Miguel Vargas Martin, PhD, PEng

Practical Cryptography

From Classical Ciphers to Post-Quantum Standards

March 23, 2026

To my Father; I wish he'd seen all this...

Preface

Once upon a time
there was a man.
The more he learned,
the more ignorant he became.
In the end he was so ignorant
that he forgot his name.
Then he had nothing to say
and nowhere to go.
Somebody saw him the other day.
They say he's doing a postdoc.

The idea for this book had been quietly taking shape for several years. Many of its themes and exercises emerged during long runs in the cold or solitary rides on quiet roads, moments when sustained effort and silence left space for ideas to wander, sharpen, and recombine. Still, it was not until the winter of 2026, during a six month research leave and the accompanying teaching relief, that I finally had the time and mental space to sit down and write it.

This book is intended for instructors and students in post-secondary education, as well as practitioners who wish to deepen their understanding of cryptography through hands-on exploration. I have taught cryptography for more than two decades at Ontario Tech University, and my approach to the subject has evolved considerably. In my early years, I strongly favoured theory, often at the expense of hands-on work. Over time, largely in response to student feedback and my own evolving perspective, I shifted toward a more practical, programming-oriented approach. That change has been consistently well received by students at both the undergraduate and graduate levels and has shaped the philosophy behind this book.

This book emphasizes practical application, omitting exhaustive mathematical descriptions of the underlying algorithms, except in Part III (Post-Quantum Cryptography). Because these post-quantum algorithms are relatively new compared to their classical predecessors, Part III is designed to be somewhat self-contained, requiring no external references for a solid understanding of the material. Conversely, for the more established algorithms in Part II (Pre-Quantum Cryptography), I recommend using this text as a practical complement to a theoretical textbook that covers the underlying mathematics in detail.

The exercises collected here are drawn from assignments, quizzes, and examination questions that I have authored and used in my own courses over the years. I spent some time debating whether solutions should be included. Ultimately, I decided in their favour, not least because some of these problems (and occasionally their answers) already circulate online, with or without my consent.

The goal of this book is therefore not to provide a bank of ready-made assessment questions for reuse, but rather a set of guided exercises intended to foster deeper understanding. I hope instructors will adapt and modify these problems to suit their own contexts, and that students will engage with them as opportunities to experiment, make mistakes, and learn. With thoughtful variation, many of these exercises can also serve as the basis for new and effective evaluation material.

All programs presented in this book were tested in the environments listed below. The Python examples were verified using Python 3.11.13 together with the cryptography.io library (version 46.0.4). The Java examples were tested with Java 21.0.9+10-LTS using the Bouncy Castle cryptography API (version 1.83). Although these specific versions were used during development, the concepts and implementations discussed throughout the book should remain applicable to most recent versions of these platforms as well as to other comparable cryptographic libraries.

The complete source code accompanying the book is available at:
https://github.com/miguevmartin/Practical-Cryptography.

Finally, a brief note about authorship in an age where even writing about cryptography can involve a conversation with machines. During the preparation of this book I frequently consulted modern language models, including ChatGPT, Gemini,

and Claude. They proved helpful in polishing sections of what sometimes began as a rather tangled narrative, in generating code examples that I could later adapt or refine, and in helping me troubleshoot my own reasoning during those moments when I was convinced I already had everything under control. For Part III in particular, I also benefited from the excellent online lectures by Alfred Menezes on Kyber and Dilithium, as well as the relevant publications of the National Institute of Standards and Technology, including the associated Federal Information Processing Standards (FIPS) documents. These resources were valuable companions during the writing process. Nevertheless, the explanations, interpretations, and examples presented throughout this book reflect my own understanding of the material, and any remaining errors, omissions, or misplaced confidence are entirely my responsibility. If the process of writing this book has reinforced anything, it is that the more one learns about cryptography, the more one becomes aware of how much there is still to understand. If this book helps readers explore the practical side of cryptography with curiosity and care, then it will have served its purpose.

March 2026 *Miguel Vargas Martin*

Acknowledgements

Integrating my personal and professional lives has allowed this book to come to fruition. As a result, this work reflects the influence of many—not only colleagues and students, but the friends and loved ones who shared this journey. Attempting to list everyone by name is a risk I am hesitant to take, fearing I might realize an omission only after the book is in print.

I must, however, acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC). While they did not fund this book directly, their steadfast support of my research for almost three decades has provided the foundation upon which my professional career was built.

Finally, I am deeply indebted to those at the centre of my world: my wife and children, for the joy they bring me every day. And a special mention to my dear four-legged friend, François, whose constant, quiet presence by my side through nearly every page of this work provided the comfort I needed to see it to completion.

Contents

Part I Hidden in Plain Sight

1 Symbolic Geometric Ciphers 3
1.1 The Radar Cipher 3
1.2 Pigpen-ish Cipher (or an undisclosed cipher) 5

2 Steganography 7
2.1 Text Within Text 7
2.2 Text or Images in LSBs 8
2.3 Network Steganography 9

Part II Pre-Quantum Cryptography

3 Number Theory 15
3.1 Integers 15
3.2 Modular Arithmetic 16
3.3 Primes and Greatest Common Divisors 17
3.4 Greatest Common Divisors and Least Common Multiples 20
3.5 Multiplicative Inverse 21
3.6 Pre-Quantum One-Way Trapdoor Functions 21
3.6.1 Prime Factorization 22
3.6.2 Discrete Logarithms 22
3.7 Putting It All Together using SymPy 23

4 Symmetric Key Cryptography 29
4.1 Avalanche Effect 29
4.2 Advanced Encryption Standard 34
4.3 Key Wrapping 47

5 Digest Functions 49
5.1 Hash Functions 49
5.2 SHA-3 Hash and Extendable-Output Functions 53

5.3 Message Authentication Codes 55

6 Stream Ciphers and Random Numbers 61
6.1 Random Enough 61
6.2 Speed of Cryptographically Secure Generators 69

7 Public-Key Cryptography 71
7.1 RSA for Encryption 71
7.2 RSA for Digital Signatures 75

Part III Post-Quantum Cryptography

8 More Number Theory 83
8.1 Basic Number Theory for PQC 84
8.2 PQC One-Way Trapdoor Functions 86

9 ML-KEM Encapsulation 87
9.1 Encapsulation 89
9.2 Decapsulation 90
9.3 Key Generation 91
9.3.1 Seed generation and vector of polynomials **A** 91
9.3.2 Small vectors of polynomials **s** and **e** 92
9.3.3 Public key 92
9.4 ML-KEM Security Levels 92

10 ML-DSA Signature Standard 101
10.1 More Notation 101
10.2 ML-DSA Decomposition Algorithms 101
10.3 Signature Generation 110
10.3.1 Commitment generation 111
10.3.2 Challenge Derivation 111
10.3.3 Potential signature calculation 111
10.3.4 Rejection sampling 111
10.4 Signature Verification 112
10.4.1 Initial validation and derivations 112
10.4.2 Challenge derivation 112
10.4.3 Verification 113
10.5 Key Generation 113
10.5.1 Sampling seeds 113
10.5.2 Expanding the public matrix **A** 113
10.5.3 Generating the secret vectors $(\mathbf{s}_1, \mathbf{s}_2)$ 113
10.5.4 Computing the public vector **t** 114
10.5.5 Compressing the public key **t** 114
10.6 One More One-Way Trapdoor Function: I-MSIS 114
10.7 ML-DSA Security Levels 115

Part IV Requiem

11 Beyond 125

Index 129

Acronyms

2DES	Double DES
3DES	Triple DES
AES	Advanced Encryption Standard
ARC4	Alleged RC4
CBC	Cipher Block Chaining
CBD	Centered Binomial Distribution
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
CFB	Cipher Feedback Mode
CMAC	Cipher-based MAC
CRC	Cyclic Redundancy Check
CRQC	Cryptographically Relevant Quantum Computer
DCT	Discrete Cosine Transform
CTR	Cipher Counter Mode
D-MLWE	Decisional-MLWE problem
DES	Data Encryption Standard
DH	Diffie-Hellman Key Exchange
DSA	Digital Signature Algorithm
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
FCS	Frame Check Sequence
FIPS	Federal Information Processing Standards
FO	Fujisaki-Okamoto transform
GCM	Galois/Counter Mode
HMAC	Hash-based MAC
I-MSIS	Inhomogeneous MSIS
LCG	Linear Congruential Generator
LLM	Large Language Model
LSB	Least Significant Bit
LWE	Learning With Errors problem
MAC	Message Authentication Codes
MANET	Mobile Ad Hoc Network

MD5 Message-Digest 5
ML-DSA Module-Lattice-Based Digital Signature Standard
ML-KEM Module-Lattice-Based Key Encapsulation Mechanism Standard
MLWE Module LWE problem
MSIS Module Short Integer Solution
NIST National Institute of Standards and Technology
NTT Number Theoretic Transform
OFB Output Feedback Mode
PKC Public-Key Cryptography
PQC Post-Quantum Cryptography
RC4 Rivest Cipher 4
RSA Rivest-Shamir-Adleman Algorithm
RSA-OAEP Optimal Asymmetric Encryption Padding RSA
RSA-PSS Probabilistic Signature Scheme RSA
SHA Secure Hash Algorithm
SHAKE Secure Hash Algorithm Keccak
SLH-DSA Stateless Hash-Based Digital Signature Standard
USB Universal Serial Bus
XOF Extendable-Output Function
XOR Exclusive-OR
XTS-AES eXclusive-or Tweakable block cipher with AES

Part I
Hidden in Plain Sight

Chapter 1
Symbolic Geometric Ciphers

"I thought of so many possibilities that I put off considering it, but still thought it indelicate to show you that I knew your secret."

—*Fyodor Dostoevsky,* Crime and Punishment

A cryptography book would hardly be complete without a few classical message-coding examples. I hope the reader finds them fun and entertaining.

Definition 1.1 A *symbolic geometric cipher* is a coding technique in which plaintext is transformed into ciphertext composed of abstract or non-alphanumeric symbols (such as geometric shapes, glyphs, or icons), rather than letters or digits, with each symbol representing one or more plaintext elements according to a defined encoding rule.

1.1 The Radar Cipher

Exercise 1.1 You intercept the following sequence of characters from a suspicious channel. It is not clear whether the message is in English. What's the message? How secure is this cipher?

)135 (315) (180) 45) (225) 225) (270))180 225))225 (270) 45))270 225))45

Solution. I came up with this cipher for an assignment in one of my cryptography courses. This cipher uses a monoalphabetic substitution technique in which each character in the plaintext is always mapped to the same symbol in the ciphertext (in this case, number and brackets). Deciphering it is hard for a number of reasons. First, the ciphertext is too small for statistical analysis. Despite the length of the ciphertext being too small, not knowing the language of the coded message makes things harder as different languages' distributions would need to be used for statistical analysis. The solution is in Spanish: *"solamente una vez"*, and can be derived from Figure 1.1. I call this the *Radar cipher*.

Let's see how many possible keys there are. In the key instance of Figure 1.1, the letters of the alphabet are arranged in a clock-wise spiral fashion along the three concentric rings, and the numbers refer to the angles of the 4 axes relative to the vertical axis at 12 o'clock. However, a key could be formed by arranging the letters of the

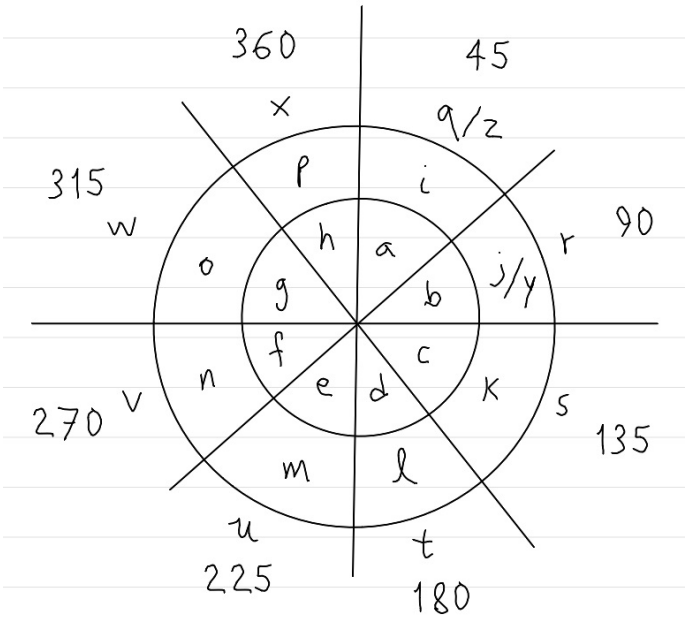


Fig. 1.1 A key instance of the Radar cipher.

alphabet and the numbers in a random fashion. If we had 26 buckets, one for each of the 26 letters of the alphabet, and considering that there are 8! different ways to arrange the 8 numbers for each of the three rings, the total number of possible ways to arrange 26 letters into 24 different symbols, according to Figure 1.1 would seem to be $26! \times 8!^3$. However, we need to adjust our calculations to account for the fact that there are only 24 buckets where to place the 26 letters. In addition, to compute the cryptographically relevant number of arrangements (i.e., "key space") we won't factor in the $8!^3$ since two keys that differ only by rearranging the degree numbers are cryptographically equivalent (i.e., they produce ciphertexts with identical structure, patterns, and repetitions).

There are $\binom{24}{2}$ ways to choose which buckets get two letters (cf. buckets in Figure 1.1 with letters j/y and q/z). Then, there are $\binom{26}{4}$ ways to choose the 4 letters that will share the two buckets, and the letters sharing the two buckets can be arranged in $\binom{4}{2}$ different ways. Finally, after removing the 4 letters that will be sharing buckets, the remaining 22 letters can be arranged in 22! ways. Thus, the key space is given by:

$$\binom{24}{2}\binom{26}{4}\binom{4}{2} \cdot 22! = 69 \cdot 26! \tag{1.1}$$

To put the key space of the Radar cipher into perspective, consider the *Permutation cipher*, a monoalphabetic substitution cipher in which each plaintext letter is mapped to a ciphertext letter via a random permutation of the alphabet. The key for such a cipher is therefore a permutation of the 26-letter alphabet, yielding a key space of size 26! (more precisely, $26! - 1$, since the identity permutation provides no encryption).

The key space of the Radar cipher given by Equation 1.1 is approximately 1.84 orders of magnitude larger than that of the Permutation cipher. This corresponds to an effective key size of 94 bits. Consequently, the Radar cipher's key space is larger than that of a 56-bit cipher (cf. Data Encryption Standard (DES)), though below the effective security level of a 112-bit cipher such as Triple DES (3DES). A caveat of monoalphabetic ciphers is that the statistical properties of the plaintext are carried over to the ciphertext, so they are vulnerable to statistical analysis. Furthermore, it is important to mention that a statistical analysis could exploit the fact that 3 letters share the same angle in 6 positions (e.g., h, p, x share angle 360), and 4 letters share the same angle in 2 positions (e.g., b, j, y, r share angle 90).

