to an earlier entry without repeating the full citation. Follow a source's link to the identified publication or project; a note about selected source reading does not claim that the whole implementation was audited.

Chapter 1: From Secret Writing to Modern Cryptography

The historical discussion uses original publications and identifies later institutional accounts separately. The substitution and four-bit examples are original teaching material; they are not historical quotations or practical encryption recommendations.

1. Auguste Kerckhoffs, *La Cryptographie militaire*, *Journal des sciences militaires*, volume IX, January 1883, pp. 5–38, and February 1883, pp. 161–191. Part I, Section II, gives six operational requirements; the second concerns not depending on secrecy of the system. The later classical-cipher discussion examines substitution, frequencies and the limitations of short samples. The collected text is Fabien Petitcolas's digitization; the publication dates follow the original bibliographic record. <https://www.petitcolas.net/kerckhoffs/>.

Back to text: [1](#) [2](#) [3](#)

2. Claude E. Shannon, *Communication Theory of Secrecy Systems*, *Bell System Technical Journal* 28(4), October 1949, pp. 656–715. The opening footnote identifies a 1945 classified predecessor. Section 10 defines perfect secrecy and develops key-space and uncertainty restrictions; printed p. 682 gives the entropy expressions for messages and keys used as background in Chapter 5. Section 11 discusses remaining uncertainty. The discussion on pp. 708–709 introduces diffusion and confusion. The probability notation in Chapter 1 is a

modern restatement, with variables explained in the text.
[doi:10.1002/j.1538-7305.1949.tb00928.x](https://doi.org/10.1002/j.1538-7305.1949.tb00928.x).

Back to text: [1](#) [2](#) [3](#) [4](#)

3. Gilbert S. Vernam, *Secret Signaling System*, US patent 1,310,719, issued 22 July 1919. The specification describes a telegraph mechanism for combining message and key signals. The patent date is used as a document milestone; the one-time pad secrecy argument is supported separately by its conditions and Shannon's analysis.
<https://patents.google.com/patent/US1310719A/en>.

Back to text: [1](#)

4. Whitfield Diffie and Martin E. Hellman, *New Directions in Cryptography*, *IEEE Transactions on Information Theory* 22(6), November 1976, pp. 644–654. The introduction and Section III distinguish public-key encryption and public exchange of keying material, and discuss the key-distribution problem. The introduction acknowledges Merkle's related work. Historical proposals and examples in the paper are not substituted for current authenticated-protocol requirements. [doi:10.1109/TIT.1976.1055638](https://doi.org/10.1109/TIT.1976.1055638).

Back to text: [1](#) [2](#)

5. Ronald L. Rivest, Adi Shamir and Leonard Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, *Communications of the ACM* 21(2), February 1978, pp. 120–126. The collected 15-page author-hosted retypeset copy has different pagination. The abstract and opening sections support the historical discussion of a concrete public-key construction, not a claim that textbook RSA is a modern secure encryption or signature interface.
[doi:10.1145/359340.359342](https://doi.org/10.1145/359340.359342).

Back to text: [1](#) [2](#) [3](#)

6. GCHQ, *James Ellis*, institutional historical account, consulted 20 October 2026. It identifies related classified work by Ellis, Cocks and Williamson and its later public announcement. This is a later account, not one of the public papers dated in the timeline.
<https://www.gchq.gov.uk/person/james-ellis>.

Back to text: [1](#)

Chapter 2: What Cryptography Protects

These notes identify the technical basis of the chapter. The laboratory order, questions and instructional diagram are original teaching material. Source numbering is local to this opening manuscript and can be extended as the book grows.

7. Dan Boneh and Victor Shoup, *A Graduate Course in Applied Cryptography*, version 0.6, January 2023. The preface distinguishes constructions, security goals and assumptions. Chapter 9, especially Sections 9.1–9.2, explains authenticated encryption and ciphertext integrity. For Chapter 4, Appendix A.2–A.3 supports modular inverses and multiplicative-group notation, while Sections 16.1.1–16.1.2 and 16.1.5 explain generic attacks, composite orders and relationships between DL, CDH and DDH. Chapter 5 also uses Appendix B.1 on birthday probabilities and Chapter 3’s game-based argument for pseudorandom generators and stream encryption. Historical parameter examples in that reference are not adopted as present-day recommendations. The book is a technical reference for the explanations here; its prose and figures are not reproduced.

<https://toc.cryptobook.us/>.

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [11](#) [12](#) [13](#) [14](#)

8. Elaine Barker, *Recommendation for Key Management: Part 1—General*, NIST SP 800-57 Part 1, Revision 5, May 2020. Executive summary and Sections 3.1–3.7 cover confidentiality, integrity, authentication, authorization, the scope of non-repudiation and combinations of security services. Chapter 2 uses those conceptual distinctions, rather than presenting this edition as a complete current deployment policy. [doi:10.6028/NIST.SP.800-57pt1r5](https://doi.org/10.6028/NIST.SP.800-57pt1r5).

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [11](#)

9. Eric Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 9846, July 2026. Section 1 describes channel authentication, confidentiality and integrity, and distinguishes those goals from limitations involving transmitted length and traffic analysis. The later TLS chapter will examine the protocol and its assumptions in detail. <https://www.rfc-editor.org/info/rfc9846>.

Back to text: [1](#)

10. Peter Saint-Andre and Rich Salz, *Service Identity in TLS*, RFC 9525, November 2023. Sections 1.2–1.4 distinguish the scope of service-identity checking from certificate issuance and certification-path validation. This chapter introduces that distinction without attempting to specify a complete verifier. <https://www.rfc-editor.org/info/rfc9525>.

Back to text: [1](#) [2](#) [3](#)

Chapter 3: Bits, Bytes and Remainders

The representations and algorithm operations are checked against their specifications. The character examples, byte diagram, comparison table, worked arithmetic and small program are original teaching material. The program illustrates operations with fixed public inputs; it is not a recommended encryption construction.

11. François Yergeau, *UTF-8, a Transformation Format of ISO 10646*, RFC 3629, November 2003. The introduction and Section 3 explain the ASCII-compatible range, code-point representation and one-to-four-octet encoding. A visible character need not correspond to one code point or one encoded byte. <https://www.rfc-editor.org/info/rfc3629>.

Back to text: [1](#)

12. Unicode Consortium, *Unicode Normalization Forms*, Unicode Standard Annex #15, consulted 20 October 2026. Sections 1.1–1.2 distinguish canonical equivalence and normalization forms; Section 10 discusses identity tests on equivalent text. The chapter’s precomposed and combining-accent examples were also checked by encoding both sequences. Normalization is a protocol requirement when specified, not permission to alter authenticated data arbitrarily. <https://www.unicode.org/reports/tr15/>.

Back to text: [1](#)

13. National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS 197, updated 9 May 2023. Sections 2–3 define the notation used in Chapter 3. Chapter 6 draws on Section 3.4 for state placement, Sections 4.1–4.4 for byte-field arithmetic, Sections 5.1–5.3 for rounds, key expansion and inversion, and Appendix B for the AES-128 worked example. Its ⁵⁷



¹³ calculation expands the reductions in Section 4.2. The state-placement diagram is original; the published numerical example is identified as test data. The 2023 update makes editorial improvements without changing the algorithm. [doi:10.6028/NIST.FIPS.197-upd1](https://doi.org/10.6028/NIST.FIPS.197-upd1).

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#)

14. Simon Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 4648, October 2006. Section 8 explains base-sixteen encoding, four-bit groups and two output characters per input octet. Encoding arbitrary bytes for transport does not itself provide secrecy. <https://www.rfc-editor.org/info/rfc4648>.

Back to text: [1](#)

15. Yoav Nir and Adam Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, June 2018, an Informational IRTF publication. Sections 1.1 and 2.1 define the word operations; Sections 2.1.1 and 2.3.2 provide the quarter-round and block vectors used in Chapter 6. Sections 2.2–2.4 specify word groups, feed-forward, serialization and stream processing. Addition modulo

2^{32}

retains internal carries and is distinct from XOR, despite the source’s misleading use of “carryless” in Section 2.3. Verified errata 5989 and 7880 clarify concatenation and the final partial-block length in Section 2.4.1; the chapter follows the required exact-length result. Its diagrams and explanatory trace are original presentations. <https://www.rfc-editor.org/info/rfc8439>; <https://www.rfc-editor.org/errata/eid5989>; <https://www.rfc-editor.org/errata/eid7880>.

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#)

16. Python Software Foundation, *Built-in Types*, Python 3.12 documentation. The entries for numeric types, integer bitwise operations, `int.from_bytes`, text and byte sequences define the program’s representations and conversions. The exact printed listing and its expected output were executed separately from the typesetting check. A successful teaching program is not a security assessment of a cryptographic implementation. <https://docs.python.org/3.12/library/stdtypes.html>.

Back to text: [1](#) [2](#)

Chapter 4: Primes, Groups and Hard Problems

The prime, inverse, power and group calculations are original teaching examples. Their small values make the operations inspectable and provide no useful cryptographic secrecy. Source 7 also supports the chapter's mathematical background. The additional notes below identify the group treatment, original research and selected software documentation used here.

17. Mike Rosulek, *The Joy of Cryptography*, online chapter *Key Exchange*, consulted 20 October 2026. Sections 13.1–13.1.2 define cyclic groups, element and group orders, generators and efficient exponentiation; Section 13.2.1 separates a DDH-based passive-security formulation from authentication. The running examples and diagram in Chapter 4 are original. <https://joyofcryptography.com/ke/>.

Back to text: [1](#) [2](#) [3](#) [4](#)

18. Neal Koblitz, *Elliptic Curve Cryptosystems, Mathematics of Computation* 48(177), January 1987, pp. 203–209. Section 2, especially printed p. 204, describes the group of curve points, its possible structure and computing point multiples by doubling and addition. The chapter uses these passages only to connect multiplicative and additive notation; the curve formulas are developed later. [doi:10.1090/S0025-5718-1987-0866109-5](https://doi.org/10.1090/S0025-5718-1987-0866109-5).

Back to text: [1](#) [2](#) [3](#) [4](#)

19. Dan Boneh, *The Decision Diffie–Hellman Problem*, in *Algorithmic Number Theory*, ANTS III, Lecture Notes in Computer Science 1423, Springer, 1998, pp. 48–63. Sections 1–2 distinguish CDH and DDH and define the sampling experiments. The author-hosted 14-page copy has different pagination from the published proceedings. Its historical examples are not current parameter advice, and Chapter 4 does not claim that DLP hardness alone implies DDH hardness. <https://crypto.stanford.edu/~dabo/abstracts/DDH.html>.

Back to text: [1](#) [2](#) [3](#)

20. OpenSSL Project, *BN_mod_exp_mont* documentation and selected passages of `crypto/bn/bn_exp.c`, collected source tagged `openssl-4.0.3`. The documented fixed-window variant and its precomputation layout illustrate an implementation concern about secret-dependent behavior. This selected source reading does not constitute an audit of

the routine or the library. The same design description was checked against the versioned online manual:

https://docs.openssl.org/3.4/man3/BN_mod_exp_mont/. Project source: <https://github.com/openssl/openssl>.

Back to text: [1](#)

21. Python Software Foundation, Python 3.12 documentation, *Built-in Functions*, entry `pow`, and *Mathematical Functions*, entry `math.gcd`. These specify three-argument modular powers, the coprimality requirement for negative exponents and gcd behavior. The exact printed Chapter 4 listing was executed with Python 3.12; it checks public values and is not a secret-exponent implementation recommendation.

<https://docs.python.org/3.12/library/functions.html#pow>;
<https://docs.python.org/3.12/library/math.html#math.gcd>.