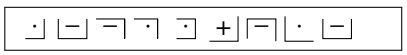
Finally, a motivated reader could devise extensions of the Radar cipher that make it polyalphabetic, that is, ciphers in which a given plaintext letter may be mapped to different ciphertext symbols at different points in the encryption process. One such extension aligns naturally with the rotational metaphor suggested by the concentric rings of the Radar cipher. Specifically, after encrypting a plaintext p, the ring containing p is rotated clockwise by p mod m positions. As a result, the same plaintext character will, in general, be mapped to a different ciphertext symbol upon its next occurrence (except in the case where $8|p$, which yields no rotation). A polyalphabetic cipher is more resistant to statistical analysis than a monoalphabetic one, but it is not immune to such attacks.

□

Now let us turn to another cipher, whose mechanism I am still not prepared to confess.

1.2 Pigpen-ish Cipher (or an undisclosed cipher)

Exercise 1.2 You intercept the following sequence of images from a suspicious channel. Not sure whether the message is in English or not. What's the message?



Solution (or not). I first encountered this puzzle at the age of 12, when two middle-school friends, Victor Hugo Romo Contreras (a.k.a. *"el compadre"*) and Adameck Collazo Gómez (a.k.a. *"el enano"*),[1] decided to bruise my ego with a short ciphertext. After several days, I conceded defeat, and they revealed the solution, only after securing my promise never to disclose it. I have honoured that promise ever since, eventually realizing that they themselves were not the authors of the cipher (a fact they never stated explicitly, though the enforced secrecy suggested as much). The puzzle is closely related to the *Pigpen cipher*.[2] To distinguish this cipher from the Pigpen cipher, we will call it

[1] In Spanish, *compadre* means the godfather of one's child, and *enano* means "little person." Neither nickname describes its bearer, but middle school was not a place where accuracy mattered.

[2] Wikipedia Contributors. *Pigpen Cipher — Wikipedia, The Free Encyclopedia*

the "Pigpen variation cipher." I am not about to break my promise here, especially since I have yet to find this particular variation described anywhere outside the mischievous pedagogy of *el compadre* and *el enano*.

The Pigpen variation cipher is probably just as old as the Pigpen cipher itself (a quick Google search suggests that its first academic description dates back to the middle ages). As the reader surely suspects, from a modern cryptanalytical perspective, the Pigpen cipher and the variation in question offer negligible security, as they maintain, like any monoalphabetic substitution cipher, a 1:1 relationship between the statistical properties of the plaintext and the ciphertext.

□

But enough of symbolic geometric ciphers and bruised egos. In the next chapter we will study steganography, a technique used to conceal a message in plain sight.

Chapter 2
Steganography

"Power is in tearing human minds to pieces and putting them together again in new shapes of your own choosing."

—George Orwell, 1984

2.1 Text Within Text

This section provides two examples of a typical technique for hiding secret messages within innocuous messages that evade casual or unguarded readers.

Definition 2.1 *Steganography* is the practice of concealing the existence of a message by embedding it within an innocuous-looking carrier so that an unintended observer is unaware that any communication is taking place at all.

Exercise 2.1 During a sting operation on a cold winter morning, a Toronto Police detective obtained the following message. An illegal asset is expected to be moved to a new location within the next 24 hours. As a steganography and code-breaking consultant, what clues can you identify?

July 12, Friday, '23
This city is as good as gold.
When the maple tree leaves
turn yellow, red, and brown,
the wind blows,
and I am at peace.

Solution. An imaginative mind could speculate about different hidden clues. However, at least to me, the most obvious one would suggest that the hidden message is *"gold leaves 5 AM"*. First, a quick check to the 2023 calendar would reveal that July 12^{th} didn't fall on a Friday, raising suspicion. Taking the purposely wrong date as the key to the puzzle, we interpret "July" as word number 7 in the text: "gold"; the next word, "leaves," happens to be word number 12. Friday is interpreted as number 5 (although it may as well be interpreted as number 6 if we start counting the week from Sunday, or, it could simply mean "Friday", but the phrase "within the next 24 hours" may be used as tie-breaker), and finally "AM" corresponds to word number 23.

□

Exercise 2.2 The detective from Exercise 2.1 has clearly had a busy week. By sheer coincidence, in an unrelated case, she recovered the following handwritten note. What message is hiding in it?

October 4 '21
This message is open for public release. The
Golden Gate Bridge will go through a series of
renovations as of midnight, November 1^{st}.

I will leave the reader to figure this one out.

Let's now get more serious. The next example illustrates how communication protocols can be exploited to hide messages in plain sight.

2.2 Text or Images in LSBs

The art of information hiding finds a fascinating manifestation in least significant bit (LSB) steganography, a technique involving the modification of least significant bits within digital images. This approach offers a straightforward yet insightful illustration of embedding secret data within seemingly innocuous carriers.

LSB steganography exploits the inherent limitations of human perception. Each pixel within a raster image, typically composed of red, green, and blue (RGB) colour channels, is represented by a fixed number of bits (commonly 8 bits per channel, resulting in 24-bit colour). The LSB, residing at the lowest order position within each channel's bit representation, contributes minimally to the overall perceived colour value. This allows for subtle alterations without introducing noticeable artifacts to the image.

Formally, let I represent the original image, and M represent the message to be hidden. Each pixel $p_{i,j}$ in I consists of colour components $r_{i,j}, g_{i,j}$, and $b_{i,j}$, each ranging from 0 to 255 (assuming 8-bit depth). The embedding process can be described as:

$$I' = E(I, M) \tag{2.1}$$

Where I' is the stego-image (the image containing the hidden message), and E is the embedding function. For each bit m_k of the message M, the LSB of a corresponding pixel channel component is modified:

$$c'_{i,j} = c_{i,j} - (c_{i,j} \bmod 2) + m_k \tag{2.2}$$

Where $c_{i,j}$ represents the pixel channel component (r, g, or b) and $c'_{i,j}$ is the modified value. Effectively, this operation ensures the LSB of $c'_{i,j}$ matches m_k. The human visual system, possessing a limited differential sensitivity, typically fails to detect these minute alterations, particularly in regions with high colour variation.

Crucially, the preservation of message integrity hinges on the utilization of lossless image compression formats. JPEG, for instance, employs a discrete cosine transform

(DCT) and quantization to achieve high compression ratios, resulting in irreversible data loss. This lossy compression inevitably corrupts the embedded LSB data, rendering the hidden message unrecoverable. Therefore, LSB steganography requires formats such as PNG or BMP, which employ lossless compression techniques, ensuring bit-perfect reconstruction of the image data during decompression.

The extraction process, denoted as $D(I') = M'$, involves reversing the embedding process. By iterating through each pixel channel component in the stego-image I' and extracting the LSB, the original message M' can be reconstructed:

$$m'_k = c'_{i,j} \bmod 2 \tag{2.3}$$

The recovered message M' will be identical to M assuming the absence of any lossy operations on the stego-image I'.

While LSB steganography provides a relatively simple and readily implementable approach to information hiding, it exhibits inherent limitations:

1. Capacity. The maximum message size is directly limited by the image dimensions and the number of bits modified per pixel.
2. Robustness: The embedded message is fragile and susceptible to even minor image manipulations (e.g., filtering, scaling).
3. Detectability: Statistical analysis techniques (e.g., chi-square analysis) can be employed to detect the presence of LSB steganography, particularly in images with low colour complexity.

Therefore, LSB steganography should be considered a basic introductory technique. More sophisticated steganographic methods employ adaptive embedding algorithms, transform domain techniques, and spread spectrum approaches to enhance capacity, robustness, and undetectability. However, this simple LSB approach offers a valuable pedagogical tool for understanding the fundamental principles of information hiding and the interplay between cryptography and digital media.

2.3 Network Steganography

Modern communication networks are not steganography-proof. Sophisticated steganography techniques exploit communication protocols to create virtual side-channels. For instance, in 802.11 networks, malicious nodes can establish covert channels by intentionally corrupting the Frame Check Sequence (FCS) field using a non-standard Cyclic Redundancy Check (CRC) polynomial. Because these frames appear to be naturally corrupted by environmental factors—such as signal fading, collisions, or interference, unsuspecting nodes simply drop them as noise. This allows illegitimate data to "hide amongst legitimate corrupted frames," making it extremely difficult to detect using traditional intrusion detection systems that rely on identifying abnormal usage patterns or behavioural norms.

The following example illustrates the concept using typical Wi-Fi household networks.

Example 2.1 This example is based on a paper I co-authored a decade ago, titled "Hamming Distance as a Metric for the Detection of CRC-Based Side-Channel Communications in 802.11 Wireless Networks."[1] This technique consists of hiding messages by exploiting the error detection mechanisms of wireless communication protocols, specifically the FCS field used in 802.11 networks (these are the common household Wi-Fi networks). This technique creates a virtual side-channel that allows malicious nodes to communicate while remaining invisible to standard monitoring tools.

Here is a description of how this protocol exploitation functions:

1. Intentional frame corruption. In standard wireless communication, every frame includes an FCS field containing a CRC value. This value is calculated using a predetermined standard polynomial to ensure the data was not altered during transmission (see Flow *A* in Figure 2.1). To hide a message, an attacker purposefully corrupts the frame by using a different CRC polynomial (such as the Koopman or Castagnoli polynomial[2] to calculate the FCS instead of the standard one (cf. Flow *B* in Figure 2.1).

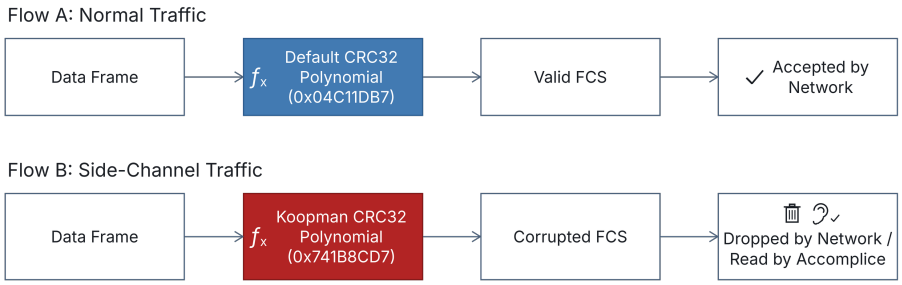


Fig. 2.1 Intentionally corrupted CRC to establish a steganographic channel.

2. Disguise as natural noise. Because the FCS value is generated with a non-standard polynomial, normal unsuspecting nodes that receive these frames will find the CRC check fails. These nodes interpret the frames as naturally corrupted, attributing the errors to signal fading, collisions, shadowing, or interference, and simply drop (discard) them. This allows the secret messages to "hide amongst legitimate corrupted frames," making them extremely difficult to distinguish from environmental noise.
3. Creating the side-channel. The covert communication is successful because the malicious nodes have agreed upon the alternative CRC polynomial in advance. While the rest of the network sees only noise and errors, the malicious receiver uses the same non-standard polynomial to interpret the "corrupted" frames and extract the hidden information.

[1] V. Chea, M. Vargas Martin, et al., "Hamming distance as a metric for the detection of CRC-based side-channel communications in 802.11 wireless networks," 2015 IEEE Conference on Communications and Network Security (CNS), Florence, Italy, 2015, pp. 218-226, doi: 10.1109/CNS.2015.7346831.

[2] Wikipedia Contributors. *Cyclic redundancy check — Wikipedia, The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Cyclic_redundancy_check`

4. Steganographic concealment. While encryption masks the content of a message, this protocol exploitation hides the very existence of the communication from untrained eyes. This is particularly effective in Wireless Ad Hoc Networks and Mobile Ad Hoc Networks (MANETs) because these infrastructures often lack central monitoring and are naturally prone to high frame error rates, providing a perfect "cloak" for the illegitimate data.

Detecting this type of hidden communication is difficult because:

1. No behavioural norms. Unlike other intrusions, side-channel frames do not follow abnormal usage patterns or profiles that can be easily flagged.
2. Variable noise. Because wireless noise is never constant (e.g., a nearby microwave can significantly change frame corruption rates), an attacker can easily blend their traffic into the unpredictable baseline of legitimate errors.
3. Lack of access. Modifying or inspecting frame error handling is often restricted by the firmware of communication chipsets, making it hard for software-based security tools to see the contents of "corrupted" frames.

Part II
Pre-Quantum Cryptography

Chapter 3
Number Theory[1]

"... I have a truly marvelous demonstration of this proposition which this margin is too narrow to contain."

—Pierre de Fermat, on the margin of Diophantus's Arithmetica

3.1 Integers

Definition 3.1 Given integers a and b with $b \neq 0$, we say that b is a *divisor* or a *factor* of a and that a is divisible by b if and only if $a = qb$ for some integer q. We write $b|a$ to signify that a is divisible by b and say "b divides a".

Theorem 3.1 *Let a, b, and c be integers, then*

1. *if $a|b$ and $a|c$, then $a|(b + c)$;*
2. *if $a|b$, then $a|bc$ for all integers c;*
3. *if $a|b$ and $b|c$, then $a|c$;*
4. *if $c|a$ and $c|b$, then $c|(xa + yb)$, for any integers x and y.*

Proof. We will prove 1 and 2.

1. if $a|b$ and $a|c$, then $a|(b + c)$. We will prove that x implies y (written $x \rightarrow y$). When proving an implication, all we have to prove is that when the hypothesis x is true, the conclusion y is also true (a quick look at the implication truth table would show that when the hypothesis is false, the implication is true regardless of the truth value of the conclusion). Thus, to prove 1, we assume that the hypothesis ($a|b$ and $a|c$) is true. So, if $a|b$ and $a|c$, we have that $b = ap$ and $c = aq$ for some integers p and q. We thus have that $b + c = ap + aq = a(p + q)$ and therefore $a|(b + c)$.
2. if $a|b$, then $a|bc$ for all integers c. If $a|b$ then we have that $b = ap$ for some integer p. Thus $bc = a(pc)$, and therefore $a|bc$.

□

A very important result is the *Division Algorithm* (a.k.a. *Division Theorem*). While the *Division Algorithm* may not look like an algorithm with sequenced steps and flow control statements, observing its elegant wording carefully would reveal its implied procedure.

[1] This chapter is based on *Discrete Mathematics and its Applications* by Kenneth H. Rosen, and *Cryptography and Network Security: Principles and Practice* by William Stallings.