Back to text: [1](#) [2](#)

22. Peter W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, *SIAM Journal on Computing* 26(5), 1997, pp. 1484–1509. The abstract of the collected author preprint, version 2 dated 25 January 1996, states efficient algorithms in a quantum-computing model and distinguishes the preliminary 1994 conference version. This supports the computational-model qualification; the chapter makes no claim about a present-day machine’s ability to break a deployed key.

[doi:10.1137/S0097539795293172](https://doi.org/10.1137/S0097539795293172); author preprint:
<https://arxiv.org/abs/quant-ph/9508027>.

Back to text: [1](#) [2](#) [3](#)

Chapter 5: Probability, Randomness and Security

The coin, guessing, collision and entropy examples are original teaching material. The collision graph uses the exact finite product under independent uniform sampling, not empirical measurements of a cryptographic implementation. Sources 7 and 2 provide additional mathematical and historical background. The source documents below address different layers of random generation and computational security.

23. Elaine Barker and John Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, NIST SP 800-90A Revision 1, June 2015. Sections 7–8 distinguish state, instantiation, generation, reseeding and resistance properties. Chapter 5 introduces their roles without specifying a new generator or adopting a historical parameter table as current deployment advice. [doi:10.6028/NIST.SP.800-90Ar1](https://doi.org/10.6028/NIST.SP.800-90Ar1).

Back to text: [1](#) [2](#) [3](#)

24. Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish and Mike Boyle, *Recommendation for the Entropy Sources Used for Random Bit Generation*, NIST SP 800-90B, January 2018. Sections 2.1–2.2 define min-entropy and the source model; Section 4 distinguishes health tests and their limits. The publication’s 29 May 2025 potential-updates sheet was also checked; Chapter 5 does not use its affected numerical estimation examples. [doi:10.6028/NIST.SP.800-90B](https://doi.org/10.6028/NIST.SP.800-90B).

Back to text: [1](#) [2](#) [3](#)

25. Elaine Barker, John Kelsey, Kerry McKay, Allen Roginsky and Meltem Sönmez Turan, *Recommendation for Random Bit Generator (RBG) Constructions*, NIST SP 800-90C, September 2025, final publication. Sections 1–2 explain the relationship among the SP 800-90 documents, ideal versus generated output, and different construction capabilities. The conceptual diagram in Chapter 5 is original and is not a normative diagram of the four construction classes. [doi:10.6028/NIST.SP.800-90C](https://doi.org/10.6028/NIST.SP.800-90C).

Back to text: [1](#) [2](#)

26. Shafi Goldwasser and Silvio Micali, *Probabilistic Encryption*, *Journal of Computer and System Sciences* 28(2), April 1984, pp. 270–

299. The abstract and Sections 1.1–1.4 distinguish reversing a function from protecting partial information and connect a probabilistic security definition to a computational assumption. These opening pages were read directly in the collected scanned paper. The book’s message-choice experiment is a modern teaching presentation, not a claim that every present-day attack model is identical to the original formulation. [doi:10.1016/0022-0000\(84\)90070-9](https://doi.org/10.1016/0022-0000(84)90070-9).

Back to text: [1](#)

27. OpenSSL Project, *RAND_bytes*, versioned 4.0 manual and the collected `openssl-4.0.3` source documentation `doc/man3/RAND_bytes.pod`. The description, seeding notes and return-value contract support the discussion of a cryptographic generation interface and application error handling. This is selected documentation reading, not a complete review or validation of the generator implementation. https://docs.openssl.org/4.0/man3/RAND_bytes/.

Back to text: [1](#)

28. Python Software Foundation, Python 3.12 documentation, *fractions*—*Rational Numbers*, *itertools*—*Functions Creating Iterators for Efficient Looping*, entry `product`, and *Mathematical Functions*, entry `log2`. These operations support exact enumeration and fraction arithmetic followed by a rounded logarithmic calculation. The exact printed program was executed independently of typesetting; it neither generates a secret nor estimates a real source’s entropy. <https://docs.python.org/3.12/library/fractions.html>; <https://docs.python.org/3.12/library/itertools.html#itertools.product>; <https://docs.python.org/3.12/library/math.html#math.log2>.

Back to text: [1](#)

Chapter 6: AES and ChaCha20

Sources 13 and 15 specify the algorithms and the published numerical examples; Source 7 supports the distinction between a block cipher and its security model. Source 16 documents the integer operations used in the optional program. The additional sources below support the field explanation and the selected connection to implementation code. The calculations, code-output checks and source reading are distinct from a full library audit or a security proof.

29. Mike Rosulek, *The Joy of Cryptography*, online appendix *A Crash Course in Binary Finite Fields*, consulted 20 October 2026. The definitions and examples explain bitstrings as polynomials, addition by XOR, reduction by an irreducible polynomial, and multiplicative inverses. Different encodings of isomorphic fields need not give the same byte-level products; Chapter 6 uses exactly the AES modulus and representation from FIPS 197.
<https://joyofcryptography.com/fields/>.

Back to text: [1](#)

30. OpenSSL Project, collected source tagged `openssl-4.0.3`. Selected passages of `crypto/chacha/chacha_enc.c` show the quarter round, the twenty-round loop, word addition and little-endian output. Selected passages of `crypto/aes/aes_core.c` show field operations, round structure and distinct implementation branches. Chapter 7 also uses the AES-GCM vectors in `test/recipes/30-test EVP data/evpciph_aes_common.txt`, which identify the GCM proposal as their source. Architecture-specific implementations and dispatch exist elsewhere in the project; this reading does not identify every deployed path, establish side-channel resistance or constitute an execution of the upstream test suite. No internal low-level interface is recommended as an application encryption API.
<https://github.com/openssl/openssl>.

Back to text: [1](#) [2](#) [3](#)

Chapter 7: Nonces and Authenticated Encryption

Source 15 also supports this chapter through RFC 8439 Sections 2.5–2.8 and its complete AEAD vector. The accumulator discussion follows the specified block extent, not the uncorrected inclusive slice in its original pseudocode; verified erratum 5689 gives that correction. Sources 31–37 below distinguish the finalized algorithms, research models, interfaces and ongoing revision work. Numerical checks and library rejection examples establish the behavior of the examples, not a complete security certification.

31. Morris Dworkin, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, NIST SP 800-38D, November 2007. Sections 5.2, 6.3–6.5 and 7 specify inputs, lengths, field multiplication, GHASH, counters and authenticated encryption/decryption. Section 8 distinguishes IV constructions and their invocation constraints; Appendix A explains the consequences of repetition. The 96-bit-IV and full-128-bit-tag examples here are a selected profile, not every permitted historical configuration. Revision status is recorded separately in Source 37. [doi:10.6028/NIST.SP.800-38D](https://doi.org/10.6028/NIST.SP.800-38D).

Back to text: [1](#) [2](#) [3](#) [4](#)

32. Phillip Rogaway, *Authenticated-Encryption with Associated-Data*, ACM Conference on Computer and Communications Security, 2002. The acquired author full version is dated 20 September 2002 and has 30 PDF pages. Its introduction and Section 3 explain the binding of clear headers to ciphertext, nonce-based syntax and separate privacy/authenticity goals. The book’s interface diagram, running header and formatting examples are original. [doi:10.1145/586110.586125](https://doi.org/10.1145/586110.586125).

Back to text: [1](#)

33. Mihir Bellare and Chanathip Namprempre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Journal of Cryptology* 21, 2008, pp. 469–491. The introduction, Section 1.2 and Section 4.3 distinguish composition rules, separate keys and weak versus strong MAC unforgeability. Theorem 4.4’s ciphertext-integrity conclusion assumes the stronger MAC property. This is the expanded journal treatment, not merely the

earlier conference version, and it does not justify every ad hoc encryption/MAC combination. [doi:10.1007/s00145-008-9026-x](https://doi.org/10.1007/s00145-008-9026-x).

Back to text: [1](#)

34. David McGrew, *An Interface and Algorithms for Authenticated Encryption*, RFC 5116, January 2008, Proposed Standard. Sections 2–3 specify the AEAD interface, exact associated-data inputs, failure result, nonce generation and coordination between shared-key senders. Its interface can package the tag inside the ciphertext; the book displays body and tag separately for explanation. The two selected constructions here require nonempty nonces of the stated length; general interface provisions for other types of algorithm are not permission to omit those nonces. <https://www.rfc-editor.org/info/rfc5116>.

Back to text: [1](#) [2](#)

35. Morris Dworkin, *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, NIST SP 800-38A, December 2001. Sections 5.3, 6.1–6.2 and Appendix C distinguish ECB’s independent blocks, the CBC IV requirement and differences among initialization rules. The CBC comparison supports the distinction between uniqueness and unpredictability; it is not a proposal to use unauthenticated CBC. Historical algorithm lists are not adopted as current deployment recommendations. [doi:10.6028/NIST.SP.800-38A](https://doi.org/10.6028/NIST.SP.800-38A).

Back to text: [1](#) [2](#)

36. Python Cryptographic Authority, *Authenticated Encryption* documentation for `cryptography` 50.0.2, entries `AESGCM`, `ChaCha20Poly1305`, their encryption/decryption methods and `InvalidTag`. The selected library example uses a fresh key for one encryption, full-length tags and authenticated decryption returning bytes only on success. The exact printed program was executed with version 50.0.2. The low-level GCM documentation also warns against using output before final authentication. These interfaces are not equivalent to a persistent record protocol.
<https://cryptography.io/en/50.0.2/hazmat/primitives/aead/>;
<https://cryptography.io/en/50.0.2/hazmat/primitives/symmetric-encryption/>.

Back to text: [1](#)

37. National Institute of Standards and Technology, *Second Pre-Draft Call for Comments: GCM and GMAC Block Cipher Modes of Operation*, SP 800-38D Revision 1 development page, 1 June 2026, checked 20 October 2026. The page states that no draft document accompanies that consultation and records the closed comment period. It discusses removing tags shorter than 96 bits and proposed wider-mode work. These are revision activities, not a replacement final algorithm silently incorporated into this chapter. The numerical GCM explanation uses the finalized 2007 specification and a 128-bit tag. <https://csrc.nist.gov/pubs/sp/800/38/d/r1/2prd>.

Back to text: [1](#)

Chapter 8: Hashes and Message Authentication

Source 7 supplies additional background on hash and MAC security models. The changed-message comparison, byte-sum example, padding calculation, HMAC data-flow figure and field encodings are original teaching presentations. The HMAC final tag is explicitly identified as published test data; its inner digest was calculated separately. Selected source reading, known-answer checks and library examples are not a proof audit or a review of all implementation paths.

38. National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS 180-4, August 2015. Sections 3–5 and 6.2–6.7 specify bit operations, message padding, initial states and SHA-2 variants. Selected passages support the SHA-256 block trace, length encoding and distinction between SHA-512/256 and ordinary SHA-512 truncation. The official publication page records a March 2023 decision to revise; the chapter uses the finalized 2015 algorithm specification rather than assuming a replacement final standard. [doi:10.6028/NIST.FIPS.180-4](https://doi.org/10.6028/NIST.FIPS.180-4); status: <https://csrc.nist.gov/pubs/fips/180-4/upd1/final>.

Back to text: [1](#)

39. National Institute of Standards and Technology, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, FIPS 202,

August 2015. Sections 3–6 and Appendix A distinguish permutation, sponge, rate/capacity, domain suffixes, fixed-output hashes, SHAKE and security properties. Section 7’s HMAC discussion specifies the rate-based block sizes. Selected passages were read directly in the collected PDF. The official page records revision activity; the explanation follows the published standard and does not depend on the known nonnormative Appendix B Algorithm 10 typo.

[doi:10.6028/NIST.FIPS.202](https://doi.org/10.6028/NIST.FIPS.202); status:
<https://csrc.nist.gov/pubs/fips/202/final>.