Theorem 3.2 The Division Algorithm. *Let $a, b \in \mathbb{Z}, b \neq 0$. Then there exist unique integers q and r, with $0 \leq r < |b|$, such that $a = qb + r$.*

Definition 3.2 If a and b are natural numbers and $a = qb + r$ for nonnegative integers q and r with $0 \leq r < b$, the integer q is called the *quotient* and the integer r is called the *remainder* when a is divided by b.

For young readers, the following notation may not ring any bells, but it does show the elements of a division, where 33 is the *dividend*, 13 is the *divisor*, 2 is the *quotient*, and 7 is the *remainder*:

$$\begin{array}{r} 2 \\ 13\overline{)33} \\ \underline{26} \\ 7 \end{array}$$

And finally, here's an example that offers a clarification to the use of negative numbers, which often creates confusion as it is counterintuitive to traditional division.

Example 3.1 Dividing -13 by 7, according to the *Division Algorithm* yields a quotient of -2 and a remainder of 1, and not a quotient of -1 with a remainder of -6 as a traditional division would suggest. The reason for this is that the *Division Algorithm* imposes that the remainder must be nonnegative. In this book, we will perform integer division according to the *Division Algorithm*.

3.2 Modular Arithmetic

To denote the remainder of an integer division we use the *mod* notation. Thus, the remainder r of dividing a by b is denoted by $r = a \bmod b$.

Definition 3.3 If a and b are integers and m is a positive integer, then *a is congruent to b modulo m*, denoted $a \equiv b \pmod{m}$, if $m|(a-b)$. In other words, $a \equiv b \pmod{m}$ means that the remainders of dividing a by m and b by m are the same, i.e, $a \bmod m = b \bmod m$.

Example 3.2 $17 \not\equiv 3 \mod 6$ because $17 \bmod 6 = 5$, while $3 \bmod 6 = 3$.

Example 3.3 $17 \equiv 5 \mod 6$ because $17 \bmod 6 = 5 \bmod 6 = 5$.

Theorem 3.3 *Let a and b be integers, and let m be a positive integer. Then $a \equiv b \bmod m$ iff $a \bmod m = b \bmod m$.*

Proof. The proof is straightforward albeit we need to prove a double implication (**iff**, meaning "if and only if" and also written as $\iff$ or $\leftrightarrow$). First from left to right ($\rightarrow$): $a \equiv b \pmod{m}$ means that $m|(a-b)$, so $a - b = mp$ for some integer p, and thus $a = mp + b$. Now, consider the remainder r from $b \bmod m = r$, leading to $b = mq + r$ for some integer q. Thus, $a = mp + mq + r = m(p+q) + r$, so $a \bmod m = r$, and therefore $b \bmod m = a \bmod m = r$.

Proving the implication in the opposite direction (←): Let $r = a \bmod m = b \bmod m$. We have that $a = ms + r$, $b = mt + r$, and thus $r = b - mt$, for some integers s and t. Substituting r in $a = ms + r$ we have that $a = ms + b - mt = m(s - t) + b$, yielding that $a - b = m(s - t)$, so $m|(a - b)$, and thus $a \equiv b \pmod{m}$. □

Theorem 3.4 *Let m be a positive integer. The integers a and b are congruent modulo m if and only if there is an integer k such that $a = b + km$.*

Proof. We need to prove that $a \equiv b \pmod{m} \leftrightarrow a = b + km$:
(→): $m|(a - b)$ means that $a - b = mk$ for some integer k. Therefore $a = b + mk$.
(←): $a = b + km$ means $a - b = km$, and therefore $m|(a - b)$. □

Theorem 3.5 *Let m be a positive integer. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a + c \equiv b + d \pmod{m}$ and $ac \equiv bd \pmod{m}$.*

Proof. We will only prove the first conclusion ($a + c \equiv b + d \pmod{m}$). We will use the fact that if $x \equiv y \pmod{z}$ then $y = x + qz$.

The hypothesis states that $a = b + mq$ and $c = mp + d$, for some integers p and q. If we add a and c, we have that $a + c = b + mq + mp + d = m(p + q) + b + d$, and thus $a + c \equiv b + d \pmod{m}$. □

Theorem 3.6 *Let a, b, and $n > 0$ be integers, then*

1. $[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$;
2. $[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$;
3. $[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$.

3.3 Primes and Greatest Common Divisors

Definition 3.4 A natural number $p \geq 2$ is called *prime* iff the only natural numbers that divide p are 1 and p. A natural number $n > 1$ that is not prime is called *composite*.

From this definition, we arrive at the following corollary.

Corollary 3.1 *The integer n is* composite *iff there exists an integer a such that $a|n$ and $1 < a < n$.*

Theorem 3.7 *Every number $n > 1$ is divisible by a prime.*

Proof. We prove by contradiction. Essentially, in a proof by contradiction, we demonstrate that the negation of what we are trying to prove always leads to an absurd (an untrue statement).

The negation of "Every number $n > 1$ is divisible by a prime" is "There are some numbers that are not divisible by any prime." In general, the negation of a universal quantifier ("Every") is the existential quantifier ("There are some") of the same predicate but negated.

So we will put all numbers that are not divisible by any prime in a set S, and let m be the smallest element in S (cf. The Well-Ordering Principle). We know that m can't be prime because m is in the set of numbers that are not divisible by any prime, S, and since any prime is divisible by itself, we know that S contains no primes. Thus since m is not prime, then it is composite, and as such, there's an integer $k(1 < k < m)$ such that $k|m$. Since m is the smallest element in S, k is not in S, meaning that there must be a prime p such that $p|k$. But from 3 in Theorem 3.1, if $p|k$ and $k|m$ we must have that $p|m$, leading us to an absurd. □

Theorem 3.8 The Fundamental Theorem of Arithmetic. *Every positive integer greater than* 1 *can be written uniquely as a prime or as the product of two or more primes where the prime factors are written in order of non-decreasing size.*

Definition 3.5 The *prime factors* (a.k.a. *prime divisors*) of an integer $n \geq 2$ are the prime numbers that divide n.

Example 3.4 Here are some examples:

- $28 = 2 \times 2 \times 7 = 2^2 \times 7$, where 2 and 7 are the prime factors of 28 (all other prime numbers are exponentiated to the power 0).
- $30 = 2 \times 3 \times 5$, where 2, 3, and 5 are the prime factors of 30.
- $24 = 2 \times 2 \times 2 \times 3 = 2^3 \times 3$, where 2 and 3 are the prime factors of 24.

Finding the prime factorization of a number has a side product: testing for primality. One (inefficient) way to test the primality of an integer n is by checking whether $p|n$, for each prime $p < n$. For example, to see whether or not 103 is prime, we would check that none of $2, 3, 5, 7, 11, 13, 17, \ldots, 101$ divides 103.

However, we can use the following theorem to narrow down the number of primes p we need to check.

Theorem 3.9 *If a natural number $n > 1$ is not prime, then n is divisible by some prime $p \leq \sqrt{n}$.*

Proof. By the Fundamental Theorem of Arithmetic, every positive integer greater than 1 has a unique prime factorization such that $n = p_1 p_2 \cdots p_k$ (observe that $k \geq 2$ for any composite number). By contradiction, assume that every prime divisor of n satisfies $p_x > \sqrt{n}$. Then the product of any two prime factors p_i and p_j satisfies $p_i p_j > \sqrt{n}\sqrt{n} = n$. Finally, since $k \geq 2$, we must have that $n = p_1 p_2 \cdots p_k \geq p_i p_j > n$, which is absurd. □

Using Theorem 3.9, to test the primality of 103 it would suffice to test divisibility for odd prime numbers $p \leq \sqrt{103} < 11$, namely 3, 5, and 7.

A number theory beginner would surely intuit that there are infinitely many primes. To avoid relying on intuition, let's prove it. In doing so, we will use *The Well-Ordering Principle* which states that any nonempty set of natural numbers has a smallest element.

Theorem 3.10 *There are infinitely many primes.*

Proof. By contradiction. Assume there are n primes. Let's now consider the product of all n primes and add 1; we will call this number k, as follows: $k = p_1 p_2 \cdots p_n + 1$. We know that among these n primes, there's one p_i that satisfies $p_i | k \wedge p_i | (p_1 p_2 \cdots p_n)$ (the symbol $\wedge$ denotes the logical binary operator "and"). Using 4 from Theorem 3.1, and letting $k = ab$, and $(p_1 p_2 \cdots p_n) = yb$ in the equation $1 = k - (p_1 p_2 \cdots p_n)$, we must have that $p_i | 1$, which is absurd. □

How would we find all the prime numbers less than n? One way to find the prime numbers less than or equal to some given integer is using the *Sieve of Eratosthenes* (276-194 B.C.), which is illustrated with the next example.

Example 3.5 To find the prime numbers less than or equal to 50, we write the numbers from 2 to 50 and then we cross those which are multiples of 2 (except 2). Then we cross those which are multiple of the next non-crossed number, i.e., 3 (except 3), and continue in this fashion until the next non-crossed number is greater than $\sqrt{50}$. At the end of the process, the non-crossed numbers are the primes less than or equal to 50.

A few more Number Theory results that will be used later in the book.

Theorem 3.11 Fermat's Little Theorem.[2] *If p is prime and a is a positive integer not divisible by p then $a^{p-1} \equiv 1 \ (mod\ p)$.*

Corollary 3.2 *If p is prime and a is a positive integer then $a^p \equiv a \ (mod\ p)$.*

Definition 3.6 The number of positive integers less than n and relatively prime to n is called *Euler's totient function*, and is denoted as $\phi(n)$. By convention $\phi(1) = 1$.

Corollary 3.3 *For a prime p, $\phi(p) = p - 1$.*

Example 3.6 $\phi(6) = 2$ because only 1 and 5 are coprimes with 6. Now take a prime, say 17; $\phi(17) = 16$ because $\gcd(i, 17) = 1$ for $i = 1 \ldots 16$.

Theorem 3.12 Euler's Theorem.[3] *For every coprimes a and n, $a^{\phi(n)} \equiv 1 \ (mod\ n)$.*

Corollary 3.4 *For every coprimes a and n, $a^{\phi(n)+1} \equiv a \ (mod\ n)$.*

More on primality testing. One of the most widely deployed primality testing algorithms is Miller-Rabin, described in Algorithm 1. *Odd* enough, but up until two years ago, the `SymPy` implementation of Miller-Rabin didn't do the proper oddness check which caused some composite numbers to be deemed as primes under certain witnesses. I happened to notice by chance as I browsed their source code, and opened a ticket with a solution that was reviewed and implemented by the developing team.

---

[2] Pierre de Fermat (1601-1665) is regarded as the founder of modern number theory. Books I recommend about Fermat include Keith Devlin's *The Unfinished Game* and *Fermat's Enigma* by Simon Singh.

[3] In Morris Kline's *Mathematics and the Search for Knowledge*, Leonhard Euler (1707-1783) is depicted as the bridge between early modern discovery and the highly formalized mathematical structures of today. In *Math Makers*, Christian Spreitzer and Alfred S. Posamentier describe Euler as the most prolific mathematician in history, highlighting his vast output, generous character, and crucial role in formalizing Fermat's ideas, including providing the first published proof of Fermat's Little Theorem.

Algorithm 1 Miller–Rabin Primality Test

Require: Integer $n > 3$, odd; security parameter $k \geq 1$
Ensure: `true` if n is probably prime, `false` if n is composite
1: Write $n - 1 = 2^s \cdot d$ with d odd
2: **for** $i = 1$ to k **do**
3: Choose a random integer a with $2 \leq a \leq n - 2$
4: $x \leftarrow a^d \bmod n$
5: **if** $x = 1$ **or** $x = n - 1$ **then**
6: **continue**
7: **end if**
8: **composite** $\leftarrow$ `true`
9: **for** $r = 1$ to $s - 1$ **do**
10: $x \leftarrow x^2 \bmod n$
11: **if** $x = n - 1$ **then**
12: **composite** $\leftarrow$ `false`
13: **break**
14: **end if**
15: **end for**
16: **if composite then**
17: **return** `false`
18: **end if**
19: **end for**
20: **return** `true`

3.4 Greatest Common Divisors and Least Common Multiples

Definition 3.7 Let a and b be integers, not both zero. The largest integer g such that $g|a$ and $g|b$ is called the *greatest common divisor* of a and b, and is denoted by $\gcd(a, b)$.

Example 3.7 $\gcd(-18, 12) = 6$; $\gcd(33, 15) = 3$; $\gcd(18, 12) = 6$; $\gcd(18, -12) = 6$; $\gcd(-33, -15) = 3$.

Euclid[4] described an algorithm to compute the gcd of two integers a and b. This algorithm is well-known as the *Euclidean Algorithm*. The algorithm is based on the fact that $\gcd(a, b) = \gcd(b, r)$, where r is the remainder of dividing a by b. Its description for nonnegative integers a and b is in Algorithm 2.

Definition 3.8 The integers a and b are relatively prime if their greatest common divisor is 1.

Definition 3.9 The integers $a_1, a_2, \ldots, a_n$ are pairwise relatively prime if $\gcd(a_i, a_j) = 1$ whenever $1 \leq i < j \leq n$.

[4] Euclid (325-265 B.C.) is commonly regarded as the Father of Geometry. His *Elements*, a 13-book treatise on geometry and logic, stands as one of the most influential works in the history of mathematics. I confess I have not read it (and probably never will), but Morris Kline's *Mathematics and the Search for Knowledge* provides an engaging overview of Euclid's monumental impact and his influence up to the present day. Also, Christian Spreitzer and Alfred S. Posamentier's *Math Makers* presents Euclid as a foundational figure in mathematics, noting that concrete details about his life are largely based on conjecture. Posamentier places Euclid in the context of early geometry and highlights his role as one of the key "makers" whose work forms the basis of the field. And for lovers of geometry, I also recommend Alfred Posamentier and Ingmar Lehmann's *The Glorious Golden Ratio*.

Algorithm 2 Euclidean Algorithm

Require: Integers a, b with $0 < b \leq a$
Ensure: $\gcd(a, b)$
1: **while** $b \neq 0$ **do**
2: $r \leftarrow a \bmod b$
3: $a \leftarrow b$
4: $b \leftarrow r$
5: **end while**
6: **return** a

Example 3.8 8, 15, and 17 are pairwise relatively prime because $\gcd(8, 15) = \gcd(8, 17) = \gcd(15, 17) = 1$.

Definition 3.10 The *least common multiple* of the positive integers a and b is the smallest positive integer that is divisible by both a and b. The least common multiple of a and b is denoted by $\text{lcm}(a, b)$.

Theorem 3.13 *Let a and b be positive integers. Then* $|ab| = gcd(a, b) \cdot lcm(a, b)$.