Back to text: [1](#) [2](#)

40. Hugo Krawczyk, Mihir Bellare and Ran Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, February 1997, Informational. Sections 2–5 describe key preparation, the inner and outer pads, saved keyed states and tag truncation. Historical MD5/SHA-1 examples and older numerical truncation recommendations are not adopted as new-design advice. The chapter uses a full HMAC-SHA-256 tag and distinguishes construction-level requirements from protocol freshness. <https://www.rfc-editor.org/info/rfc2104>.

Back to text: [1](#)

41. Mihir Bellare, Ran Canetti and Hugo Krawczyk, *Keying Hash Functions for Message Authentication*, CRYPTO ’96, Springer, Lecture Notes in Computer Science 1109, pp. 1–15. Selected introduction, construction and security-discussion passages were read in the 19-page author-hosted full version dated June 1996. Its research assumptions are distinguished from the later analysis in Source 42. [doi:10.1007/3-540-68697-5_1](https://doi.org/10.1007/3-540-68697-5_1); author version: <https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf>.

Back to text: [1](#)

42. Mihir Bellare, *New Proofs for NMAC and HMAC: Security without Collision Resistance*, *Journal of Cryptology* 28(4), 2015, pp. 844–878; published online 10 June 2014, following a preliminary CRYPTO 2006 version. Selected introductory and NMAC-to-HMAC passages, especially the qualifications on PDF page 3 and the later HMAC discussion, distinguish the removed collision assumption from

compression-function assumptions. Transferring the analysis to HMAC involves the stated dual-PRF and restricted related-key conditions for its key-derived inputs. The chapter does not claim that an arbitrary compression-function PRF hypothesis alone proves every single-key HMAC instantiation secure. [doi:10.1007/s00145-014-9185-X](https://doi.org/10.1007/s00145-014-9185-X).

Back to text: [1](#)

43. Magnus Nyström, *Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512*, RFC 4231, December 2005, Proposed Standard. Section 4.2 supplies Test Case 1's twenty-byte key, eight-byte message and final HMAC-SHA-256 result. The printed trace and listing reproduce this identified known-answer data; additional published cases check short and oversized key handling in the accompanying calculation record. <https://www.rfc-editor.org/info/rfc4231>.

Back to text: [1](#)

44. Python Software Foundation, Python 3.12 documentation, *hashlib—Secure Hashes and Message Digests* and *hmac—Keyed-Hashing for Message Authentication*. The update, digest, hexadecimal-output and comparison entries support byte representation, incremental concatenation and `compare_digest`. The exact printed listing was executed with Python 3.12.14; the online documentation consulted identifies patch version 3.12.15. The type/length checks and comparison do not certify the complete application as constant time. <https://docs.python.org/3.12/library/hashlib.html>; <https://docs.python.org/3.12/library/hmac.html>.

Back to text: [1](#) [2](#) [3](#)

45. OpenSSL Project, collected source tagged `openssl-4.0.3`. Selected passages of `crypto/hmac/hmac.c` and `doc/man7/EVP_MAC-HMAC.pod` expose key normalization, the two pads, raw inner-digest processing and the application-facing MAC interface. Selected entries in `test/recipes/30-test_evp_data/evpmd_sha.txt` supply SHA-256 and SHA-3 known-answer checks. The check record distinguishes reading these data and executing local teaching checks from building

the collected library, running its upstream suite or auditing all provider and optimized paths. <https://github.com/openssl/openssl>.

Back to text: [1](#)

Chapter 9: Passwords and Key Derivation

Sources 12, 38, 40 and 44 supply additional background on text normalization, SHA-256, HMAC and byte interfaces. The cost arithmetic, password-record presentation and key-purpose diagram are original teaching material. The HKDF example identifies published test data; the two intermediate expansion blocks are separately calculated. Example timings are assumptions rather than measured hardware performance, and the fixed password record is public test data. Parameter selection, operational approval and security analysis are distinguished from example correctness.

46. National Institute of Standards and Technology, *Digital Identity Guidelines: Authentication and Authenticator Management*, SP 800-63B-4, final July 2025. Selected passages in Section 3.1.1.2, PDF pages 25–28, support length policy, blocklists, password managers, normalization, rate limiting, protected channels, salted costly storage, recorded parameters and a separately protected verifier secret. Selected Section 4.2–4.2.1.2 passages distinguish account recovery, saved codes and issued codes, including their storage, use and validity rules. The public final-publication record was checked 20 October 2026. Its remote-authentication scope and references to approved schemes are not represented as Argon2-specific NIST validation or universal password-entropy guarantees. Recovery is discussed as an application boundary, not a claim that this chapter implements every assurance-level requirement. [doi:10.6028/NIST.SP.800-63B-4](https://csrc.nist.gov/pubs/sp/800/63/b/4/final); status: <https://csrc.nist.gov/pubs/sp/800/63/b/4/final>.

Back to text: [1](#) [2](#) [3](#)

47. Alex Biryukov, Daniel Dinu and Dmitry Khovratovich, *Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications*, IEEE European Symposium on Security and Privacy, 2016. The collected 11-page author copy’s opening design

discussion explains offline guessing, memory costs, addressing and time–memory trade-offs. Its historical Argon2d/Argon2i discussion and measurements are not substituted for the later Argon2id specification or current performance measurements.

[doi:10.1109/EuroSP.2016.31](https://doi.org/10.1109/EuroSP.2016.31).

Back to text: [1](#)

48. Alex Biryukov, Daniel Dinu, Dmitry Khovratovich and Simon Josefsson, *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*, RFC 9106, September 2021, Informational IRTF/CFRG publication. Sections 1, 3.1–3.4 and 4 specify version 1.3, the variant distinction, input binding, memory units, lanes, block allocation, passes and the two recommended profiles. Section 5.3 supplies the Argon2id known-answer vector used in the calculation checks, including its optional secret and associated data; its 32-KiB memory is test data, not the book’s password-storage profile. Selected specified operations were read without claiming a full cryptanalytic proof audit. <https://www.rfc-editor.org/info/rfc9106>.

Back to text: [1](#) [2](#)

49. Python Cryptographic Authority, *Key Derivation Functions*, `cryptography` 50.0.2 documentation. Entries for `Argon2id`, `derive_phc_encoded`, `verify_phc_encoded`, `HKDF` and `HKDFExpand` specify units, record format, failure behavior and full versus expansion-only interfaces. The `PBKDF2` and `scrypt` entries support the brief resource-control comparison; their example parameter values are not imported as universal recommendations. Both exact printed programs were executed with version 50.0.2. The Argon2 program’s generated salt varies; a separate fixed-salt record is checked for reproducibility. <https://cryptography.io/en/50.0.2/hazmat/primitives/key-derivation-functions/>.

Back to text: [1](#) [2](#) [3](#)

50. Hugo Krawczyk, *Cryptographic Extraction and Key Derivation: The HKDF Scheme*, CRYPTO 2010, Springer, Lecture Notes in Computer Science 6223, pp. 631–648. The collected 18-page proceedings copy’s introduction and selected extraction discussion

explain source distributions, attacker side information, computational extraction, context and construction-specific assumptions. It is distinguished from the extended ePrint version. The chapter does not treat deterministic expansion as fresh information-theoretic entropy or claim that every source satisfies the paper’s extraction conditions.
[doi:10.1007/978-3-642-14623-7_34](https://doi.org/10.1007/978-3-642-14623-7_34).

Back to text: [1](#)

51. Hugo Krawczyk and Pasi Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, RFC 5869, May 2010, Informational. Sections 2–4 define HMAC argument order, extraction, expansion, the one-byte counter, output limit, salt independence and reuse, context, and the absence of password stretching. Appendix A.1 supplies the printed SHA-256 test inputs, PRK and final 42-byte OKM. The calculation record also checks Appendices A.2–A.3, including long output and the absent-salt case. Historical examples of other protocols are not described as current protocol advice.
<https://www.rfc-editor.org/info/rfc5869>.

Back to text: [1](#)

52. Libsodium Project, collected release 1.0.22-RELEASE, selected passages of
`src/libsodium/crypto_pwhash/argon2/pwhash_argon2id.c` and
`src/libsodium/crypto_kdf/hkdf/kdf_hkdf_sha256.c`. These show random salt generation, encoded record creation/checking, the byte-to-KiB memory conversion in the password interface, and HKDF’s extraction/expansion data flow. The project’s *pwhash* API* documentation supports saved parameters, rehash checks, error returns and deployment-specific resource tuning. Source reading does not imply a build, upstream-suite run or complete implementation audit.
<https://github.com/jedisct1/libsodium>;
https://doc.libsodium.org/password_hashing/default_phf.

Back to text: [1](#) [2](#)

Chapter 10: Agreeing on a Secret

Sources 4, 17, 19 and 21 support the historical exchange, group notation, DH-problem distinctions and modular arithmetic. Sources 34, 49 and 51

support the AEAD and KDF background. The small-number exchange, substituted messages, diagrams and library programs are original teaching material. The selected HPKE values are identified published test data. Numerical correctness, protocol security, identity binding, implementation behavior and approval requirements are kept distinct.

53. Mihir Bellare and Phillip Rogaway, *Entity Authentication and Key Distribution*, full author version dated August 1993; an abridged version appears in the CRYPTO '93 proceedings, published by Springer in 1994. Selected opening passages, especially Sections 1.2–1.3, distinguish entity authentication, matching conversations, fresh and exposed session keys, concurrent instances and an adversary controlling delivery. Its detailed constructions address the symmetric two-party setting. It is not cited as a proof of bare DH, HPKE or every public-key authenticated exchange.
<https://cseweb.ucsd.edu/~mihir/papers/eakd.pdf>.

Back to text: [1](#)

54. Gorjan Alagic, Elaine Barker, Lily Chen, Dustin Moody, Angela Robinson, Hamilton Silberg and Noah Waller, *Recommendations for Key-Encapsulation Mechanisms*, NIST SP 800-227, final September 2025. Selected Sections 2.2, 4.2–4.4 and 4.6 distinguish interfaces, correctness, secrecy, deployment conditions, post-processing, confirmation and multi-algorithm combiners. The public final-publication record was checked 20 October 2026. The chapter's generic parameter-free interface suppresses the selected parameter set for readability. No claim is made that every RFC suite or Python implementation automatically meets NIST approval or validation requirements. [doi:10.6028/NIST.SP.800-227](https://doi.org/10.6028/NIST.SP.800-227); status: <https://csrc.nist.gov/pubs/sp/800/227/final>.

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#)

55. Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp and Christopher A. Wood, *Hybrid Public Key Encryption*, RFC 9180, February 2022, Informational IRTF/CFRG publication. Selected Sections 4–5 specify the KEM/KDF/AEAD interfaces, DHKEM, labeled derivations, modes, key schedule and record nonce. Sections 8.1 and 9.6–9.8 distinguish application context, direction, replay,

randomness, metadata and forward-secrecy limits. Appendix A.1.1 supplies the printed base-suite values and record inputs; the check record follows the full labeled schedule and checks all six published sequence-number examples. The low-level published vector has associated data, while the printed library program uses its documented single-shot info interface. <https://www.rfc-editor.org/info/rfc9180>.

Back to text: [1](#) [2](#)

56. Mike Rosulek, *The Joy of Cryptography*, online chapter *Authenticated Key Exchange*, consulted 20 October 2026. Selected Sections 18.1–18.2 distinguish intended-peer authentication, long-term and ephemeral material, active sessions and binding the two shares, roles and identities. This is a contemporary explanatory source, separate from Bellare–Rogaway’s original symmetric-setting paper. The book’s toy values and intermediary figure are original rather than reproductions of its examples. <https://joyofcryptography.com/ake/>.