Example 3.9 $\text{lcm}(27, 6) = \frac{27 \times 6}{\gcd(27,6)} = \frac{162}{3} = 54$.

3.5 Multiplicative Inverse

Definition 3.11 Let b and m be integers with $m > 1$. An integer x is called *multiplicative inverse of b modulo m* if $b \cdot x \equiv 1 \pmod{m}$.

Theorem 3.14 *Let b and m be integers with* $m > 1$. *The multiplicative inverse of b modulo m exists if and only if* $\gcd(b, m) = 1$.

Corollary 3.5 *Let b and m be integers with* $m > 1$. *If the multiplicative inverse of b modulo m exists, it is unique modulo m. That is, if x and y are both inverses of b modulo m, then* $x \equiv y \pmod{m}$.

To obtain the multiplicative inverse of $b \bmod m$ we use an extended version of the Euclidean Algorithm listed in Algorithm 3.

3.6 Pre-Quantum One-Way Trapdoor Functions

Definition 3.12 A *one-way trapdoor function* in cryptography is a mathematical operation that can be easily computed in one direction but is hard to reverse if a piece of information is missing.

This section describes two of the most widely used mathematical foundations in cryptography: prime factorization and discrete logarithms. While the former underpins

Algorithm 3 Extended Euclidean Algorithm to compute $b^{-1} \bmod m$

Require: Integers $m > 1$, $0 < b < m$
Ensure: $d = \gcd(m, b)$ and integers x, y such that $mx + by = d$; if $d = 1$, $y \bmod m$ is the modular inverse of b

$x_0 \leftarrow 1,\ y_0 \leftarrow 0$
$x_1 \leftarrow 0,\ y_1 \leftarrow 1$
$a \leftarrow m,\ \ r \leftarrow b$
while $r \neq 0$ **do**
    $q \leftarrow \lfloor a/r \rfloor$
    $(a, r) \leftarrow (r, a - q \cdot r)$
    $(x_0, x_1) \leftarrow (x_1, x_0 - q \cdot x_1)$
    $(y_0, y_1) \leftarrow (y_1, y_0 - q \cdot y_1)$
end while
$d \leftarrow a$
if $d = 1$ **then**
    return (d, x_0, y_0) ▷ $b \cdot y_0 \equiv 1 \pmod{m}$
else
    return $(d, x_0, y_0,$ "No inverse exists")
end if

the RSA algorithm,[5] the latter is the basis for Diffie-Hellman (DH) key exchange[6], and Elliptic Curve Cryptography (ECC).[7] Although these algorithms are vulnerable to quantum computing, they remain central to modern security. Chapter 7 explores RSA; however, DH and ECC fall outside the scope of this book.

3.6.1 Prime Factorization

Definition 3.13 Given two large prime numbers, p and q, it is easy to find their product $n = pq$. But given n alone, it is hard to find its prime factors p and q. This is known as the *prime factorization problem*.

3.6.2 Discrete Logarithms

Definition 3.14 In a finite group[8] formed via modulo prime p, it is easy to calculate $a = b^e \bmod p$, but it is hard to find e when only a, b, and p are known. This is known as the *discrete logarithm problem*.

[5] Wikipedia Contributors. *RSA cryptosystem — Wikipedia, The Free Encyclopedia*

[6] Wikipedia Contributors. *Diffie-Hellman key exchange — Wikipedia, The Free Encyclopedia*

[7] Wikipedia Contributors. *Elliptic-Curve cryptography — Wikipedia, The Free Encyclopedia*

[8] A finite group satisfies closure, associativity, identity, and inverse axioms, and the total count of elements is finite.

3.7 Putting It All Together using SymPy

The following program illustrates different number theory functions available in the SymPy manual. Some functions were not covered in this book but are important in the area of cryptography such as the Chinese Remainder Theorem. The reader is reminded that the complete source code accompanying this book is available at: https://github.com/miguevmartin/Practical-Cryptography.

```
"""
TODO-based exploration of number theory with SymPy.
Author: Miguel Vargas Martin using SymPy manual and ChatGPT
Last modified: January 2026

Instructions:
- Read each TODO carefully.
- Before running a line, try to predict its output.
- Uncomment lines only when instructed.
- Answer the questions in the comments or in a separate worksheet.
"""

import warnings
warnings.filterwarnings('ignore')

from sympy import *

# TODO 1: Basic integers

a, b, n = 3, 18, 25

# TODO 1.1:
# Predict gcd(a, b). Then uncomment and verify.
# print(igcd(a, b))

# TODO 1.2:
# Predict lcm(a, b). What is the relationship between gcd and lcm?
# print(ilcm(a, b))

# TODO 1.3:
# List all positive divisors of n.
# How does this relate to the prime factorization of n?
# print(divisors(n))

# TODO 1.4:
# How many divisors does n have?
# print(divisor_count(n))

# TODO 1.5:
# Find the prime factorization of n.
# print(factorint(n))


# TODO 2: Primality
```

```
# TODO 2.1:
# Is n prime? Why or why not?
# print(isprime(n))

# TODO 2.2:
# Pick a different value for n that *is* prime and test it.


# TODO 3: Euler's Totient Function

from sympy.ntheory import totient

# TODO 3.1:
# Compute   (a),   (b), and   (n).
# What does   (n) count?
# print(totient(a))
# print(totient(b))
# print(totient(n))

# TODO 3.2:
# Verify   (p) = p     1 for a prime p of your choice.


# TODO 4: Chinese Remainder Theorem (CRT)

from sympy.ntheory.modular import crt

# Solve the system:
#   x     49 (mod 99)
#   x     76 (mod 97)
#   x     65 (mod 95)

# TODO 4.1:
# Before running this, check whether the moduli are pairwise
    coprime.
solution = crt([99, 97, 95], [49, 76, 65])
print(solution)

# TODO 4.2:
# Extract the solution x and verify each congruence manually.


# TODO 5:  M i l l e r Rabin  Primality Test

from sympy.ntheory.primetest import mr

# TODO 5.1:
# The number below is large. Do you expect it to be prime?
print(mr(1373651, [2, 3, 5]))

# TODO 5.2:
# Change the bases. Does the result change?
# Why is  M i l l e r Rabin  called a "probabilistic" test?
```

```
# TODO 6: Primitive Roots

# TODO 6.1:
# Is a a primitive root modulo n?
print(is_primitive_root(a, n))

# TODO 6.2:
# Find a primitive root modulo n (if one exists).
g = primitive_root(n)
print(g)

# TODO 6.3:
# Verify that g generates all units modulo n.


# TODO 7: Discrete Logarithm

from sympy.ntheory import discrete_log

# Solve: find x such that 7^x     15 (mod 41)
x = discrete_log(41, 15, 7)
print(x)

# TODO 7.1:
# Verify the result by computing 7**x % 41.
# TODO 7.2:
# Why is the discrete logarithm problem considered hard in general?
```

Listing 3.1 Some warm up examples to reinforce familiarity with basic number theory.

Listing 3.1 uses the statement `warnings.filterwarnings('ignore')` only to suppress warnings that might distract from the core goal of the exercise. This is not meant as a best practice, as warnings often signal serious issues, including potential security vulnerabilities. Nevertheless, for introductory cryptography exercises in a controlled environment, ignoring warnings can help students focus on the relevant number-theoretic concepts.

Here's a Java-equivalent program generated by ChatGPT from the program in Listing 3.1:

```
/*
* TODO-based exploration of number theory with SymPy.
* Author: ChatGPT with small adaptations by Miguel Vargas Martin
* Last modified: January 2026
*
* Instructions:
* - Read each TODO carefully.
* - Before running a line, try to predict its output.
* - Uncomment lines only when instructed.
* - Answer the questions in the comments or in a separate worksheet
    .
*/

import java.math.BigInteger;
```

```
import java.util.*;

import org.apache.commons.math3.util.ArithmeticUtils;
import org.apache.commons.math3.primes.Primes;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

import java.security.Security;

public class NumberTheoryExploration {

    public static void main(String[] args) {

        Security.addProvider(new BouncyCastleProvider());

        // TODO 1: Basic integers

        int a = 3, b = 18, n = 25;

        // TODO 1.1: GCD
        System.out.println("gcd(a,b) = " + ArithmeticUtils.gcd(a, b
    ));

        // TODO 1.2: LCM
        System.out.println("lcm(a,b) = " + ArithmeticUtils.lcm(a, b
    ));

        // TODO 1.3: Divisors of n
        System.out.println("Divisors of n:");
        for (int d = 1; d <= n; d++) {
            if (n % d == 0) {
                System.out.print(d + " ");
            }
        }
        System.out.println();

        // TODO 1.4: Number of divisors
        int divisorCount = 0;
        for (int d = 1; d <= n; d++) {
            if (n % d == 0) divisorCount++;
        }
        System.out.println("Number of divisors of n = " +
    divisorCount);

        // TODO 1.5: Prime factorization
        System.out.println("Prime factorization of n:");
        factorize(n);

        // TODO 2: Primality
        System.out.println("Is n prime? " + Primes.isPrime(n));

        // TODO 3: Euler's Totient Function
        System.out.println("phi(a) = " + totient(a));
        System.out.println("phi(b) = " + totient(b));
        System.out.println("phi(n) = " + totient(n));
```

```
        // TODO 4: Chinese Remainder Theorem (CRT)
        BigInteger[] moduli = {
            BigInteger.valueOf(99),
            BigInteger.valueOf(97),
            BigInteger.valueOf(95)
        };

        BigInteger[] residues = {
            BigInteger.valueOf(49),
            BigInteger.valueOf(76),
            BigInteger.valueOf(65)
        };

        BigInteger crtSolution = chineseRemainder(residues, moduli)
    ;
        System.out.println("CRT solution x = " + crtSolution);

        // TODO 5: M i l l e r Rabin Primality Test
        BigInteger test = BigInteger.valueOf(1_373_651);
        System.out.println(" M i l l e r Rabin result: " +
                test.isProbablePrime(10));

        // TODO 6: Primitive Roots (limited implementation)
        System.out.println("Primitive root modulo 25:");
        System.out.println(findPrimitiveRoot(25));

        // TODO 7: Discrete Logarithm (brute-force demo)
        System.out.println("Solving 7^x     15 (mod 41)");
        System.out.println("x = " + discreteLog(7, 15, 41));
    }

    // ------------------ Helpers ------------------
    static void factorize(int n) {
        int x = n;
        for (int p = 2; p * p <= x; p++) {
            while (x % p == 0) {
                System.out.print(p + " ");
                x /= p;
            }
        }
        if (x > 1) System.out.print(x);
        System.out.println();
    }

    static int totient(int n) {
        int result = n;
        for (int p = 2; p * p <= n; p++) {
            if (n % p == 0) {
                while (n % p == 0) n /= p;
                result -= result / p;
            }
        }
        if (n > 1) result -= result / n;
        return result;
    }
```

```
    static BigInteger chineseRemainder(BigInteger[] a, BigInteger[]
     m) {
        BigInteger M = BigInteger.ONE;
        for (BigInteger mi : m) M = M.multiply(mi);

        BigInteger result = BigInteger.ZERO;
        for (int i = 0; i < m.length; i++) {
            BigInteger Mi = M.divide(m[i]);
            BigInteger inv = Mi.modInverse(m[i]);
            result = result.add(a[i].multiply(Mi).multiply(inv));
        }
        return result.mod(M);
    }

    static int findPrimitiveRoot(int n) {
        if (totient(n) <= 2) return -1;
        int phi = totient(n);

        for (int g = 2; g < n; g++) {
            Set<Integer> seen = new HashSet<>();
            int x = 1;
            for (int i = 0; i < phi; i++) {
                x = (x * g) % n;
                seen.add(x);
            }
            if (seen.size() == phi) return g;
        }
        return -1;
    }

    static int discreteLog(int base, int target, int mod) {
        int value = 1;
        for (int x = 0; x < mod; x++) {
            if (value == target) return x;
            value = (value * base) % mod;
        }
        return -1;
    }
}
```

Listing 3.2 Some warm up examples to reinforce familiarity with basic number theory.

Chapter 4
Symmetric Key Cryptography

"Ne pleure pas, Alfred! J'ai besoin de tout mon courage pour mourir à vingt ans!"

—Évariste Galois' last words

The exercises in this chapter are designed to help students understand the main symmetric ciphers modes of operation. In particular, those standardized in NIST Special Publication 800-38A: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). I have also included Galois/Counter[1] Mode (GCM), standardized in NIST Special Publication 800-38D, and XTS-AES, standardized in NIST Special Publication 800-38E (where XTS stands for *eXclusive-or Encrypt eXclusive-or Tweakable block cipher with ciphertext Stealing*, and AES stands for *Advanced Encryption Standard*, which is standardized in FIPS 197). The chapter also includes exercises on key wrapping, standardized as an additional mode of operation in NIST Special Publication 800-38F.

4.1 Avalanche Effect

Definition 4.1 The *avalanche effect* refers to the extent to which a small change in the input (such as flipping a single bit) causes widespread changes in the output of a block cipher or cryptographic hash function. It is typically measured by the proportion of output bits that flip as the algorithm progresses through its internal rounds. A strong cipher or hash function exhibits an avalanche effect close to 50% after enough rounds, and at the output.

Listing 4.1 uses Triple DES[2] (3DES) from cryptography.io to illustrate the avalanche effect in two ways. First, it encrypts the same plaintext (`b"a secret message"`) using two keys (`k1, k2`) that differ by one bit. Then, it encrypts two plaintexts that differ by one bit (`b"a secret"` and `b"a secreu"`) using the same key (`k1`). The program also offers

[1] In *Math Makers*, Christian Spreitzer and Alfred S. Posamentier recount the tragic death of Évariste Galois (1811-1832) while highlighting his foundational contributions to group theory and algebra. Eduardo Sáenz de Cabezón offers a similarly moving perspective in his YouTube video *El increíble caso de Évariste Galois*, which provides a touching narrative of the mathematician's final days.

[2] NIST withdrew 3DES as of January 1, 2024. For legacy, the withdrawn specification is available at NIST Special Publication 800-67 Revision 2.

a few challenges for the curious reader.

```
"""
Avalanche Effect Exploration using Triple DES (3DES)

This program illustrates how small changes in the key or plaintext
may (or may not!) affect the ciphertext, depending on the cipher
and mode of operation.

Last modified: February 2026
Author: Miguel Vargas Martin
"""

import warnings
warnings.filterwarnings('ignore')

from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
from cryptography.hazmat.backends import default_backend


def hamming(a, b):
    """
    Compute the Hamming distance between two byte strings.

    The Hamming distance counts the number of bit positions
    in which two equal-length byte strings differ.

    Parameters
    ----------
    a, b : bytes
        Byte strings of equal length.

    Returns
    -------
    int
        Number of differing bits.
    """
    return bin(
        int.from_bytes(a, 'big') ^ int.from_bytes(b, 'big')
    ).count('1')


# Key setup
backend = default_backend()
# Two closely related 3DES keys (24 bytes each)
k1 = b'12345678' * 3
k2 = b'123456781234567812345679'

# Guiding questions:
# 1) If hamming(k1, k2) = 1, why is hamming(ct1, ct2) = 0?
#    (Hint: think about how 3DES keys are internally structured.)
# 2) Now set:
#       k2 = b'123456781234567812345677'
```

```
#    and observe that hamming(k1, k2) = 4. Why?
# 3) Exercise:
#    Find k1 and k2 such that:
#       hamming(k1, k2) = 1
#       hamming(ct1, ct2) > 0
#    Example attempt:
#    k1 = b'12345679' * 3
#    k2 = b'12345679123456791234567 9'

# Avalanche effect with respect to the key
cipher1 = Cipher(algorithms.TripleDES(k1), modes.ECB(), backend=
    backend)
cipher2 = Cipher(algorithms.TripleDES(k2), modes.ECB(), backend=
    backend)

encryptor1 = cipher1.encryptor()
encryptor2 = cipher2.encryptor()

# Same plaintext encrypted under two slightly different keys
ct1 = encryptor1.update(b"a secret message") + encryptor1.finalize
    ()
ct2 = encryptor2.update(b"a secret message") + encryptor2.finalize
    ()

print(
    'Avalanche effect (same plaintext, different keys):',
    hamming(ct1, ct2)
)

# Avalanche effect with respect to the plaintext
encryptor3 = cipher1.encryptor()
encryptor4 = cipher1.encryptor()

# Two plaintexts differing by a single bit:
#   hamming(b'a secret', b'a secreu') = 1
ct3 = encryptor3.update(b"a secret") + encryptor3.finalize()
ct4 = encryptor4.update(b"a secreu") + encryptor4.finalize()

# Question:
# What happens if we instead use b"a secres"?
# Does the bit position of the change matter?

print(
    'Avalanche effect (different plaintexts, same key):',
    hamming(ct3, ct4)
)
```

Listing 4.1 Exploring the avalanche effect in Triple DES: comparing ciphertext differences resulting from minimal changes in keys and plaintexts.

I prompted ChatGPT to generate equivalent code to Listing 4.1 using Java and the Bouncy Castle[3] cryptographic library. The result is in Listing 4.2.

[3] Bouncy Castle – Open-source cryptographic APIs: https://www.bouncycastle.org/

```
/*
* Avalanche Effect Exploration using Triple DES (3DES)
*
* This program illustrates how small changes in the key or
    plaintext
* may (or may not!) affect the ciphertext, depending on the cipher
* and mode of operation.
*
* Last modified: February 2026
* Author: ChatGPT with some modifications by Miguel Vargas Martin
*/

import org.bouncycastle.crypto.engines.DESedeEngine;
import org.bouncycastle.crypto.params.KeyParameter;
import org.bouncycastle.crypto.BufferedBlockCipher;
import org.bouncycastle.crypto.modes.ECBBlockCipher;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

import java.security.Security;
import java.util.Arrays;

public class AvalancheEffect3DES {

    static {
        Security.addProvider(new BouncyCastleProvider());
    }

    /**
     * Compute the Hamming distance (bitwise) between two byte
    arrays.
     * Both arrays must have the same length.
     */
    public static int hamming(byte[] a, byte[] b) {
        if (a.length != b.length) {
            throw new IllegalArgumentException("Inputs must have
    equal length");
        }

        int distance = 0;
        for (int i = 0; i < a.length; i++) {
            distance += Integer.bitCount(a[i] ^ b[i]);
        }
        return distance;
    }

    /**
     * Encrypt a single block-aligned plaintext using 3DES in ECB
    mode.
     */
    public static byte[] encrypt(byte[] key, byte[] plaintext)
    throws Exception {
        BufferedBlockCipher cipher =
                new BufferedBlockCipher(new ECBBlockCipher(new
    DESedeEngine()));
```

```
        cipher.init(true, new KeyParameter(key));

        byte[] output = new byte[cipher.getOutputSize(plaintext.
    length)];
        int len = cipher.processBytes(plaintext, 0, plaintext.
    length, output, 0);
        cipher.doFinal(output, len);

        return output;
    }

    public static void main(String[] args) throws Exception {

        // Key setup
        byte[] k1 = "12345678".repeat(3).getBytes();
        byte[] k2 = "123456781234567812345679".getBytes();

        // Same plaintext, different (closely related) keys
        byte[] plaintext = "a secret message".getBytes();

        byte[] ct1 = encrypt(k1, plaintext);
        byte[] ct2 = encrypt(k2, plaintext);

        System.out.println(
            "Avalanche effect (same plaintext, different keys): "
            + hamming(ct1, ct2)
        );

        // Plaintext avalanche effect
        byte[] p1 = "a secret".getBytes();
        byte[] p2 = "a secreu".getBytes(); // differs by one bit

        byte[] ct3 = encrypt(k1, p1);
        byte[] ct4 = encrypt(k1, p2);

        System.out.println(
            "Avalanche effect (different plaintexts, same key): "
            + hamming(ct3, ct4)
        );

        // Sanity check for students
        System.out.println(
            "Hamming distance of plaintexts: "
            + hamming(p1, p2)
        );
    }
}
```

Listing 4.2 Exploring the avalanche effect in Triple DES: comparing ciphertext differences resulting from minimal changes in keys and plaintexts.

4.2 Advanced Encryption Standard

Listing 4.3 illustrates the avalanche effect of encrypting the same plaintext with two keys that differ by one bit, as well as the avalanche effect of encrypting two plaintext that differ by one bit with the same key. Just for kicks, the encryption time is measured. Again, a few challenges are offered for the interested reader.

```
"""
Avalanche Effect Exploration using AES

This exercise mirrors the earlier 3DES experiment, but now uses AES
    .
Students are encouraged to compare the avalanche behavior of modern
block ciphers under similar conditions.

Last modified: February 2026
Author: Miguel Vargas Martin
"""

import warnings
warnings.filterwarnings('ignore')

import time
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
from cryptography.hazmat.backends import default_backend


def hamming(a, b):
    """
    Compute the Hamming distance (bitwise) between two byte strings
    .

    Parameters
    ----------
    a, b : bytes
        Byte strings of equal length.

    Returns
    -------
    int
        Number of differing bits.
    """
    return bin(
        int.from_bytes(a, 'big') ^ int.from_bytes(b, 'big')
    ).count('1')

# AES setup
backend = default_backend()

# AES keys (128-bit)
# Exercise (b): try 192-bit and 256-bit keys
k1 = b'12345678' * 2                  # 16 bytes
```

```
k2 = b'1234567812345679'              # differs by one bit

# Initialization Vector (CBC mode requires a random IV in practice)
iv = b'fedcba9876543210'               # 16 bytes (AES block size)

# Plaintext must be a multiple of 16 bytes when using CBC with no
    padding
plaintext = b"a secret message"        # exactly 16 bytes

# Guiding questions:
# 1) If hamming(k1, k2) is small, do we always get a large
#    hamming(ct1, ct2)? Why or why not?
# 2) What changes if the IV is modified?
# 3) How does AES compare to 3DES in terms of avalanche behaviour?

# Avalanche effect: key difference
cipher1 = Cipher(algorithms.AES(k1), modes.CBC(iv), backend=backend
    )
cipher2 = Cipher(algorithms.AES(k2), modes.CBC(iv), backend=backend
    )

encryptor1 = cipher1.encryptor()
encryptor2 = cipher2.encryptor()

start = time.time()
ct1 = encryptor1.update(plaintext) + encryptor1.finalize()
end = time.time()

ct2 = encryptor2.update(plaintext) + encryptor2.finalize()

print(
    'Avalanche effect (same plaintext, different keys):',
    hamming(ct1, ct2)
)
print('Encryption time (single block): {:.6f}s'.format(end - start)
    )

# Avalanche effect: plaintext difference
encryptor3 = cipher1.encryptor()
encryptor4 = cipher1.encryptor()

# These two plaintexts differ by a single bit
p1 = b'a secret message'
p2 = b'a secret messagd'

ct3 = encryptor3.update(p1) + encryptor3.finalize()
ct4 = encryptor4.update(p2) + encryptor4.finalize()

# Guiding question:
# Compare hamming(p1, p2) with hamming(ct3, ct4).
# Does AES exhibit a stronger avalanche effect than 3DES?

print(
    'Avalanche effect (different plaintexts, same key):',
    hamming(ct3, ct4)
```

```
)
```

Listing 4.3 Avalanche effect of Advanced Encryption Standard.

Exercise 4.1 Use 3DES in CFB mode to encrypt plaintext `p = b'But the soup is getting cold'` utilizing the following combination of keys `k` and initialization vector `iv`:

```
k1 = b'4k15coLr'
iv = b'hicrypto'
c1 = TripleDES_CFB(k1, iv, p) #this is ciphertext 1

k2 = b'5j04boMsyoswyi3pyoswyi3p'
iv = b'hicrypto'
c2 = TripleDES_CFB(k2, iv, p) #this is ciphertext 2

k3 = b'c5hzpd4lc5hzpd4l4k15cnMs'
iv = b'hicrypto'
c3 = TripleDES_CFB(k3, iv, p) #this is ciphertext 3
```

What are the values `c1`, `c2`, and `c3`? Do you see anything peculiar about these values?

Solution. The peculiarity is that `c1 = c2 = c3 = b'\x85\x85G\xcc\x8f\xbd\x15\xacwB\x80\xd8p\x1fZ\x90\xb3\xbd&N_X\xc3\xc6=aT\x13\x93\xef\xcb\x99'`. The explanation is as follows:

a) First, we note that `k1` is a 64-bit (8-byte) key, whereas `k2` and `k3` are 192-bit keys composed of three 64-bit subkeys. Let's call the subkeys of key `ki` using the following notation: `ki = ki,3 + ki,2 + ki,1`, where each `ki,j` is a 64-bit subkey and the symbol + means concatenation.
b) Second, we note that the 3DES implementation in cryptography.io provides backward compatibility with DES by making `k2=k3=k1`, when a 64-bit key is given, such as `k1 = 4k15coLr`. So, 3DES performs the following operations: `C = E(k3, D(k2, E(k1, p)))`.
   Thus, `c1` is a 3DES encryption with `k1 = k2 = k3 = b'4k15coLr'`.
c) Then, looking closely at `k2` and `k3`, we note that some 64-bit subkeys are identical, namely:
   ```
   k2 = k2,3 + k2,2 + k2,1 = 5j04boMs + yoswyi3p + yoswyi3p
   k3 = k3,3 + k3,2 + k3,1 = c5hzpd4l + c5hzpd4l + 4k15cnMs
   ```
   And thus, we can represent the computation of `c2` and `c3` as follows:
   ```
   c2 = E(k2,3, D(k2,2, E(k2,1, p)))
   c3 = E(k3,3, D(k3,2, E(k3,1, p)))
   ```
   Now, since `k2,2 = k2,1`, we have that `c2 = E(k2,3, p)`, and since `k3,3 = k3,2`, we have that `c3 = E(k3,1, p)`.

d) Analyzing the binary representation of `k1`, `k2,3`, and `k3,1`, we observe that `k2,3` and `k3,1` are only different from `k1` by the least significant bit of some bytes. See Table 4.1.

Table 4.1 Binary representations of `k1`, `k2,3` and `k3,1`.

Keys	Binary (big-endian)
`4k15coLr`	00110100 01101011 00110001 00110101 01100011 01101111 01001100 01110010
`5j04boMs`	0011010**1** 0110101**0** 0011000**0** 0011010**0** 0110001**0** 01101111 0100110**1** 0111001**1**
`4k15cnMs`	00110100 01101011 00110001 00110101 01100011 0110111**0** 0100110**1** 0111001**1**

We know that DES ignores the least significant bit of each byte in the key, resulting in a 56-bit key. Thus, encrypting with either of these three keys `k1`, `k2,3`, and `k3,1` would yield the same ciphertext.

e) Finally, since encrypting with `k2,3 = 5j04boMs` is equivalent to encrypting with `k3,1 = 4k15cnMs` or encrypting with `k1 = 4k15coLr` for the reasons stated in Part d) above, we have that `c1 = c2 = c3`.

Listing 4.4 offers a hint to the full solution code.

```
keys = [b'4k15coLr',b'5j04boMsyoswyi3pyoswyi3p',b'
    c5hzpd4lc5hzpd4l4k15cnMs']
iv = b'hicrypto'
mode = modes.CFB(iv)
message = b'But the soup is getting cold'
for k in keys:
    padder = padding.PKCS7(64).padder()
    cipher1 = Cipher(algorithms.TripleDES(k), mode, backend=backend
    )
    encryptor1 = cipher1.encryptor()
    plaintext = padder.update(message)
    plaintext += padder.finalize()
    ct1 = encryptor1.update(plaintext)
    encryptor1.finalize()
    print('k='+str(k)+':')
        print(ct1) # b'\x85...\x90\xb3\xbd&N_X\xc3\xc6=aT\x13\x93\
    xef\xcb\x99'
```

Listing 4.4 Solution snippet.

□

Exercise 4.2 Encrypt `p1` and `p2` below using AES in mode ECB, CFB, OFB, and CTR, and using always the same value of the key `k` provided below. Do not use padding. Then, compute and compare Hamming distances of the ciphertexts obtained in each mode, i.e., `hamming(AES_ECB(k, p1), AES_ECB(k, p2))` vs `hamming(AES_CFB(k, p1), AES_CFB(k, p2))` vs `hamming(AES_OFB(k, p1), AES_OFB(k, p2))` vs `hamming(AES_CTR(k, p1), AES_CTR(k, p2))`.

```
p1 = b'a secret message'
p2 = b'a secret lessage'
k = b'abcdefghijklmnop'
iv = ctr = b'0123456789abcdef' #if an iv/ctr value is needed
```

Solution. Since the messages are 128 bits long and we are not using padding, we assume a one-block encryption. In ECB mode the plaintext and the key are the inputs to the encryption algorithm, so a Hamming distance of around 64 is expected. In contrast, CFB, OFB, and CTR, use the `iv/ctr` and the key as inputs to the encryption algorithm, but not the plaintext. Now we note that `hamming(p1,p2) = 1`. Therefore, the fact that the two plaintexts differ by one bit results in a Hamming distance of 1 in the CFB, OFB, and CTR ciphertexts. The code snippet in Listing 4.5 shows a sketch of the solution.

```
k = b'abcdefghijklmnop'
iv = b'0123456789abcdef'
ms = [modes.ECB(), modes.CFB(iv), modes.OFB(iv), modes.CTR(iv)]
p1 = b'a secret message'
p2 = b'a secret lessage'
for m in ms:
    cipher1 = Cipher(algorithms.AES(k), m, backend=backend)
    encryptor1 = cipher1.encryptor()
    c1 = encryptor1.update(p1)
    encryptor1.finalize()
    encryptor2 = cipher1.encryptor()
    c2 = encryptor2.update(p2)
    encryptor2.finalize()
    print(hamming(c1,c2)) #output is: 69 1 1 1
```

Listing 4.5 Solution snippet.