Back to text: [1](#)

57. Python Cryptographic Authority, *HPKE (Hybrid Public Key Encryption) and X25519 Key Exchange*, `cryptography` 50.0.2 documentation. The public `Suite` interface, selected suite enumerations, `encrypt/decrypt`, `info` input, combined `enc || ct` output, encapsulation length and `InvalidTag` behavior support the exact printed program. Selected collected Python and type-declaration files identify the interface’s backend bindings; no upstream build or test suite was run. The program uses the installed version 50.0.2 and base mode, with no inferred sender identity. <https://cryptography.io/en/50.0.2/hazmat/primitives/hpke/>; <https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/x25519/>.

Back to text: [1](#)

58. Adam Langley, Mike Hamburg and Sean Turner, *Elliptic Curves for Security*, RFC 7748, January 2016, Informational IRTF/CFRG publication. Selected Sections 5–7 define X25519’s byte interface, key exchange, test data and input/output caveats. Chapter 10 introduces its role without duplicating Chapter 12’s curve arithmetic. Finite-field toy-subgroup checks are explicitly not presented as the X25519

validation procedure. The check record uses the published Section 6.1 exchange as an additional byte-level known-answer test.

<https://www.rfc-editor.org/info/rfc7748>.

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#)

Chapter 11: RSA and ElGamal

Sources 7, 5, 19, 21 and 22 provide the modern mathematical treatment, original RSA research, decision-DH distinction, Python arithmetic and quantum-computing qualification. For this chapter, selected Sections 10.3 and 16.4 of Source 7 support RSA inversion and factoring; Exercises 11.5–11.6 support multiplicative ElGamal, its prime-order-group condition and the message-space caveat. The worked values, figures and programs are original teaching presentations. The following sources supply the scheme specification, original ElGamal and OAEP research, the qualified RSA-OAEP proof and selected implementation checks.

59. Kathleen Moriarty (editor), Burt Kaliski, Jakob Jonsson and Andreas Rusch, *PKCS #1: RSA Cryptography Specifications Version 2.2*, RFC 8017, November 2016, Informational IETF publication. Sections 3–5 specify key conditions, byte/integer conversions and primitives. Section 7.1 specifies RSAES-OAEP, its capacity, label, encoding, decoding and qualified proof history; Appendix B.2.1 defines MGF1. The described PKCS #1 usage has an empty label; the generic equations and library can represent labels more broadly. Source 5's collected Section V was also read for the historical arithmetic, without adopting its raw message processing as modern advice. <https://www.rfc-editor.org/info/rfc8017>.

Back to text: [1](#) [2](#) [3](#) [4](#)

60. Taher ElGamal, *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, *IEEE Transactions on Information Theory* 31(4), July 1985, pp. 469–472. The abstract and encryption section define the exponentiation, shared factor and ciphertext pair, and identify a relation exposed by reused encryption randomness. Its full nonzero-residue presentation and discussion of discrete logs are distinguished from the modern prime-order-group/DDH statement used here. Its signature construction is a separate scheme, not the encryption-reuse example in this chapter. [doi:10.1109/TIT.1985.1057074](https://doi.org/10.1109/TIT.1985.1057074).

Back to text: [1](#) [2](#)

61. Mihir Bellare and Phillip Rogaway, *Optimal Asymmetric Encryption*, EUROCRYPT '94 proceedings, Lecture Notes in Computer Science 950, Springer, published 1995, pp. 92–111. Selected

opening sections introduce the randomized encoding and ideal-hash setting. The 1994 conference and 1995 proceedings dates are distinguished. The original generic argument is not presented as the final adaptive chosen-ciphertext proof for RSA-OAEP; Sources 59 and 62 provide the later qualification. [doi:10.1007/BFb0053428](https://doi.org/10.1007/BFb0053428).

Back to text: [1](#)

62. Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval and Jacques Stern, *RSA-OAEP Is Secure under the RSA Assumption*, *Journal of Cryptology* 17, 2004, pp. 81–104; online publication 12 August 2002. The abstract and selected Sections 1–2 distinguish the earlier OAEP gap, partial-domain inversion, the RSA-specific result and the random-oracle/CCA setting. RFC 8017’s Section 7.1 discussion supplies the implementation-format qualification and non-tightness explanation. No full proof audit or concrete cost equality is claimed. [doi:10.1007/s00145-002-0204-y](https://doi.org/10.1007/s00145-002-0204-y).

Back to text: [1](#)

63. Daniel Bleichenbacher, *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1*, CRYPTO ’98, Lecture Notes in Computer Science 1462, Springer, 1998, pp. 1–12. Selected Sections 1–3 explain an adaptive oracle revealing conformance to the PKCS #1 v1.5 *encryption* format. This is not attributed to OAEP or RSA-PSS. The chapter explains the information-leak principle without asserting that every RSA service exposes this oracle. [doi:10.1007/BFb0055716](https://doi.org/10.1007/BFb0055716).

Back to text: [1](#)

64. Python Cryptographic Authority, *RSA*, `cryptography` 50.0.2 documentation. The key-generation, encryption, decryption, OAEP and MGF1 interfaces support the exact printed library example. Its empty label and explicit SHA-256 choices are retained. Both printed programs were executed independently of the typesetting check. Interface demonstrations and selected negative cases do not certify the library or define a complete application protocol. <https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/rsa/>.

Back to text: [1](#)

65. OpenSSL Project, collected openssl-4.0.3 source, selected passages of `crypto/rsa/rsa_oaep.c`: seed generation, both masks, decoding checks and constant-time helpers. C2SP Wycheproof, collected `testvectors_v1/rsa_oaep_2048_sha256_mgf1sha256_test.json`: public test keys and 37 valid/invalid cases. The chapter check follows their metadata, independently decodes the specified OAEP fields and compares the installed library's results. No upstream build, timing certification or complete source audit is claimed.
<https://github.com/openssl/openssl>;
<https://github.com/C2SP/wycheproof>.

Back to text: [1](#)

Chapter 12: Elliptic Curves

Sources 7, 18, 21, 22, 49, 51 and 58 supply mathematical background, the Koblitz proposal, modular arithmetic, the quantum-model qualification, HKDF and X25519. Selected Sections 1–2 of the collected Koblitz paper were reread for the group law and finite-field setting; its historical parameters are not deployment recommendations. RFC 7748 Sections 4.1, 5–5.2, 6.1 and 7 support the specified curve, subgroup order, scalar decoding, ladder, published values, noncanonical inputs, twist and output qualifications. The real and finite curves, worked coordinates, scalar table, diagrams and teaching program are original presentations. The five hexadecimal values in the library program are identified RFC test data. The following sources supplement that foundation.

66. Victor S. Miller, *Use of Elliptic Curves in Cryptography*, CRYPTO '85 proceedings, Springer, Lecture Notes in Computer Science 218, published 1986, pp. 417–426. The collected full text's group discussion and selected key-exchange passages support point addition, scalar multiplication and the historical proposal. The conference year, proceedings year and publisher's later online date are distinguished. Historical speed estimates and assumptions are not presented as current benchmarks or a general equivalence of discrete logarithms and computational Diffie–Hellman. [doi:10.1007/3-540-39799-X_31](https://doi.org/10.1007/3-540-39799-X_31).

Back to text: [1](#) [2](#)

67. Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby and Christopher A. Wood, *Hashing to Elliptic Curves*, RFC 9380, August 2023, Informational IRTF/CFRG publication. Selected Sections 2–3 distinguish forms, subgroup/cofactor notation, serialization, encodings, distribution properties and domain separation. The brief explanation does not implement a suite or substitute an improvised point search for one. <https://www.rfc-editor.org/info/rfc9380>.

Back to text: [1](#)

68. Certicom Research, *Standards for Efficient Cryptography 1 (SEC 1): Elliptic Curve Cryptography*, Version 2.0, 21 May 2009. Selected Sections 2.2.1, 2.3.3–2.3.4 and 3.2.2 specify the short Weierstrass equation and group law, point conversion and full public-key validation under the stated domain conditions. These are distinguished

from the X25519 input rules. The document's cover identifies Version 2.0; a search index's version label is not used as the authority.

<https://www.secg.org/sec1-v2.pdf>.

Back to text: [1](#) [2](#) [3](#)

69. Python Cryptographic Authority, *X25519 Key Exchange*, cryptography 50.0.2 documentation. The generation, private/public byte, raw serialization and exchange interfaces support the printed program. The installed version's wrong-length and all-zero-exchange behavior was checked separately; no general timing claim follows. The HKDF interface is documented in Source 49.
<https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/x25519/>.

Back to text: [1](#) [2](#)

70. OpenSSL Project, collected `openssl-4.0.3`, selected X25519 passages of `crypto/ec/curve25519.c`, including its shared-output zero test. C2SP Wycheproof, collected `testvectors_v1/x25519_test.json`, contains 518 selected computation cases with valid and acceptable classifications. Their metadata distinguish mathematical computation from permitted rejection of some inputs. The chapter check compares an independent public-data RFC calculation with these expected results and the installed library, recording accepted and rejected cases separately. No upstream build, full upstream-suite run, side-channel certification or complete implementation audit is claimed.
<https://github.com/openssl/openssl>;
<https://github.com/C2SP/wycheproof>.

Back to text: [1](#)

Chapter 13: Signatures, Certificates and Identity

Sources 7, 8, 10, 21, 22 and 59 support the security background, key management, service identity, modular arithmetic, quantum-model qualification and RSA-PSS specification. Selected RFC 8017 Sections 8.1 and 9.1 were read for the PSS interface and encoding, separately from Chapter 11's OAEP reading. The toy Schnorr framing, small ECDSA calculation, nonce-recovery values, diagrams and explanations are original

teaching presentations. The Ed25519 program identifies published RFC test data. Correctness, forgery resistance, implementation behavior, identity binding, policy and legal interpretation are kept distinct.

71. Shafi Goldwasser, Silvio Micali and Ronald L. Rivest, *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks*, *SIAM Journal on Computing* 17(2), April 1988, pp. 281–308. The collected full text’s abstract and selected introductory/definition passages explain new-message forgery under adaptive signing requests. This is the signature paper, not the similarly abbreviated GMR zero-knowledge paper. The historical scheme’s assumptions and proof are not attributed to every later signature family. [doi:10.1137/0217017](https://doi.org/10.1137/0217017).

Back to text: [1](#)

72. Claus-Peter Schnorr, *Efficient Signature Generation by Smart Cards*, *Journal of Cryptology* 4, 1991, pp. 161–174. Selected Section 2 passages support subgroup-based signing, challenge/response reasoning and the reused-secret relation. The chapter uses a positive-public-key convention and its own disclosed framing; it does not reproduce the paper’s inverse-public-key notation, historical parameter recommendations or wire format. [doi:10.1007/BF00196725](https://doi.org/10.1007/BF00196725).

Back to text: [1](#)

73. National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, FIPS 186-5, 3 February 2023. Selected Sections 3–7, especially 6.3–6.4, support algorithm roles, scalar generation and ECDSA’s signing/verification conditions. The public record’s 12 May 2025 potential-updates notice was checked on 20 October 2026; its spreadsheet was not accessible through the research tools. The explanation uses the conventional equations with explicit nonzero and bit-length conditions, not OCR-corrupted inequalities or a claim of compliance certification. [doi:10.6028/NIST.FIPS.186-5](https://doi.org/10.6028/NIST.FIPS.186-5); status: <https://csrc.nist.gov/pubs/fips/186-5/final>.

Back to text: [1](#)

74. Simon Josefsson and Ilari Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, January 2017, Informational

IRTF/CFRG publication. Selected Sections 5.1.2, 5.1.5–5.1.7 and 8.1–8.6 specify encoding, seed processing, deterministic signing, scalar range, verification equations and variant distinctions. Section 7.1, Test 2, supplies the printed seed, public key, message and signature. The permissive cofactor equation and sufficient stricter equation are distinguished; no universal acceptance-policy equivalence is claimed. <https://www.rfc-editor.org/info/rfc8032>.

Back to text: [1](#) [2](#) [3](#)

75. Thomas Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, RFC 6979, August 2013, Informational Independent Submission. Selected Sections 3.2–3.4 explain secret-dependent HMAC processing and unchanged verification. It does not prescribe a constant scalar or derive a secret from the public message alone. <https://www.rfc-editor.org/info/rfc6979>.

Back to text: [1](#)

76. David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley and William Polk, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 5280, May 2008, Standards Track. Selected Sections 3, 4.1–4.2, 5 and 6 support certificate fields, CA/key-usage constraints, critical extensions, revocation and trust-anchor-based path validation. Service identity is checked separately under Source 10. The chapter explains the model rather than implementing a complete current application profile or every subsequent update. <https://www.rfc-editor.org/info/rfc5280>.

Back to text: [1](#)

77. Python Cryptographic Authority, `cryptography` 50.0.2 documentation, entries *Ed25519 Signing*, *Elliptic Curve Cryptography* and *RSA*. Documented byte interfaces, verification failures, ECDSA DER encoding and explicit RSA-PSS parameters support the examples and selected checks. The printed program uses plain Ed25519, not an inferred prehash variant. Source reading and numerical tests do not certify platform timing or regulatory approval. <https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/ed255>

[19/](#);
<https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/ec/>;
RSA documentation is also listed in Source 64.

Back to text: [1](#)

78. OpenSSL Project, collected openssl-4.0.3, selected Ed25519 signing/verification passages of `crypto/ec/curve25519.c` and PSS decoding passages of `crypto/rsa/rsa_pss.c`. C2SP Wycheproof, selected files `testvectors_v1/ed25519_test.json`, `testvectors_v1/ecdsa_secp256r1_sha256_test.json` and `testvectors_v1/rsa_pss_2048_sha256_mgf1_32_test.json`, supply valid and invalid verification data. Their metadata and observed acceptance are recorded in the chapter check. These are selected files, not the entire upstream suite, a source build or a complete audit.
<https://github.com/openssl/openssl>;
<https://github.com/C2SP/wycheproof>.

Back to text: [1](#)

Chapter 14: Managing Keys

Source 8 is the main lifecycle reference. Selected SP 800-57 Part 1 Revision 5 Sections 5.2–5.5, 6, 7, 8.1.5, 8.2.2–8.2.3, 8.3 and 9.2 were reread for purpose separation, usage periods, key states, establishment, backup, archive, destruction, revocation and inventory. Its final-publication record and the separate December 2025 initial public draft of Revision 6 were checked; a draft is not treated as a replacement final standard. Sources 23–25 support the generation roles, Sources 49–51 key derivation and Source 46 account recovery. The lifecycle and envelope diagrams, document-service scenario and teaching program are original. They are not a normative module state machine or a complete storage protocol.

79. Morris Dworkin, *Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*, NIST SP 800-38F, December 2012. The specified AES KW and KWP constructions distinguish deterministic key protection and input-length rules from improvised encryption/MAC combinations. The final publication and NIST’s May 2026 revision-planning announcement were checked. The chapter does

not adopt the legacy TDEA method or infer current approval of arbitrary combinations from historical text. [doi:10.6028/NIST.SP.800-38F](https://doi.org/10.6028/NIST.SP.800-38F); revision context: <https://csrc.nist.gov/News/2026/techniques-for-key-wrapping>.

Back to text: [1](#)

80. Jim Schaad and Russ Housley, *Advanced Encryption Standard (AES) Key Wrap Algorithm*, RFC 3394, September 2002, Informational. Selected Section 2.2 specifies the round calculation, default integrity value and rejection; Section 4 supplies six public known-answer cases. The independent calculation check uses those vectors as test material, not secret deployment keys. Historical introductory approval language is not represented as present policy. <https://www.rfc-editor.org/info/rfc3394>.

Back to text: [1](#)

81. National Institute of Standards and Technology, *Security Requirements for Cryptographic Modules*, FIPS 140-3, 22 March 2019. The publication's scope and security areas support the distinction between an identified module's validated boundary and the security of a whole application. No named hardware product, configuration or service is certified by this book. [doi:10.6028/NIST.FIPS.140-3](https://doi.org/10.6028/NIST.FIPS.140-3).

Back to text: [1](#)

82. Ramaswamy Chandramouli and Eric Hibbard, *Guidelines for Media Sanitization*, NIST SP 800-88 Revision 2, September 2025, especially selected Section 3.2 passages on cryptographic erase and its preconditions. The official FAQ dated 16 July 2026, selected Questions 5–9, clarifies boundaries, verification, validation and storage limitations. Their scope is distinguished from deleting a high-level program variable or claiming control of an attacker's copies. [doi:10.6028/NIST.SP.800-88r2](https://doi.org/10.6028/NIST.SP.800-88r2); FAQ: <https://csrc.nist.gov/files/pubs/sp/800/88/r2/final/docs/sp800-88r2-faq.pdf>.

Back to text: [1](#)

83. Python Cryptographic Authority, *cryptography* 50.0.2 documentation, *Key Wrapping* and *Authenticated Encryption*, entries

for `aes_key_wrap`, `aes_key_unwrap`, `InvalidUnwrap` and `AESGCM`. Python 3.12's `os.urandom` documentation specifies the operating-system byte interface. These support the exact printed program; the example is not a persistent key service or a zeroization demonstration. <https://cryptography.io/en/50.0.2/hazmat/primitives/keywrap/>; <https://cryptography.io/en/50.0.2/hazmat/primitives/aead/>; <https://docs.python.org/3.12/library/os.html#os.urandom>.

Back to text: [1](#)

84. OpenSSL Project, collected `openssl-4.0.3`, selected KW passages of `crypto/modes/wrap128.c`: length checks, default integrity value, AES-block/counter ordering and checked unwrapping. Selected collected PyCA keywrap bindings identify the library interface. C2SP Wycheproof files `testvectors_v1/aes_wrap_test.json` and `testvectors_v1/aes_kwp_test.json` supply selected valid, invalid and acceptable computation cases, whose classifications are retained in the check record. The chapter check independently reconstructs the public KW calculations and compares selected negative cases through the installed library. No upstream build, full suite, entropy-source validation, hardware audit or complete key-management-system certification is claimed. <https://github.com/openssl/openssl>; <https://github.com/pyca/cryptography>; <https://github.com/C2SP/wycheproof>.

Chapter 15: Securing a Connection with TLS 1.3

The explanations and two diagrams are original teaching material. Source 10 supplies the service-identity rules in Sections 4 and 6 of RFC 9525. Sources 49–51 support the earlier HKDF explanation. The chapter distinguishes the current base specification, a historical trace, the version modeled in the journal analysis and selected implementation passages. No complete TLS implementation, upstream build or full conformance audit is claimed.

85. Eric Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 9846, July 2026. Sections 1.2–2 describe the relationship to RFC 8446, negotiation and exchange modes; Sections 4.1–4.5 cover

transcript, parameters and authentication; Sections 4.7, 5–8 cover tickets, updates, records, derivation and replay. Selected Appendix F discussions support the stated security boundaries. This backward-compatible update obsoletes the August 2018 RFC 8446 without changing the TLS 1.3 version number. The chapter follows its term main secret while retaining the specified byte labels. Earlier Source 9 introduces channel goals only. <https://www.rfc-editor.org/info/rfc9846>; predecessor: <https://www.rfc-editor.org/info/rfc8446>.

Back to text: [1](#) [2](#) [3](#) [4](#) [5](#)

86. Martin Thomson, *Example Handshake Traces for TLS 1.3*, RFC 8448, January 2019, Informational. Section 3 supplies the public X25519, SHA-256, HKDF, RSA-PSS and AES-GCM trace inputs and outputs used for selected checks. The printed example starts from its shared secret and supplied transcript digests, then reproduces selected directional secrets, keys, IVs and one record. The chapter check also reconstructs those digests from the published handshake bytes and checks additional trace operations. These are historical test credentials and values, not a present-day trust store or approved deployment configuration. <https://www.rfc-editor.org/info/rfc8448>.

Back to text: [1](#)

87. Benjamin Dowling, Marc Fischlin, Felix Günther and Douglas Stebila, *A Cryptographic Analysis of the TLS 1.3 Handshake Protocol*, Journal of Cryptology 34, article 37, published 30 July 2021. Abstract, Sections 1.3, 3–4 and the scope statements introducing the analyses distinguish full, PSK and PSK with ephemeral agreement, stage authentication, forward secrecy and replay properties. Its analyzed standard is RFC 8446. The model’s long-term and stage-key compromise capabilities do not include unrestricted exposure of ephemeral randomness or session state. The chapter does not extend the result to every implementation, the entire record layer or all later specification changes. [doi:10.1007/s00145-021-09384-1](https://doi.org/10.1007/s00145-021-09384-1).

Back to text: [1](#)

88. OpenSSL Project, collected `openssl-4.0.3`: selected TLS 1.3 secret/key derivation and update passages in `ssl/tls13_enc.c`, nonce/header handling in `ssl/record/methods/tls13_meth.c`, and

message transitions in `ssl/statem/statem_clnt.c` and `ssl/statem/statem_srvr.c`. These source routes are not a complete audit. Python 3.12's `hmac.digest` and `hashlib.sha256` support the printed derivation; PyCA cryptography 50.0.2 documents AESGCM's byte interface and InvalidTag failure. Its HKDFExpand interface supplies an additional comparison in the selected checks.

<https://github.com/openssl/openssl>;

<https://docs.python.org/3.12/library/hmac.html>;

<https://docs.python.org/3.12/library/hashlib.html>;

<https://cryptography.io/en/50.0.2/hazmat/primitives/aead/>;

<https://cryptography.io/en/50.0.2/hazmat/primitives/key-derivation-functions/>.

Back to text: [1](#)

Chapter 16: Secure Messages and Changing Keys

The scenarios, notation table, three-message diagram and teaching programs are original. The chapter's initial-agreement explanation is explicitly classical X3DH before the separate PQXDH discussion. Sources 49–51 support HKDF; Chapters 7, 12–15 supply AEAD, curve, authentication, key-management and channel foundations. Current project specifications are distinguished from the historical accepted research manuscripts. The checks do not implement the full Signal wire protocol or claim a new proof of its security.

89. Trevor Perrin (editor), Moxie Marlinspike and Rolfe Schmidt (revision 3 onward), *The Double Ratchet Algorithm*, Revision 4, 4 November 2025. Sections 2–3 define chains, ratchet initialization, directional state, headers, skipped keys and failed-decryption state handling; Sections 5–7 define Sparse Post-Quantum and Triple Ratchet interfaces and PQXDH integration; Sections 8.1–8.2, 8.4 and 8.8–8.11 qualify deletion, compromise recovery, retained keys and sparse-ratchet progress. Section 7.2 supplies the HMAC constant-byte and root-HKDF instantiation choices. The example's AES-GCM record is a separate teaching choice, not the recommended concrete Signal record format. Revision and page contents checked 21 October 2026.
<https://signal.org/docs/specifications/doubleratchet/>.

Back to text: [1](#) [2](#) [3](#)

90. Ehren Kret and Rolfe Schmidt, *The PQXDH Key Agreement Protocol*, Revision 3, 24 May 2023, last updated 23 January 2024. Sections 2–3 define key roles, encodings, the KEM contribution, derivation, prekey signatures, fallback and processing; Sections 4.1–4.3 and 4.6–4.12 distinguish authentication, replay, compromise, passive later quantum analysis, active quantum attackers, server trust and identity/key binding. The chapter does not silently replace the document's CRYSTALS-Kyber examples with ML-KEM or infer an application's current parameter set from this specification.
<https://signal.org/docs/specifications/pqxdh/>.