□

Exercise 4.3 This exercise aims at illustrating why CFB, OFB, and CTR modes of operation should use a different `iv` and `key` values every time. Suppose an attacker knows the following:

- The first 2 bytes (16 bits) of a plaintext `p1` are `01101000 01101001`.
- The first 2 bytes (16 bits) of the ciphertext obtained from encrypting `p1` and another plaintext, `p2`, with the same `iv/nonce/ctr` and the same key `k` are:

  ```
  c1 = E(k, p1) = 00110001 10010111
  c2 = E(k, p2) = 00110111 10010001
  ```

  Show how the attacker would find the first 2 bytes (16 bits) of `p2`.

Solution. The attacker will easily figure out `p2` as follows. We will use the notation `F2B(x)` to denote "the first 2 bytes of `x`", and I have added some extra brackets for clarity:

```
F2B(c1) XOR F2B(c2) = (F2B(p1) XOR E(k, iv/nonce/ctr)) XOR (F2B(p2) XOR E(k, iv
/nonce/ctr)) = F2B(p1) XOR F2B(p2).
```

Then, knowing the `F2B(p1)`, the attacker can obtain `p2` as follows:

`F2B(c1) XOR F2B(c2) XOR F2B(p1) = F2B(p2) = 01101110 01101111.`

□

Exercise 4.4 This exercise is equivalent to Exercise 4.3. Johnny used AES-128 in CTR mode to encrypt confidential information. After transmitting encrypted documents over an insecure network, Johnny realizes that he used the same counter value with the same key to encrypt two different plaintexts. In other words, Johnny performed the following encryptions:

a) `C1 = P1 XOR E(K, Counter1)`, and
b) `C2 = P2 XOR E(K, Counter1)`.

To make things worse for Johnny, `P1` starts with the phrase *"Sovereign Artic"* [sic], and it's very likely an attacker would make this assumption.

Why is Johnny so concerned? How would an attacker proceed to obtain `P2`? Assume `P1` and `P2` are each 16 bytes long.

Solution. Johnny knows an attacker will easily figure out `P2` as follows:

`C1 XOR C2 = (P1 XOR E(K, Counter1)) XOR (P2 XOR E(K, Counter1)) = P1 XOR P2.`

Then, assuming `P1` contains *"Sovereign Artic"*, the attacker can obtain `P2` as follows:

`C1 XOR C2 XOR "Sovereign Artic"`, which will be *almost* equal to `P2`. □

Exercise 4.5 Use cryptography.io to compare the Hamming distance between AES ciphertexts obtained when encrypting plaintexts `p1` and `p2` with `k` (i.e., `hamming(E(k, p1), E(k, p2))` using modes of operation ECB, CFB, OFB, and CTR.

Use the following values for `p1, p2`, and `k`:

```
p1 = b'a secret message'
p2 = b'a secret messagd'
k = b'12345678'*2
```
If/when in need of an initial value use `iv = ctr = b'fedcba9876543210'`
Do not use padding; you don't need it anyways.

1. What was the Hamming distance in ECB mode? (Answer: 80)
2. What was the Hamming distance in CFB mode? (Answer: 1)
3. What was the Hamming distance in OFB mode? (Answer: 1)
4. What was the Hamming distance in CTR mode? (Answer: 1)

Solution. In ECB mode, the plaintext and the key are the inputs to the encryption algorithm, so a Hamming distance of around 64 is expected. In contrast, CFB, OFB, and CTR, use the `iv` and the key `k` as inputs to the encryption algorithm, but not the plaintext.

Therefore, the fact that the two plaintexts differ by one bit results in a Hamming distance of 1 in the ciphertexts. □

Exercise 4.6 While this exercise doesn't directly concern AES, it sheds some light on the futility to enhance DES with two keys (2DES), which eventually led to AES's "predecessor," 3DES. This example illustrates the meet-in-the-middle attack on 2DES which encrypts a plaintext `P` using the following operations:

```
C=E(K2, E(K1, P)).
```

You have a plaintext-ciphertext pair `(P, C)` obtained using 2DES in the following way:

`X = E(K1, P) and C = E(K2, X)`, where:
`P = b'order retreat am'`
`C = b'w\x0b\x92jz\x9a\xbd;\x97\x89B\xde\xd4\x85\xc3m'`

Assume that the mode of operation used was ECB (no padding), that `K1 = b'1a64m678'`, and that `K2` is partially known, `K2 = b'1bq4f_ _ _'` (i.e., the last three values are still unknown). Find the missing values of `K2`.

Solution. Either of the following values of `K2` is a match:

```
K2 = b'1bq4ffd2'
K2 = b'1bq4ffd3'
K2 = b'1bq4ffe2'
K2 = b'1bq4ffe3'
K2 = b'1bq4fgd2'
K2 = b'1bq4fgd3'
K2 = b'1bq4fge2'
K2 = b'1bq4fge3'
```

The reason for having multiple matches is that DES doesn't use parity bits for encryption. Finally, the code snippet of Listing 4.6 sketches a possible solution.

```
for i in range(0,2**55):
    kx = K2 + os.urandom(3)
    cipherx = Cipher(algorithms.TripleDES(kx), modes.ECB(), backend
    =backend)
    decryptorx = cipherx.decryptor(); xx = decryptorx.update(C);
    decryptorx.finalize()
    if xx==x: # remember that x = E (K1,P)
        print('match in {} trials; kx={}'.format(i,kx))
        break
```

Listing 4.6 Solution snippet.

□

Exercise 4.7 Write a Python program using the cryptography library, that:

a) Encrypts and decrypts a ≥ 5 MB file with AES-256 in the following modes: ECB, CBC, CFB, CFB8, OFB, CTR, XTS, and GCM. Include ARC4 (insecure) and its "successor" ChaCha20. In total, 10 encryption/decryption pairs.
b) Measures the elapsed time (in milliseconds) for both encryption and decryption for each of the 10 encryption/decryption pairs.
c) Prints the results clearly in a table-like format.
d) Includes your thoughts about the times, comparing:

   a. For each of the 10 options, your thoughts about encryption vs decryption time.
   b. Your thoughts comparing the decryption times across the 10 options.

Solution. Listing 4.7 provides a full solution, including some thoughts for Part d).

```
"""
AES and Stream Cipher Performance Comparison

Last modified: February 2026
Author: Prof. Miguel Vargas Martin

This program benchmarks the encryption and decryption time of
    several
block cipher modes (AES) and stream ciphers (RC4, ChaCha20).

       IMPORTANT:
- Timing results are implementation- and hardware-dependent.
- This code is for educational and comparative purposes only.
- Several constructions shown here (ECB, RC4) are
    cryptographically insecure
  and MUST NOT be used in real systems.
"""

import os, time
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.decrepit.ciphers import algorithms as
    decrepit_algs

def getPlaintext():
    """
    Reads a binary file and returns its contents as plaintext.

    We use a moderately large file (~5 MB) to make timing
    differences
    between cipher modes observable.

    Returns:
        bytes: plaintext message, or b'None' on error
    """
    try:
```

```
        # print('File name:', end=' '); infile = open(input(), '
    rb')
        # OR, instructors may replace this file with any
    sufficiently large binary
        infile = open('large_file.pptx', 'rb') # large_file
    should be large (e.g., > 5 MB)
        message = infile.read()
        infile.close()
        return message
    except:
        print('An error occurred')
        return b'None'

# Cryptographic backend abstraction (OpenSSL underneath)
backend = default_backend()

# 256-bit AES key (hard-coded for reproducibility; DO NOT do this
     in practice)
k = b'abcdefghijklmnop'*2

# Random 128-bit IV (used by most AES modes)
iv = os.urandom(16)

# List of AES modes to benchmark
# Note: ECB is included strictly for comparison and pedagogical
    reasons
ms = [modes.ECB(), modes.CBC(iv), modes.CFB(iv), modes.CFB8(iv),
    modes.OFB(iv), modes.CTR(iv), modes.GCM(iv)]

message = getPlaintext()
print('mode\t\ttime' + '\t\t\t')

for m in ms:
    # PKCS#7 padding is required for block modes operating on
    arbitrary-length data
    padder = padding.PKCS7(128).padder()

    cipher1 = Cipher(algorithms.AES(k), m, backend=backend)

    # Encryption:
    encryptor1 = cipher1.encryptor()
    plaintext = padder.update(message)
    plaintext += padder.finalize()
    t1 = time.perf_counter()
    ct1 = encryptor1.update(plaintext)
    t2 = time.perf_counter()
    encryptor1.finalize()

    print('{} enc\t\t{:.3f} ms'.format(m.name, 1000*(t2-t1)))

    # Decryption:
    # GCM requires special handling due to authentication tags
    if(m.name != "GCM"):
        decryptor1 = cipher1.decryptor()
        t1 = time.perf_counter()
```

```
        pt1 = decryptor1.update(ct1)
        t2 = time.perf_counter()
        decryptor1.finalize()
        print('{} dec\t\t{:.3f} ms'.format(m.name, 1000*(t2-t1)))
    else:
        # GCM decryption requires the authentication tag
    generated at encryption
        tag = encryptor1.tag
        cipher1 = Cipher(algorithms.AES(k), modes.GCM(iv, tag),
    backend=backend)
        decryptor1 = cipher1.decryptor()
        t1 = time.perf_counter()
        decryptor1.authenticate_additional_data(b"")
        pt1 = decryptor1.update(ct1)
        t2 = time.perf_counter()
        decryptor1.finalize()
        print('{} dec\t\t{:.3f} ms'.format(m.name, 1000*(t2-t1)))

# Stream ciphers
# RC4 (deprecated and insecure, included for historical
    comparison)
cipher2 = Cipher(decrepit_algs.ARC4(k), mode=None, backend=
    backend)
encryptor2 = cipher2.encryptor()
t1 = time.perf_counter()
ct2 = encryptor2.update(plaintext)
t2 = time.perf_counter()
print('RC4 enc\t\t{:.3f} ms'.format(1000*(t2-t1)))

decryptor2 = cipher2.decryptor()
t1 = time.perf_counter()
ct2 = decryptor2.update(ct2)
t2 = time.perf_counter()
print('RC4 dec\t\t{:.3f} ms'.format(1000*(t2-t1)))

# ChaCha20 (modern stream cipher, designed to be fast in software
    )
cipher3 = Cipher(algorithms.ChaCha20(k,iv), mode=None)
encryptor3 = cipher3.encryptor()
t1 = time.perf_counter()
ct3 = encryptor3.update(plaintext)
t2 = time.perf_counter()
print('ChaCha20 enc {:.3f} ms'.format(1000*(t2-t1)))

decryptor3 = cipher3.decryptor()
t1 = time.perf_counter()
ct3 = decryptor3.update(ct3)
t2 = time.perf_counter()
print('ChaCha20 dec {:.3f} ms'.format(1000*(t2-t1)))

# AES-XTS (used for disk encryption)
k = b'0123456789abcdeffedcba9876543210'
sector = b'1234567800000000'
cipher4 = Cipher(algorithms.AES(k), modes.XTS(sector), backend=
    backend)
```

```
encryptor4 = cipher4.encryptor()
t1 = time.perf_counter()
ct4 = encryptor4.update(plaintext)
t2 = time.perf_counter()
print('XTS enc\t\t {:.3f} ms'.format(1000*(t2-t1)))

decryptor4 = cipher4.decryptor()
t1 = time.perf_counter()
ct4 = decryptor4.update(ct4)
t2 = time.perf_counter()
print('XTS dec\t\t {:.3f} ms'.format(1000*(t2-t1)))

# Part d)a.
'''
mode    time
ECB enc   2.278 ms
ECB dec   4.026 ms
enc is generally faster.
This is in part due to encryption and decryption not being the
    same algorithm.
But most importantly, keep in mind that the matrix used in the
    MixColumns transformation is optimized for encryption, not
    for decryption.

CBC enc   5.258 ms
CBC dec   2.340 ms
dec is generally faster.
Note that unlike encryption, decryption can be parallelized.

CFB enc   6.957 ms
CFB dec   6.900 ms
dec is generally faster.
Decryption can be parallelized, encryption cannot.

CFB8 enc    91.941 ms
CFB8 dec    89.420 ms
dec is generally faster.
Idem (decryption can be parallelized, encryption cannot.)

OFB enc   5.579 ms
OFB dec   6.238 ms
After many observations, there is not a clear winner because enc
    and dec are identical.

CTR enc   2.907 ms
CTR dec   2.576 ms
Idem.

GCM enc   2.846 ms
GCM dec   2.668 ms
Idem.

RC4 enc   10.709 ms
RC4 dec   11.519 ms
Idem.
```

```
ChaCha20 enc 3.942 ms
ChaCha20 dec 3.326 ms
Idem.

XTS enc    2.366 ms
XTS dec    2.681 ms
enc is generally faster.
This is in part due to encryption and decryption not being the
    same algorithm.
But most importantly, keep in mind that the matrix used in the
    MixColumns transformation is optimized for encryption, not
    for decryption.
'''

# Part d)b
'''
From fastest to the slowest:
CBC dec   2.340 ms The power of parallelization!
CTR dec   2.576 ms CBC consistently beats CTR. This is most
    likely implementation-dependent.
GCM dec   2.668 ms GCM requires the GHASH calculation.
XTS dec    2.681 ms After a few observations, no clear winner
    between XTS and GCM.
ChaCha20 dec 3.326 ms If you run the code in a hardware-optimized
     AES machine, it is likely that AES will outperform non-AES
    algorithms.
ECB dec   4.026 ms The weakness of sequentiality.
OFB dec   6.238 ms OFB may be pre-computable but not
    parallelizable.
CFB dec   6.900 ms CFB may be parallelizable once ciphertext is
    available, but it is not pre-computable.
RC4 dec   11.519 ms If you run the code in a hardware-optimized
    AES machine, it is likely that AES will outperform non-AES
    algorithms.
CFB8 dec    89.420 ms CFB needs to process ~16x the number of
    blocks than other modes.
'''
```

Listing 4.7 Solution to Exercise 4.7.