Back to text: [1](#)

91. Rolfe Schmidt, *The ML-KEM Braid Protocol*, Revision 1, 21 February 2025, last updated 26 September 2025. Sections 1–2.1 introduce SCKA, epoch outputs, the incremental KEM interface and

encoded chunks; Sections 3.1, 3.4 and 3.7–3.8 qualify vulnerable messages, traffic dependence, the reported symbolic ProVerif analysis and epoch representation. The reported proof concerns its model, not every implementation or a new proof of ML-KEM’s mathematical assumptions. <https://signal.org/docs/specifications/mlkembraid/>.

Back to text: [1](#) [2](#)

92. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt and Douglas Stebila, *A Formal Security Analysis of the Signal Messaging Protocol*, accepted manuscript associated with the 2017 IEEE European Symposium on Security and Privacy, pp. 451–466, DOI 10.1109/EuroSP.2017.27. The archive records acceptance in October 2016 and publication in 2017; the local source identifier’s 2016 date is not represented as the proceedings publication year. Selected Sections 2–4 introduce the abstract core and multi-stage compromise model; Sections 6–7 state limitations and scope, including authenticated prekey distribution, omitted skipped-key storage, multiple devices, privacy and implementation attacks. This historical analysis is not asserted to prove the later PQXDH or post-quantum ratchet revisions. [doi:10.1109/EuroSP.2017.27](https://doi.org/10.1109/EuroSP.2017.27); accepted source: <https://ora.ox.ac.uk/objects/uuid:b32f4e17-ceae-4c53-a54c-edeca815d827>.

Back to text: [1](#)

93. Katriel Cohn-Gordon, Cas Cremers and Luke Garratt, *On Post-Compromise Security*, 2016 IEEE 29th Computer Security Foundations Symposium, pp. 164–178. The abstract, introduction and Sections 2–3 distinguish weak and total compromise, later honest exchange, key continuity and modeled attacker capabilities. The result is not a guarantee of recovery while an adversary continues controlling all relevant endpoint state. [doi:10.1109/CSF.2016.19](https://doi.org/10.1109/CSF.2016.19); accepted source: <https://ora.ox.ac.uk/objects/uuid:241da365-1c73-4b6a-826c-f122c4c1e1b8>.

Back to text: [1](#)

94. Moxie Marlinspike and Trevor Perrin (editor), *The X3DH Key Agreement Protocol*, Revision 1, 4 November 2016. Sections 2–3 specify identities, signed and optional one-time prekeys, the three/four

DH outputs, domain-separated HKDF and associated identities; Sections 4.1–4.3 specify key-verification and replay limitations and the fresh reply-key requirement. This is used as a named classical foundation, not a description of every later deployed setup.

<https://signal.org/docs/specifications/x3dh/>.

Back to text: [1](#) [2](#)

95. Signal Messenger, *libsignal*, selected

`rust/protocol/src/ratchet/keys.rs` source viewed 21 October 2026: ChainKey’s separate HMAC input bytes, next-chain calculation, message-material derivation and its public chain-derivation test. The printed program matches the published next-chain vector; it distinguishes the abstract HMAC message output from the implementation’s subsequently derived cipher and MAC keys. Source reading is limited to this route, not the whole repository or a deployment audit. Python 3.12’s HMAC interface and PyCA cryptography 50.0.2 X25519, HKDF and AESGCM interfaces support the teaching example and additional independent checks.

<https://github.com/signalapp/libsignal/blob/main/rust/protocol/src/ratchet/keys.rs>; <https://docs.python.org/3.12/library/hmac.html>; <https://cryptography.io/en/50.0.2/hazmat/primitives/asymmetric/x25519/>; <https://cryptography.io/en/50.0.2/hazmat/primitives/key-derivation-functions/>; <https://cryptography.io/en/50.0.2/hazmat/primitives/aead/>.

Back to text: [1](#)

Chapter 17: How Good Cryptography Fails

The nonce calculation, reduced RSA interval oracle, CBC byte relation, timing-noise model and field encodings are original teaching examples. Their assumptions and limited conclusions are stated beside the calculations. Earlier chapters supply the construction definitions. Historical research below is used within its reported versions, observations and experimental conditions; these are not assertions about every current deployment.

96. Daniel Bleichenbacher, *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1*, CRYPTO '98, Springer, 1998, pp. 1–12. The collected full text's Sections 2–3 specify block conformance, the oracle and interval narrowing; Sections 6–7 qualify the constructed-oracle experiments. The decimal model in this book is not a conforming PKCS #1 block or a full attack implementation. [doi:10.1007/BFb0055716](https://doi.org/10.1007/BFb0055716).

Back to text: [1](#)

97. Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, CRYPTO '96, Springer, 1996, pp. 104–113. The introduction and private-exponentiation discussion explain timing dependence even when functional outputs are correct. The book's two-mean noise arithmetic is an original illustrative model, not a reported experimental result. [doi:10.1007/3-540-68697-5_9](https://doi.org/10.1007/3-540-68697-5_9).

Back to text: [1](#)

98. David Brumley and Dan Boneh, *Remote Timing Attacks Are Practical*, 12th USENIX Security Symposium, 2003. The collected 13-page author paper's introduction, OpenSSL arithmetic discussion and blinding countermeasure section support the local-network scope and named historical versions. The research does not establish a universal internet query cost or a vulnerability in every later release. <https://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html>.

Back to text: [1](#)

99. Nadhem J. AlFardan and Kenneth G. Paterson, *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*, IEEE Symposium on Security and Privacy, 2013, pp. 526–540. Selected construction, attack and countermeasure passages distinguish TLS CBC's padding/MAC

processing from the book's PKCS #7-style teaching tail. The historical discussion is separate from TLS 1.3 AEAD; old deployment advice is not adopted as current guidance. [doi:10.1109/SP.2013.42](https://doi.org/10.1109/SP.2013.42).

Back to text: [1](#)

100. Daniel J. Bernstein and colleagues, *KyberSlash: Exploiting Secret-Dependent Division Timings in Kyber Implementations*, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2), pp. 209–234. The collected 30-page author version is dated 15 January 2025. Selected introductory, attack-model, platform, verification-boundary and appendix passages support the distinction between decoding and decapsulation's re-encryption, and between correct arithmetic and compiler/processor timing. The reduced formula assumes a canonical coefficient. This is not a claim that the lattice problem or every ML-KEM implementation was broken. [doi:10.46586/tches.v2025.i2.209-234](https://doi.org/10.46586/tches.v2025.i2.209-234); author project: <https://kyberslash.cr.yp.to/papers.html>.

Back to text: [1](#)

101. Sazzadur Rahaman and colleagues, *CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects*, *ACM CCS*, 2019, pp. 2455–2472. The collected paper's abstract, introduction and analysis discussion support the distinction between detecting specified API misuse and proving a primitive or whole application secure. The chapter makes no claim to have run this tool or reproduced its evaluation. [doi:10.1145/3319535.3345659](https://doi.org/10.1145/3319535.3345659).

Back to text: [1](#) [2](#)

102. OpenSSL Project, selected collected `crypto/bn/bn_blind.c` paths: `BN_BLINDING_create_param`, `conversion` and `inverse` routines. Reading covers random invertible-factor preparation, public exponentiation and operand conversion/removal. Python Cryptographic Authority, selected collected `docs/hazmat/primitives/constant-time.rst`, especially the different-length versus equal-length scope of `bytes_eq`. This is source and interface reading, not a source build, upstream test run or compiled leakage audit. <https://github.com/openssl/openssl>; <https://cryptography.io/en/50.0.2/hazmat/primitives/constant-time/>.

Back to text: [1](#)

Additional sources for Chapter 1: Enigma and Alan Turing

These later institutional histories supplement the original publications listed with the opening chapter. The letter examples and consistency-search diagram are original explanations. The account distinguishes the earlier Polish achievement, the British crib-based Bombe, engineering and operational contributions, and the separate Lorenz/Colossus effort. It makes no precise counterfactual claim about how many years a single invention shortened the war.

103. Jennifer Wilcox, *Solving the Enigma: History of the Cryptanalytic Bombe*, Center for Cryptologic History, National Security Agency, revised 2006. Selected passages in the 60-page public PDF, especially PDF pages 4–14 and 18–19, support the Polish foundations, Turing/Welchman reasoning, Harold Keen’s engineering, operators’ checking work and changes in the naval problem. The chapter does not use the account’s prototype-date typo or its speculative war-duration comparison.
https://www.nsa.gov/Portals/70/documents/about/cryptologic-heritage/historical-figures-publications/publications/wwii/solving_enigma.pdf.

Back to text: [1](#) [2](#)

104. The National Museum of Computing, Tony Sale: Codes and Ciphers series, *The Principle of the Enigma*, posted 19 November 2018. The rotor and reflector explanation supports changing substitutions, reciprocity at a fixed configuration and the no-self-encryption constraint. Four-letter examples in this book are not historical wiring or traffic. John Jackson’s museum account, *The Enigma Machine*, separately supports the equipment description and distinction among variants. Only selected passages are used; the stepping discussion is not a complete mechanical model. <https://www.tnmoc.org/notes-from-the-museum/2018/11/19/the-principle-of-the-enigma>; <https://www.tnmoc.org/bh-2-the-enigma-machine>.

Back to text: [1](#) [2](#) [3](#)

105. Magda Szkuta, *Polish Mathematicians and Cracking the Enigma*, British Library, 2 January 2018. Selected historical passages identify Rejewski, Różycki, Zygalski, French intelligence assistance, the 1932 breakthrough and the July 1939 transfer. The book does not adopt the article’s broad claims about all later traffic or precise years by which the war was shortened. <https://www.bl.uk/stories/blogs/posts/polish-mathematicians-and-cracking-the-enigma>.

Back to text: [1](#)

106. The National Museum of Computing, John Jackson, *Enter Turing and Welchman*, Bombe history, consulted 21 October 2026. The crib, menu and diagonal-board descriptions support the conceptual constraint-elimination explanation. The diagram in this book does not reproduce a museum illustration or implement a complete menu. <https://www.tnmoc.org/bh-6-enter-turing-and-welchman>.

Back to text: [1](#)

107. The National Museum of Computing, *Colossus*, gallery history, consulted 21 October 2026. The account identifies Lorenz teleprinter traffic and Tommy Flowers’s engineering work, supporting the distinction from the Enigma Bombe. Broad priority claims about the first computer are outside the book’s explanation. <https://www.tnmoc.org/colossus>.

Back to text: [1](#)

Chapter 18: Testing and Verified Software

Earlier Sources 15 and 34 supply the ChaCha20-Poly1305 construction and AEAD interface. The byte transformations, random-miss calculation, evidence comparison and specification-to-execution figure are original teaching material. The printed program identifies RFC 8439 Section 2.8.2's public known-answer data; its selected mutations are additional checks, not a claim to reproduce an attack. The local run and source inspection below are distinguished from upstream suites and proof checking.

108. C2SP, *Project Wycheproof*, collected README, AEAD schema, common result-label schema and

`testvectors_v1/aes_gcm_test.json` and

`testvectors_v1/chacha20_poly1305_test.json`, inspected 21

October 2026. The selected files contain 316 and 325 cases. This chapter selects 96-bit nonce and 128-bit tag groups: 197 AES-GCM and 316 ChaCha20-Poly1305 cases. Other profiles are explicitly excluded; no acceptable cases occur in the selection. Case meanings are interpreted with the interface and algorithm notes. These are two files, not the project's whole coverage.

<https://github.com/C2SP/wycheproof>.

Back to text: [1](#)

109. Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko and Benjamin Beurdouche, *HACL*: A Verified Modern Cryptographic Library*, ACM CCS, 2017, pp. 1789–1806. Selected introductory, verification-goal and translation passages in the collected author manuscript distinguish memory safety, functional correctness, secret independence and cryptographic security. This paper is used as historical foundations, not a description of every current source file.

[doi:10.1145/3133956.3134043](https://doi.org/10.1145/3133956.3134043).

Back to text: [1](#) [2](#)

110. Marina Polubelova and colleagues, *HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms)*, ACM CCS, 2020, pp. 899–918. The collected author preprint has placeholder publisher fields; the published citation is given separately. Selected PDF pages 1–4 and 14–15 support generic specialization, Low* contracts, compiler trust and historical integration. The paper's SIMD compiler limitations and measured performance are not presented as current

universal facts. The collected HACL* README supplies the present tool name KaRaMeL and project citation guidance.

[doi:10.1145/3372297.3423352](https://doi.org/10.1145/3372297.3423352); author version:

<https://eprint.iacr.org/2020/572>.

Back to text: [1](#)

111. Jonathan Protzenko and colleagues, *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*, IEEE Symposium on Security and Privacy, 2020, pp. 983–1002. Selected introduction, Section II and Section III passages in the collected 20-page author copy distinguish agility, multiplexing, verification properties, trusted specifications/tools and unverified callers. The publication identity and DOI follow the authors’ ePrint metadata. Selected collected `dist/gcc-compatible/EverCrypt_Chacha20Poly1305.c` paths show vector/scalar dispatch, without building or proof-checking those paths. [doi:10.1109/SP40000.2020.00114](https://doi.org/10.1109/SP40000.2020.00114); author record: <https://eprint.iacr.org/2019/757>; source: <https://github.com/hacl-star/hacl-star>.

Back to text: [1](#) [2](#)

112. Python Cryptographic Authority, *Authenticated Encryption*, `cryptography` 50.0.2 documentation, ChaCha20Poly1305/AESGCM interface and InvalidTag behavior. The exact printed program was executed with 50.0.2; its installed backend reports OpenSSL 4.0.3. It matched the published complete RFC result and rejected the five specified changed inputs. Additional selected Wycheproof checks matched 372 valid and rejected 141 invalid cases; 128 other-profile cases were excluded. No separately collected library was compiled and no upstream conformance, leakage or proof suite was run. Runtime identity, vector-file hashes, case selection and check results are retained in the accompanying research record. <https://cryptography.io/en/50.0.2/hazmat/primitives/aead/>.

Back to text: [1](#) [2](#)

113. Dongpeng Xu, Jiang Ming and Dinghao Wu, *Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping*, IEEE Symposium on Security and Privacy, 2017, pp. 921–937. The collected paper’s abstract and introductory method

description support the limited comparison with reference implementations and function identification. The chapter does not reproduce its evaluation or extend its results to safe application use. [doi:10.1109/SP.2017.56](https://doi.org/10.1109/SP.2017.56).

Back to text: [1](#)

114. Yikun Hu and colleagues, *Binary Cryptographic Function Identification via Similarity Analysis with Path-Insensitive Emulation*, *Proceedings of the ACM on Programming Languages* 9, OOPSLA1, article 81, 2025. The supplied 31-page extended accepted author manuscript contains placeholder publication fields, so those fields are not used as final citation metadata. Selected opening passages describe interval-domain emulation and candidate identification, not an LLM method or a security proof for a complete application. No detection accuracy is generalized beyond the paper’s evaluation. [doi:10.1145/3720415](https://doi.org/10.1145/3720415).

Back to text: [1](#)

115. *REA—Reverse Engineer Anything*, collected repository, selected `src/filesystem/StableFileHash.ts`, `src/process/DetachedPgpSignature.ts` and `src/process/DetachedPgpSignature.test.ts`, reread 21 October 2026. The file-digest checks, pinned primary-key identity, detached binary-signature verification and defined negative cases support the short optional application example. This is source and test-definition inspection, not execution of the upstream application suite, a full filesystem proof or a certification of its dependencies. The chapter does not identify REA as an implementation of CryptoHunt or BinCrypto. <https://github.com/morluto/rea>.

Back to text: [1](#)

Chapter 19: Quantum Computers and Post-Quantum Cryptography

Earlier Source 54 supplies the final NIST recommendations for KEM use, including multi-algorithm combinations. The small qubit-probability calculation, factorization of 15, lattice drawing, modulo-17 equations and exhaustive teaching program are original explanations. The small program is not an implementation of a quantum algorithm or a standardized post-quantum mechanism. Publication, selection and draft status below were checked on 21 October 2026.

116. Peter W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM Journal on Computing 26(5), 1997, pp. 1484–1509. The collected expanded author version is dated 25 January 1996; the earlier conference result appeared in 1994. Selected introductory, modular-exponentiation, order-finding and gcd passages support the algorithmic threat and its success conditions. This is not a resource estimate for present hardware. [doi:10.1137/S0097539795293172](https://doi.org/10.1137/S0097539795293172); author version: <https://arxiv.org/abs/quant-ph/9508027>.

Back to text: [1](#)

117. Lov K. Grover, *A Fast Quantum Mechanical Algorithm for Database Search*, ACM Symposium on Theory of Computing, 1996, pp. 212–219. Selected introductory and amplitude-amplification passages support the idealized square-root query scale and the need to arrange useful measurement probabilities. A query is not equated with a web request or a complete hardware operation. [doi:10.1145/237814.237866](https://doi.org/10.1145/237814.237866).

Back to text: [1](#) [2](#)

118. Oded Regev, *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography*, Journal of the ACM 56(6), article 34, 2009. The collected expanded author file is dated 2 May 2009; the original conference version appeared at STOC 2005. Selected introduction, noisy-equation definition and reduction-scope passages support the LWE explanation and original quantum-reduction qualification. The book does not extend that theorem to every module-based construction or claim that a reduction proves hardness unconditionally. [doi:10.1145/1568318.1568324](https://doi.org/10.1145/1568318.1568324).

Back to text: [1](#)

119. National Institute of Standards and Technology, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, FIPS 203, final 13 August 2024. Selected definitions, Section 3.2 and Table 1, Table 3, public interface requirements and Algorithm 18 support parameter-set roles, key/ciphertext sizes, the 32-byte secret, the small nonzero honest decapsulation-failure probability and decapsulation’s implicit rejection. An explicit error and a mismatch of honestly generated secrets are distinguished. The publication page’s 17 November 2025 planning note identifies a potential update; it is not a new final standard. No implementation validation is claimed.

[doi:10.6028/NIST.FIPS.203](https://nvlpubs.nist.gov/nistpubs/fips/fips203/final.pdf); status:
<https://csrc.nist.gov/pubs/fips/203/final>.

Back to text: [1](#) [2](#)

120. National Institute of Standards and Technology, *Module-Lattice-Based Digital Signature Standard*, FIPS 204, final 13 August 2024. Selected Sections 3.2–3.6 and Table 2 support the ML-DSA role, module-LWE and SelfTargetMSIS assumptions, context and input conditions, and ML-DSA-65’s 1,952-byte public key and 3,309-byte signature. The official page’s 31 July 2026 planning note records minor issues intended for a future update. The short-integer explanation in this chapter is conceptual, not the full security theorem.

[doi:10.6028/NIST.FIPS.204](https://nvlpubs.nist.gov/nistpubs/fips/fips204/final.pdf); status:
<https://csrc.nist.gov/pubs/fips/204/final>.

Back to text: [1](#) [2](#)

121. National Institute of Standards and Technology, *Stateless Hash-Based Digital Signature Standard*, FIPS 205, final 13 August 2024. Selected introductory, hash-tree, parameter and signing-usage passages support SLH-DSA’s role and the meaning and limits of stateless signing. The chapter does not describe a bare public message digest as a signature or infer unlimited unconditional signing security.

[doi:10.6028/NIST.FIPS.205](https://nvlpubs.nist.gov/nistpubs/fips/fips205/final.pdf); status:
<https://csrc.nist.gov/pubs/fips/205/final>.

Back to text: [1](#)

122. Joppe Bos and colleagues, *CRYSTALS—Kyber: A CCA-Secure Module-Lattice-Based KEM*, IEEE European Symposium on Security

and Privacy, 2018, pp. 353–367. The collected proceedings full text’s abstract and introductory design discussion identify the module-lattice KEM and chosen-ciphertext goal. Its historical construction and performance comparisons are distinguished from the final ML-KEM specification and current deployment measurements.
[doi:10.1109/EuroSP.2018.00032](https://doi.org/10.1109/EuroSP.2018.00032).

Back to text: [1](#)

123. Léo Ducas and colleagues, *CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme*, IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018(1), pp. 238–268. The collected publisher full text’s abstract and introductory design discussion support the research lineage and lattice-signature role. Its historical performance and implementation claims are not generalized to all ML-DSA implementations. [doi:10.13154/tches.v2018.i1.238-268](https://doi.org/10.13154/tches.v2018.i1.238-268).

Back to text: [1](#)

124. NIST, official publication records for FIPS 203–205 and SP 800-227, checked 21 October 2026. NIST’s *HQC Announced as a 4th Round Selection*, 11 March 2025, establishes selection for standardization, not an additional final FIPS. NIST IR 8547, *Transition to Post-Quantum Cryptography Standards*, 12 November 2024, was listed as an initial public draft at this check; proposed transition material is not represented as a final worldwide deadline. The book’s

$S + M > Q$

relation is an explained planning model, not a prediction derived from that draft. <https://csrc.nist.gov/News/2025/hqc-announced-as-a-4th-round-selection>; <https://csrc.nist.gov/pubs/ir/8547/ipd>; <https://csrc.nist.gov/pubs/sp/800/227/final>.

Back to text: [1](#) [2](#)

125. Open Quantum Safe, collected *liboqs* repository, selected README.md, docs/algorithms/kem/ml_kem.md and src/kem/ml_kem/kem_ml_kem_768.c, inspected 21 October 2026. The wrapper names FIPS 203 and separates key generation, encapsulation and decapsulation, with explicit length fields and function pointers. The README qualifies research and integration scope. This is source-

interface reading; no library build, upstream test suite, validation or leakage audit is claimed. <https://github.com/open-quantum-safe/liboqs>.

Back to text: [1](#)

Further sources for Chapter 19: quantum foundations and hardware

These sources support the expanded route through quantum information, circuit behavior and physical error handling. The numerical circuit examples, classical programs, statistical calculations and hardware schematic are original teaching material. The discussion identifies manufacturer documentation, peer-reviewed papers and an author preprint separately. No hardware experiment or commercial-service benchmark was performed for this book.

126. Richard P. Feynman, *Simulating Physics with Computers*, International Journal of Theoretical Physics 21, 1982, pp. 467–488. The simulation motivation is distinguished from a general speedup claim. The publisher records the June 1982 publication; the historical argument concerns representing and simulating physical systems, not the capabilities of a current device. [doi:10.1007/BF02650179](https://doi.org/10.1007/BF02650179); institutional record: <https://authors.library.caltech.edu/records/a1kgk-xyk45>.

Back to text: [1](#)

127. IBM Quantum Learning, John Watrous, *Basics of Quantum Information*, selected single-system quantum-information passages; and *General Formulation of Quantum Information, Density Matrix Basics*, consulted 21 October 2026. State vectors, complex amplitudes, normalization, plus/minus states and the standard-basis measurement rule support the notation. The density-matrix treatment supports the distinction between pure states, classical mixtures and local subsystems. The chapter’s complex-number values and arithmetic are original examples.
<https://quantum.cloud.ibm.com/learning/en/courses/basics-of-quantum-information/single-systems/quantum-information>;
<https://quantum.cloud.ibm.com/learning/en/courses/general->

[formulation-of-quantum-information/density-matrices/density-matrix-basics](#).

Back to text: [1](#) [2](#)

128. IBM Quantum Learning, *Basics of Quantum Information*, selected quantum-circuit and limitations passages, including the no-cloning argument; and *Foundations of Quantum Error Correction*, introductory repetition-code discussion, consulted 21 October 2026. The chapter's figures and state calculations are original presentations. A restricted bit-flip code is not presented as protection against all quantum errors.
<https://quantum.cloud.ibm.com/learning/en/courses/basics-of-quantum-information/quantum-circuits/introduction>;
<https://quantum.cloud.ibm.com/learning/en/courses/basics-of-quantum-information/quantum-circuits/limitations-on-quantum-information>;
<https://quantum.cloud.ibm.com/learning/en/courses/foundations-of-quantum-error-correction/correcting-quantum-errors/introduction>.