□

Exercise 4.8 Suppose you work for an organization that uses a "control vector"[4] (cv) scheme to manage session keys. Encryption and decryption of the session key (sk) are computed as follows:

esk=E(Hash(cv) XOR mk, sk), and
sk=D(Hash(cv) XOR mk, esk),

[4] This scheme was proposed by IBM in Matyas, S.M. *Key processing with control vectors*. J. Cryptology 3, 113–136 (1991).

where `esk` stands for "encrypted sk", `mk` is the "master key". `Hash` is a hash function (refer to exercises on hash functions in Chapter 5).

This control vector scheme uses SHA-256 as the `Hash`, and AES-256 in ECB mode without padding as encryption `E` and decryption `D`. The control vector `cv` associated to your employee ID is given below, along with a master key (`mk`) that IT Services has shared with you in a USB flash drive.

```
cv = b'full-time, admin5, top security clearance' #any number of bits
mk = b'thisKeyWasGivenToYouViaAUSBbyITS' #256 bits
esk = b'\x1ft0<\xff\xd4\xb1\xa5\x1a\xdd\xadM\x19\xc2\xcd\xa6' #128 bits
c = b'\xd4\xce\x837:6\xf6K\xd3\xa6\x936-\xd80\xf8' #128 bits
```

1. Obtain the session key `sk` off the encrypted session key `esk` above. Note: to make debugging easier, `sk` contains English text.
2. Decrypt ciphertext `c` which was encrypted using AES in ECB mode with key `sk`. Don't use padding. Note: to make debugging easier, the plaintext contains English text.

Solution. Listings 4.8 and 4.9 offer sketches of possible solutions to parts 1 and 2.

```
def xor(a, b):
    return (int.from_bytes(a,'big')^int.from_bytes(b,'big')).
    to_bytes(32,byteorder='big')

digest = hashes.Hash(hashes.SHA256())
digest.update(cv)
h=digest.finalize()
K=xor(h,mk)

cipher = Cipher(algorithms.AES(K), modes.ECB(), backend=backend)
decryptor = cipher.decryptor()
sk=decryptor.update(esk)
decryptor.finalize()
print('sk:', sk) # b'secretKey-a3d8fg'
```

Listing 4.8 Sketched solution for Part 1.

```
cipher = Cipher(algorithms.AES(sk), modes.ECB(), backend=backend)
decryptor2 = cipher.decryptor()
p=decryptor2.update(c)
decryptor2.finalize()
print('p:', p) # b'Oh wow! * three!'
```

Listing 4.9 Sketched solution for Part 2.

□

4.3 Key Wrapping

Exercise 4.9 You are given wrapped key `wdk` wrapped with wrapping key `wgk`. Unwrap `wdk` and use it to decrypt ciphertext `c`, which was encrypted using AES in ECB mode without padding. Note: to make debugging easier, both the unwrapped key and the plaintext contain English text.

```
wdk=b'\xec\xfc|\x9dMu\x84\x90\xd4%\x14\x0e\n_\x11\x00B\xdc\xa4h\xc6\x0bT\xf3'
wgk = b"This'WrappingKey"
c = b'}\xf9\x04G\xc7\xff\xdfQNMy\xa09\xa9\xdd/'
```

Solution. Listing 4.10 shows a sketch of a possible solution.

```
uk=aes_key_unwrap(wgk, wdk)
cipher = Cipher(algorithms.AES(uk), modes.ECB(), backend=
    default_backend())
decryptor2 = cipher.decryptor()
p=decryptor2.update(c)
decryptor2.finalize()
print('p:', p) #  b'helpIneedSomebdy'
```

Listing 4.10 Sketched solution.

□

Exercise 4.10 Johnny obtained a wrapped_key by wrapping a plain_key a while ago. He used cryptography.io's `aes_key_wrap`. Unfortunately, he just realized that he lost the wrapping_key, and of course he doesn't have the plain_key. Fortunately for him, he was able to recover all but the last byte of the wrapping_key. Please help Johnny recover his plain_key.

```
wrapped_key = b'\xa4\xe00\xdb\xfa\x02\x18\xaa\x85\xb2\x9db\xa3\xe2\x02\xea\xe2
\xf3\x9d}\xa5\xa9\xba('
wrapping_key = b'kdnc649snc04ajx' (missing last character, right after 'x')
```

Solution. A sketch of a possible solution is offered in Listing 4.11.

```
k = b'kdnc649snc04ajx'
for _ in range (2**16):
    kk = k + os.urandom(1)
    try:
        uk = aes_key_unwrap(kk,
b'\xa4\xe00\xdb\xfa\x02\x18\xaa\x85\xb2\x9db\xa3\xe2\x02\xea\xe2\
    xf3\x9d}\xa5\xa9\xba(',
                            backend=default_backend())
        print(uk) # b'ThanK You FolKs!'
        print(kk) # b'kdnc649snc04ajx&
        break
    except:
        continue
```

Listing 4.11 Sketched solution.

□

Chapter 5
Digest Functions

> *"We were struck with the fact, that in all the mass of material of which the record is composed, there is hardly one authentic document."*
>
> —Bram Stoker, Dracula

This chapter explores hash functions and Message Authentication Codes (MACs). Although MACs are not technically "digest functions" because they require a secret key, they are functionally similar: both compress message content into a fixed-size value to ensure data integrity.

5.1 Hash Functions

Exercise 5.1 You are provided with an MD5[1] hash value and partial information about the original message `y`. You are given the hash `MD5(y)`:

```
MD5(y) = h = b'\xb7\xd2v~w\xd4=\x86\xef\xf7\x84sv\xb0\xc7\x16'
```

It is known that `y` is a DES key (64 bits), with the first 48 bits being `b'imagin'`

Determine the last 16 bits (2 bytes) of `y`.

Solution. Code snippet of Listing 5.1 offers a possible solution.

```
h=b'\xb7\xd2v~w\xd4=\x86\xef\xf7\x84sv\xb0\xc7\x16'
y=b'imagin'
for _ in range (2**32):
    y1 = y + os.urandom(2)
    digest = hashes.Hash(hashes.MD5(), backend=default_backend())
    digest.update(y1)
    h1 = digest.finalize()
    if h1 == h:
        print(y1) #b'imagine+'
        print(f"found in {_} attempts")
```

[1] Weaknesses in MD5 have been found for over 30 years. Needless to say, it is cryptographically insecure and must not be used in real systems.

```
        break
```

Listing 5.1 Sketch of a possible solution.

□

Exercise 5.2 Johnny comes up with a signature[2] scheme to sign two or more strings of equal length `s1, s2, ..., sn` by appending a hash value to the concatenated strings, i.e., a signed group of strings would be represented by the following concatenation:

```
s1 + s2 + ... + sn + Hash(s1, s2, ..., sn)
```

All looks acceptable up to here. However, Johnny used a weak hash function. His hash function is computed as the bitwise XOR of the strings, as in the example below:

```
#legit message:
s1='sell everything'
s2='and tell no one'
hash1 = [(ord(a) ^ ord(b)) for a,b in zip(s1,s2)]
```

He later discovered that an attacker could easily craft forged messages that yield signatures identical to those of valid messages. For example:

```
#bogus message:
s3='buy all at once'
s4='/transaction #:'
s5= # This value is withheld
hash2 = [(ord(a) ^ ord(b) ^ ord(c)) for a,b,c in zip(s3,s4,s5)]
#hash1 is equal to hash2
```

Find the value of `s5` in the example above. To facilitate debugging, don't expect `s5` to contain regular English text, or even regular printable characters, as it may contain nonprintable ones such as `\n`, `\r`, or hex numbers such as `\x05`.

Solution (or not). Since the examples already provide substantial guidance, the implementation details are left to the reader. The solution is `s5 = '_\n\x03\r[\x1f\x17JG\nTIH@]'`.

□

Exercise 5.3 In a hacking competition, you need to brute-force an MD5 hash value, found in a 1994 PlayStation, against the one-way property (i.e., for a given hash value `h`, find a `y` such that `MD5(y) = h`). You are provided with the following hash value:

```
MD5(y) = h = b'\x1dP\x00q6\x0b5\x96\x1b\x86\x90\xa57\xc2\xcb\x1a'
```

[2] We will study public-key signatures in Chapter 7.

You know that the value `y` was definitely a DES key (i.e., 64 bits long) with the first 48 bits "likely" being `y = b'dh$g.u'`. That is, you need to find the last two bytes of `b'dh$g.u'`.

Solution. The code in Listing 5.2 offers a sketch to the solution.

```
y = b'dh$g.u'
for _ in range (2**32):
    y1 = y + os.urandom(2)
    digest = hashes.Hash(hashes.MD5(), backend=default_backend())
    digest.update(y1)
    h1 = digest.finalize()
    if h1 == h:
        print(y1) #b'dh$g.uj#'
        break
```

Listing 5.2 Solution sketch.

□

Exercise 5.4 Consider Johnny's hash function of Exercise 5.2. Reflect whether or not Johnny's hash function is preimage resistant (one-way property), second preimage resistant (weak collision resistant), and collision resistant (strong collision resistant).

Solution. The reflection is left to the reader, but it is clear that none of these properties are met by Johnny's hash function. A hash function based solely on XOR is linear and therefore fails preimage resistance, second preimage resistance, and collision resistance.

□

Exercise 5.5 This exercise illustrates the birthday paradox in the context of hash collisions, using a simple toy "hash" function based on bitwise XOR, and demonstrating that collisions can occur much more easily than intuition may suggest.

Write a Python program that performs the following steps:

1. Generate XOR "hashes" for prime sets:
   - Randomly generate 16 sets[3] of three distinct *prime* numbers, all less than 256 (2^8).
   - Compute the hash of each set by taking the bitwise XOR of the three integers:

     `h = p1 XOR p2 XOR p3 (def xor_hash(a,b,c): return a^b^c)`

     Store these 16 hash values as your reference set.
2. Search for a matching composite set:

[3] Remember that a set is an unordered collection of distinct items, so no repetitions are allowed. For example, $\{5, 11, 5\}$ is invalid, and $\{2, 7, 19\}$ is the same as $\{19, 2, 7\}$.

- Repeatedly generate random sets of three distinct *composite* numbers less than 256.
- Compute the XOR hash for each composite set and compare it to the 16 prime-set hashes in your reference set.
- Stop as soon as you find a composite set whose hash matches one of the prime-set hashes in your reference set.

Solution. Because XOR outputs very small integer values (only a few bits when numbers $< 2^8$), collisions should appear quickly, often after testing a dozen composite sets. This models the birthday paradox: the probability of two values colliding rises sharply once you generate roughly $\sqrt{N}$ distinct hashes, where N is the number of possible outputs (in the case at hand, after generating $\sqrt{256} = 16$ hashes).

□

Now let's see a rather foolish attempt at finding SHA-256 collisions.

Exercise 5.6 This exercise extends Exercise 5.5 by replacing the simple XOR operation with a cryptographically secure hash function (SHA-256). The goal is to show empirically how unlikely it is to find *any* collision, that is, any two different inputs with the same digest, when using a modern cryptographic hash function.

Write a Python program that performs the following steps:

1. Generate SHA-256 digests of prime sets:

   - Randomly generate 16 sets of three distinct *prime* numbers, all less than 256.
   - For each set $\{p_1, p_2, p_3\}$, compute the SHA-256 digest of the concatenation of their ASCII decimal representations (e.g., `b"251", b"157", b"191"`).
   - Store these 16 hash values as your reference digests.

2. Generate SHA-256 digests of composite sets:

   - Generate 1 000 sets of three distinct *composite* numbers, all less than 256.
   - Compute their SHA-256 digests in exactly the same format as in Step 1.
   - For each composite digest, check whether it matches any of the 16 prime digests. Stop early if a collision is found.
   - To avoid accidental false collisions:
     – Do not use `bytes(n)` directly, as it produces a sequence of null bytes (`\x00`) that can unintentionally overlap between inputs. For example, `bytes(3)` generates `\x00\x00\x00`.
     – Avoid ambiguous string concatenations such as `"2" + "41" + "227"` and `"24" + "12" + "27"`, which both yield `"241227"`. Instead, include delimiters (e.g., `b':'`) or use separate `digest.update(b"...")` calls.

Solution. You are not expected to find a real collision. Even if you computed all $P(200, 3) = 7,880,400$ possible permutations of three composite numbers less than 256, a true collision with SHA-256 would still be astronomically unlikely, requiring roughly 2^{128} (that is $\approx 3.40282367 \times 10^{38}$, a 39-digit number) trials on average. For perspective, generating 1.39×10^6 SHA-256 hashes takes about 40 seconds on my regular off-the-shelf laptop, but 2^{128} hashes would take $\approx 3.1 \times 10^{26}$ years (that is

approx. 310 septillion years, a 27-digit number of years). This is far longer than the expected lifetime of our home galaxy, the Milky Way.

□

5.2 SHA-3 Hash and Extendable-Output Functions

Let's now turn our attention to the SHA-3 family, standardized in FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. We will cover SHA-3-512, and `SHAKE-128`.

Exercise 5.7 Repeat Exercise 5.6 but now use SHA-3-512.

Solution. The program in Listing 5.3 shows the use of SHA-3-512 in cryptography.io. The remaining is left to the reader.

```
from cryptography.hazmat.primitives import hashes
digest = hashes.Hash(hashes.SHA3_512())
data = b"Long live PQC!"
digest.update(data)
hash_bytes = digest.finalize()
print(hash_bytes)
```

Listing 5.3 Use of SHA-3-512 in cryptography.io.