Back to text: [1](#)

129. Qiskit contributors, *Qiskit*, collected public repository, inspected 21 October 2026. Selected `README.md`,
`qiskit/circuit/library/standard_gates/h.py`,
`qiskit/circuit/library/standard_gates/x.py` and
`qiskit/quantum_info/states/statevector.py` support the gate definitions, index convention and statevector initialization interface. This is selected source reading, not a build or upstream-suite run. The printed four-amplitude program is independent teaching code, not execution of the Qiskit package. <https://github.com/Qiskit/qiskit>.

Back to text: [1](#)

130. IBM, *IBM Debuts Next-Generation Quantum Processor & IBM Quantum System Two*, 4 December 2023; and IBM Quantum Documentation, *Processor Types*, consulted 21 October 2026. The first source identifies the launch configuration with three Heron processors, cooling/control infrastructure and classical runtime. The second distinguishes Heron revisions, including r1's 133 qubits and later 156-qubit devices. The book does not describe an old revision as the

newest processor or sum several chips into one demonstrated logical register. <https://newsroom.ibm.com/2023-12-04-IBM-Debuts-Next-Generation-Quantum-Processor-IBM-Quantum-System-Two,-Extends-Roadmap-to-Advance-Era-of-Quantum-Utility>; <https://quantum.cloud.ibm.com/docs/en/guides/processor-types>.

Back to text: [1](#)

131. Dorit Aharonov and Michael Ben-Or, *Fault-Tolerant Quantum Computation with Constant Error*, ACM STOC, 1997, pp. 176–188; expanded author manuscript *Fault-Tolerant Quantum Computation with Constant Error Rate*, arXiv, 1999. Selected expanded introduction and noise-model passages describe constant-threshold fault tolerance, locality assumptions and permitted correlation qualifications. The threshold is not generalized to every physical device, code or attack model. Historical parameters are not adopted as current engineering estimates. [doi:10.1145/258533.258579](https://doi.org/10.1145/258533.258579); collected expanded version: <https://arxiv.org/abs/quant-ph/9906129>.

Back to text: [1](#)

132. Qedma, *About Us*, co-founder roles; Technion Department of Physics, *Netanel Lindner*, faculty research profile, consulted 21 October 2026. These institutional/company records support affiliations and the identification of Lindner as CTO, Dorit Aharonov as CSO and Asif Sinay as CEO. Roles are contextual information, not independent validation of product-performance claims. <https://www.qedma.com/about/>; <https://phys.technion.ac.il/en/component/adm/person/641>.

Back to text: [1](#)

133. Netanel H. Lindner and Terry Rudolph, *A Photonic Cluster State Machine Gun*, Physical Review Letters 103, 113602, 2009. Selected abstract, opening source proposal and error-localization discussion in the collected accepted author version support the bounded description of a photonic cluster-state source. Its proposal and model-dependent estimates are distinguished from a completed large fault-tolerant machine and from later QESEM experiments. [doi:10.1103/PhysRevLett.103.113602](https://doi.org/10.1103/PhysRevLett.103.113602); full author version: <https://arxiv.org/abs/0810.2587v2>.

Back to text: [1](#)

134. Dorit Aharonov, Netanel H. Lindner and the other listed co-authors, *Reliable High-Accuracy Error Mitigation for Utility-Scale Quantum Circuits*, arXiv:2508.10997v2, first submitted 2025, revised 2026. The collected versioned PDF and selected introduction, overview, bias/runtime and drift passages describe characterization, quasi-probabilistic mitigation and the distinction between unbiased and controlled-bias configurations. The cited record is an author preprint, not a claimed final journal publication. The book's weighted-mean numbers are original teaching data, not a reproduced benchmark. <https://arxiv.org/abs/2508.10997v2>.

Back to text: [1](#)

135. Ryuji Takagi, Suguru Endo, Shintaro Minagawa and Mile Gu, *Fundamental Limits of Quantum Error Mitigation*, npj Quantum Information 8, article 114, 22 September 2022. Selected abstract, introduction and estimator-spread discussion derive lower bounds within a defined protocol/noise framework. The limits are not interpreted as an unconditional impossibility of useful mitigation. [doi:10.1038/s41534-022-00618-z](https://doi.org/10.1038/s41534-022-00618-z).

Back to text: [1](#)

136. IBM Quantum Documentation, *QESEM—A Qiskit Function by Qedma*; and Qedma, *QESEM Best Practices*, consulted 21 October 2026. Selected function description and runtime/precision passages identify the service interface, characterization, circuit/observable dependence and additional sampling costs. This is documentation reading, not a service execution, subscription recommendation or independent product certification. <https://quantum.cloud.ibm.com/docs/en/guides/qedma-qesem>; https://docs.qedma.io/best_practices/.

Back to text: [1](#) [2](#)

Chapter 20: Proving and Computing with Private Data

The transcript, secret-sharing points, shared-arithmetic numbers, scaling calculations and diagrams are original teaching material. Their checks

establish selected arithmetic and integration properties, not secure implementations of the complete protocols. Earlier chapters supply the group, signature, encoding and key-management foundations.

137. Shafi Goldwasser, Silvio Micali and Charles Rackoff, *The Knowledge Complexity of Interactive Proof Systems*, SIAM Journal on Computing 18(1), February 1989, pp. 186–208. The collected 23-page journal scan was read visually for the introduction, interactive-proof properties and Section 3’s simulation/indistinguishability framework, especially PDF pages 1, 4–7. Its scanned equations are not claimed to be fully text-indexed. Honest-verifier transcript simulation in this chapter is identified separately from general malicious-verifier zero knowledge. [doi:10.1137/0218012](https://doi.org/10.1137/0218012).

Back to text: [1](#)

138. Feng Hao, *Schnorr Non-interactive Zero-Knowledge Proof*, RFC 8235, September 2017, selected Sections 2.1–2.3 and security discussion. The RFC uses a subtraction-response convention; this chapter explicitly defines a plus-response convention and its matching check. The tiny group and challenge space are original teaching choices, not adoption of historical parameter recommendations. The RFC is informational, not an Internet Standard. <https://www.rfc-editor.org/rfc/rfc8235>.

Back to text: [1](#)

139. Jens Groth, *On the Size of Pairing-Based Non-interactive Arguments*, EUROCRYPT 2016, Springer LNCS 9666, pp. 305–326. Selected abstract, introduction and setup/argument definitions support the three-group-element construction, arithmetic-circuit scope, common reference string and witness-privacy distinction. The size statement is not generalized to all proof systems or every component of verification work. [doi:10.1007/978-3-662-49896-5_11](https://doi.org/10.1007/978-3-662-49896-5_11).

Back to text: [1](#)

140. Adi Shamir, *How to Share a Secret*, MIT-LCS-TM-134, May 1979; the inner author page is dated April 1979. Selected scanned PDF pages 2–7, especially the polynomial construction and uniform-field argument on pages 5–6, were read as page images. This collected

technical report is distinguished from the later typeset Communications of the ACM article. The example's coefficients, identifiers and interpolation arithmetic are original.
<https://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-134.pdf>;
institutional record:
<https://dspace.mit.edu/entities/publication/91c51bfc-5678-409c-80c5-44ce8fcdbadf>.

Back to text: [1](#)

141. Chelsea Komlo and Ian Goldberg, *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*, SAC 2020 proceedings, published 2021, Springer LNCS 12804, pp. 34–65. The collected author paper's abstract and introductory coordination/adversary discussion support the research motivation. Preprocessing and signing-round descriptions are distinguished from the later two-round RFC interface. [doi:10.1007/978-3-030-81652-0_2](https://doi.org/10.1007/978-3-030-81652-0_2).

Back to text: [1](#)

142. Deirdre Connolly, Chelsea Komlo, Ian Goldberg and Christopher A. Wood, *The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures*, RFC 9591, June 2024, informational CFRG/IRTF publication. Selected Sections 4–5, 7 and Appendix C identify nonce generation, binding factors, commitments, message binding, security assumptions and the trusted-dealer alternative. General distributed key generation is outside the main specification. The document and current publication record were consulted 21 October 2026; no upstream conformance suite was run.
<https://www.rfc-editor.org/rfc/rfc9591>.

Back to text: [1](#)

143. Ivan Damgård, Valerio Pastro, Nigel Smart and Sarah Zakarias, *Multiparty Computation from Somewhat Homomorphic Encryption*, CRYPTO 2012, Springer LNCS 7417, pp. 643–662. Selected abstract, introduction, preprocessing, authenticated-sharing and multiplication passages distinguish the active-adversary architecture from the book's unauthenticated arithmetic model. The paper's historical timings are not presented as current benchmarks; security with abort is not

converted into guaranteed output delivery. [doi:10.1007/978-3-642-32009-5_38](https://doi.org/10.1007/978-3-642-32009-5_38).

Back to text: [1](#)

144. Craig Gentry, *A Fully Homomorphic Encryption Scheme*, Stanford University doctoral dissertation, September 2009. Selected abstract and opening homomorphic-evaluation/bootstrapping passages support the foundational construction and refresh idea. Required key information and assumptions are not omitted from the interpretation. The thesis's historical efficiency statements do not describe every modern implementation. <https://crypto.stanford.edu/craig/craig-thesis.pdf>.

Back to text: [1](#)

145. Jung Hee Cheon, Andrey Kim, Miran Kim and Yongsoo Song, *Homomorphic Encryption for Arithmetic of Approximate Numbers*, ASIACRYPT 2017, Springer LNCS 10624, pp. 409–437. Selected abstract, introduction and rescaling/precision passages support the distinction between approximate arithmetic and exact modular computation. The chapter's decimal rounding example is unencrypted teaching arithmetic and is not a reproduction of the complete scheme or its ciphertext-error theorem. [doi:10.1007/978-3-319-70694-8_15](https://doi.org/10.1007/978-3-319-70694-8_15).

Back to text: [1](#)

146. Tariq Bontekoe, Dimka Karastoyanova and Fatih Turkmen, *Verifiability for Privacy-Preserving Computing on Distributed Data—A Survey*, International Journal of Information Security 24, article 141, published 24 May 2025. Selected abstract, introductory distinctions and security/correctness definitions support separating confidentiality, active security and verification of outsourced/distributed work. The chapter does not reproduce the survey's evaluations or claim one universally best composition. [doi:10.1007/s10207-025-01047-7](https://doi.org/10.1007/s10207-025-01047-7).

Back to text: [1](#)

147. Weinan Liu and colleagues, *From Accuracy to Approximation: A Survey on Approximate Homomorphic Encryption and Its Applications*, Computer Science Review 55, article 100689, 2025; available online 22 November 2024. Selected opening scope and CKKS/rescaling/bootstrapping discussion support the

accuracy/resource context. The collected publisher article is 28 pages. Historical platform results are not generalized to present workloads. [doi:10.1016/j.cosrev.2024.100689](https://doi.org/10.1016/j.cosrev.2024.100689).

Back to text: [1](#)

148. Microsoft SEAL contributors, collected public repository, selected README.md, SECURITY.md and native/examples/5_ckks_basics.cpp, inspected 21 October 2026. Reading covers exact/approximate scheme roles, scale/level bookkeeping, unauthenticated ciphertexts, circuit-privacy limits and the decryption-feedback boundary. This is selected source/documentation reading, not a library build, upstream-suite run, secure deployment recommendation or encrypted performance measurement. <https://github.com/microsoft/SEAL>.

Back to text: [1](#)