□

Unlike traditional hash functions like SHA-256, `SHA-256`, `SHAKE-128` is an Extendable-Output Function (XOF), meaning it can generate an output of any length. The term used to obtain an output from `SHAKE-128` is "squeeze," and the output bits do not change based on the requested length: a shorter output is simply a prefix of a longer one. The "128" refers to the maximum security strength of the function, not to its output size. Requesting more output never raises the strength above 128 bits, but requesting too little lowers it. For an output of d bits, FIPS 202 gives a collision resistance of $\min(d/2, 128)$ bits and a preimage (and second preimage) resistance of $\min(d, 128)$ bits. For example, a 256-bit output offers 128 bits of collision resistance, the most SHAKE-128 can provide, whereas a 128-bit output offers only 64. In other words, you need to squeeze at least 256 bits to obtain the full 128-bit strength against collisions, and anything beyond that adds length but not security. Listing 5.4 demonstrates the use of `SHAKE-128` with Bouncy Castle.

```
/**
 * Demonstration of SHAKE128 using Bouncy Castle.
 *
 * SHAKE128 is an Extendable Output Function (XOF) defined in FIPS
    202.
```

```
 * Unlike SHA-256 or SHA-3-256, SHAKE allows arbitrary-length
    output.
 *
 * Internally, SHAKE is based on the Keccak sponge construction.
 *
 * Author: ChatGPT, modified by Miguel Vargas Martin
 *
 * Last modified: February 2026
 */

import org.bouncycastle.crypto.digests.SHAKEDigest;
import org.bouncycastle.util.encoders.Hex;

public class SHAKE128 {

    public static void main(String[] args) {

        // Input message to be hashed
        String input = "example data";

        // Convert the input string to bytes (UTF-8 encoding is
    recommended explicitly)
        byte[] inputBytes = input.getBytes();


        // 1. Initialize SHAKE128
         // The parameter (128) indicates 128-bit security strength
    .
        // This does NOT mean the output is 128 bits long.
        // Output length is chosen during the squeezing phase.
        SHAKEDigest digest = new SHAKEDigest(128);

         // 2. Absorb phase (update the sponge with input data)
         // The update() method feeds data into the sponge
    construction.
        // Parameters:
        //   - input array
        //   - offset (start position)
        //   - length (number of bytes to read)
        digest.update(inputBytes, 0, inputBytes.length);

         // 3. Squeeze phase (extract output bytes)
         // SHAKE is an XOF: we can request as many output bytes as
     needed.
        // Here, we request 64 bytes (512 bits).
        byte[] output = new byte[64];

        // doOutput() extracts bytes from the sponge.
        // If more bytes are requested later, squeezing continues.
        digest.doOutput(output, 0, output.length);

        // Convert the output to hexadecimal for readable display
        System.out.println("SHAKE128 Output (64 bytes): " + Hex.
    toHexString(output));
    }
```

```
}
```

Listing 5.4 Use of `SHAKE-128` in Bouncy Castle.

5.3 Message Authentication Codes

The Galois/Counter Mode (GCM) explored in Exercise 4.7 functions as a Message Authentication Code. Readers should consult that exercise to complement the material covered in this section.

Exercise 5.8 Find the last two bytes of the partial AES-128 CMAC key below. The full CMAC tag, full message, and partial key (missing the last two bytes) are as follows:

```
tag = b's\xaf\x12\n\xdc\xc6\x16˜\xd0\xcaI\xf3o\x9au0' #full
message = b"message to authenticate" #full
key = b'0123456789abcd' #missing the last two bytes
```

Solution. The program in Listing 5.5 offers a full solution. Note that the computation of the `tag` was unnecessary since it's already given as part of the exercise.

```
# -*- coding: utf-8 -*-
"""
Brute-Forcing a Short AES-CMAC Key

Last modified: February 2026
Author: Miguel Vargas Martin

This program demonstrates why short keys are insecure,
even when used with strong primitives such as AES-CMAC.

We:
  1) Compute a valid CMAC tag using a full 128-bit AES key.
  2) Assume that only part of the key is known.
  3) Brute-force the remaining unknown bits.
  4) Recover the correct key by verifying the MAC.

       Educational purpose only.
       Real systems must NEVER use short or partially secret keys.
"""

from cryptography.hazmat.primitives import cmac
from cryptography.hazmat.primitives.ciphers import algorithms
import os

# Let's compute the tag, instead of copying it from the exercise's
    text

# 16-byte (128-bit) AES key
key = b'0123456789abcdi/'
```

```
message = b"message to authenticate"

# Create CMAC object using AES
c = cmac.CMAC(algorithms.AES(key))
c.update(message)

# Generate authentication tag
tag = c.finalize()

print("Valid tag:", tag) #b's\xaf\x12\n\xdc\xc6\x16~\xd0\xcaI\xf3o\
    x9au0'

# Now let's assume we are missing the last 2 bytes (16 bits) of the
     key

key = b'0123456789abcd'   # 14 known bytes, last 2 are unknown

# Total search space: 2^16 possible values
# (We loop slightly larger here: 2^17 attempts)
for _ in range(2**17):

    # Guess the missing 2 bytes
    key1 = key + os.urandom(2)

    try:
        # Compute CMAC under guessed key
        c = cmac.CMAC(algorithms.AES(key1))
        c.update(message)

        # Check whether computed tag matches the known valid tag
        c.verify(tag)

        # If verification succeeds, we have found the correct key
        print("Recovered key:", key1) # b'0123456789abcdi/'
        break

    except:
        # Verification failed     wrong key guess
        continue
```

Listing 5.5 A possible solution.

□

Exercise 5.9 Write a program that:

1. Measures the time it takes to generates the SHA-256 hash of a large binary file (> 5 MB).
2. Measures the time it takes to generate and verify the MAC of the same binary file.

3. Uses the (now withdrawn) Digital Signature Algorithm[4] (DSA) to sign the same binary file.

   Solution. The implementation is detailed in Listing 5.6, which also features supplementary exercises for readers who wish to explore the topic further.

```
"""
Hashing, HMAC, and Digital Signatures     Timing and Functional
    Comparison

Last modified: February 2026
Author: Original code by Miguel Vargas Martin. TODOs added by
    ChatGPT

This program demonstrates:
1) A cryptographic hash (SHA-256)
2) A keyed MAC (HMAC-SHA-256)
3) A digital signature (DSA)

We compare:
- Functionality
- Verification behavior
- Rough execution time

       Timing values are illustrative only.
       DSA-1024 is used for demonstration purposes (not
    recommended today).
"""

# Helper: Load message from file
def getMessage():
    """
    Reads a binary file and returns its contents.

    Used to simulate hashing/authenticating a realistic-sized
    file
    (~5 MB PowerPoint in this example).
    """
    try:
        # TODO 1:
        # Modify the program so that the filename is read from
    input().
        # Ensure binary mode is preserved.

        infile = open('large_file.pptx', 'rb') # large_file
    should be "large"
        message = infile.read()
        infile.close()
        return message
```

[4] Signatures are covered in Chapters 7 and 10. However this is the right occasion to see one way in which digital signatures make use of hash functions.

```
    except:
        print('An error occurred')
        return b'None'


# Part 1     Cryptographic Hash (SHA-256)

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
import time

# Create SHA-256 hash object
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())

digest.update(getMessage())

# Measure only finalization time (not update time)
t1 = time.time()
hash_value = digest.finalize()
t2 = time.time()

print('Hash computation takes {} seconds'.format(t2 - t1))

# TODO 2:
# Measure the time for BOTH update() and finalize().
# Compare total hashing time with MAC and signature times.

# Part 2     HMAC (Keyed Message Authentication Code)

# 32-byte (256-bit) symmetric key
#        Hardcoded for reproducibility     NEVER hardcode keys in
     real systems.
key = b'12345678' * 4  # TODO 3: Replace with os.urandom(32)

h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
h.update(getMessage())

t1 = time.time()
mac = h.finalize()
t2 = time.time()
print('MAC generation takes {} seconds'.format(t2 - t1))

# HMAC Verification
h1 = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
h1.update(getMessage())

t1 = time.time()
h1.verify(mac)   # Raises exception if verification fails
t2 = time.time()

print('MAC verification takes {} seconds'.format(t2 - t1))

# TODO 4:
# Modify one byte of the message and observe what happens during
    verification.
```

```
# Explain why HMAC verification must be constant-time.

# Part 3     Digital Signature (DSA)
from cryptography.hazmat.primitives.asymmetric import dsa

# Generate DSA private key
#        1024-bit DSA is obsolete for production use.
private_key = dsa.generate_private_key(
    key_size=1024,
    backend=default_backend()
)
data = b"this is some data I'd like to sign"

# Sign the data (internally hashes using SHA-256)
signature = private_key.sign(
    data,
    hashes.SHA256()
)

# Obtain public key for verification
public_key = private_key.public_key()

# Verify signature
public_key.verify(
    signature,
    data,
    hashes.SHA256()
)

# TODO 5:
# Measure the time required for:
#   - Key generation
#   - Signing
#   - Verification
# Compare these times to hash and HMAC.
# Explain why digital signatures are significantly slower.

# TODO 6:
# Replace DSA with RSA or ECDSA and compare:
#   - Key sizes
#   - Signature sizes
#   - Performance
#   - Security levels

# TODO 7:
# Explain conceptually:
#   - Why a hash alone does NOT provide authentication
#   - Why HMAC provides integrity + authentication
#   - Why digital signatures provide non-repudiation
```

Listing 5.6 Timing Hash vs MAC vs signature.

□

Chapter 6
Stream Ciphers and Random Numbers

"For it is even as Thou hast appointed, that every inordinate affection should bring its own punishment."

——St. Augustine of Hippo, Confessions

Exercise 4.7 illustrated stream ciphers RC4 (insecure) and ChaCha20 in action. This chapter focuses on the generation of cryptographically secure random numbers, which are essential for stream ciphers among many other cryptographic operations.

6.1 Random Enough

Exercise 6.1 Some people would claim a large language model (LLM) can be used as a random number generator. Prompt an LLM (e.g., ChatGPT, DeepSeek, Gemini) with: *"Generate a sequence of* 10 000 *bits (*0*s and* 1*s) that looks random."* Then copy the output into a text file. Use Python (code may also be obtained from an LLM) to:

a) Read the LLM sequence from the text file.
b) Generate a 10 000-bit sequence with `secrets.token_bytes()` or some other function from `secrets`.
c) For each sequence, compute:[1]
   a. Monobit frequency (fraction of 1s).
   b. Runs count (number of consecutive flips).
   c. Byte diversity (number of distinct 8-bit values).
   d. Maurer-type compressibility (using `gzip` to determine compression ratios).
d) Within your code, insert comments about the purpose of each test and the expected outcome.
e) Within your code, insert comments about the results for both sequences.

Solution. Listing 6.1 shows a possible solution along with comments about the output. The program assumes the existence of a file named `llm_bits.txt` which is a sequence of 10 000 LLM-generated "random" numbers.

[1] These simple tests are by no means as rigorous as the statistical test suite detailed in NIST SP 800-22 Rev. 1: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Nonetheless, they do resemble, to a degree, some of the tests therein.

```
"""
Simple Statistical Tests for Bit Sequences

Last modified: February 2025

Original version generated by ChatGPT-5
Adapted and extended by Prof. Miguel Vargas Martin

This program compares bit sequences produced by:
  1) a Large Language Model (LLM), and
  2) Python's `secrets` module (cryptographically secure PRNG).

The goal is NOT to prove randomness, but to illustrate how
simple statistical tests can reveal non-random structure.
"""

import secrets
import gzip

def load_bits_from_file(filename):
    """
    Load a sequence of bits from a text file.

    The file may contain arbitrary characters; only '0' and '1'
    are extracted. This allows students to paste raw LLM output
    without preprocessing.

    Args:
        filename (str): path to input file

    Returns:
        list[int]: list of bits (0s and 1s)
    """
    with open(filename, "r") as f:
        text = f.read()
    return [int(ch) for ch in text if ch in "01"]

def monobit_frequency(bits):
    """
    Compute the frequency of 1s and 0s (monobit test).

    For a random sequence, the fraction of 1s and 0s
    should be close to 0.5, but small deviations are expected
    for finite-length sequences.

    Args:
        bits (list[int]): input bit sequence
    Returns:
        tuple(float, float): (fraction_of_ones, fraction_of_zeros)
    """
    n = len(bits)
    ones = sum(bits)
    return ones / n, (n - ones) / n

def runs_count(bits):
```

```
    """
    Count the number of runs in the bit sequence.

    A run is a maximal subsequence of identical bits
    (e.g., 111 or 00). Too many or too few runs may
    indicate non-random structure.

    Args:
        bits (list[int]): input bit sequence
    Returns:
        int: number of runs
    """
    runs = 1
    for i in range(1, len(bits)):
        if bits[i] != bits[i - 1]:
            runs += 1
    return runs

def byte_diversity(bits):
    """
    Measure byte-level diversity.

    The bit sequence is grouped into 8-bit chunks and
    interpreted as bytes. We then count how many distinct
    byte values appear.

    For truly random data, most of the 256 possible byte
    values should occur when the sequence is long enough.

    Args:
        bits (list[int]): input bit sequence
    Returns:
        int: number of distinct byte values observed
    """
    byte_vals = [
        int("".join(map(str, bits[i:i + 8])), 2)
        for i in range(0, len(bits), 8)
    ]
    return len(set(byte_vals))

def gzip_compression_ratio(bits):
    """
    Estimate compressibility using gzip.

    Random data is largely incompressible, while structured
    or biased data tends to compress better.

    The ratio is:
        compressed_size / original_size

    Values closer to 1 indicate low compressibility and
    therefore higher apparent randomness.

    Args:
        bits (list[int]): input bit sequence
```

```
    Returns:
        float: gzip compression ratio
    """
    byte_vals = bytes(
        int("".join(map(str, bits[i:i + 8])), 2)
        for i in range(0, len(bits), 8)
    )
    original_size = len(byte_vals)
    compressed = gzip.compress(byte_vals)
    compressed_size = len(compressed)
    return compressed_size / original_size

# Data sources

# Load bits generated by an LLM (stored as text)
llm_bits = load_bits_from_file("llm_bits.txt")

# Generate 10,000 bits using a cryptographically secure PRNG
# 1250 bytes    8 bits = 10,000 bits
seq_secrets = [
    int(b)
    for byte in secrets.token_bytes(1250)
    for b in f"{byte:08b}"
][:10000]

# Analysis and comparison
for name, seq in [("LLM", llm_bits), ("secrets", seq_secrets)]:
    ones_frac, zeros_frac = monobit_frequency(seq)
    runs = runs_count(seq)
    unique_bytes = byte_diversity(seq)
    ratio = gzip_compression_ratio(seq)

    print(f"=== {name} sequence ===")
    print(f"Length: \t\t\t\t{len(seq)} bits")
    print(f"Fraction of 1s: \t\t{ones_frac:.4f}")
    print(f"Runs: \t\t\t\t{runs}")
    print(f"Unique bytes: \t\t\t{unique_bytes}")
    print(f"Gzip compression ratio: \t{ratio:.3f}\n")

'''
=== LLM sequence ===
Length:          10000 bits
Fraction of 1s:      0.5000
Runs:            5540
Unique bytes:        230
Gzip compression ratio: 1.018

=== secrets sequence ===
Length:          10000 bits
Fraction of 1s:      0.4984
Runs:            4941
Unique bytes:        256
Gzip compression ratio: 1.018

The expected fraction of 1s in a random sequence is 0.5. While the
```

```
LLM-generated sequence achieved perfect balance in this metric, it
performed poorly on the others.

The expected number of runs is approximately n/2 (here, 5,000 for
10,000 bits). The sequence generated by secrets exhibited a healthy
level of alternation, whereas the LLM sequence showed reduced
variability, indicating underlying structure.

For byte diversity, the ideal value is 256 (the full range of
8-bit patterns). The secrets sequence reached this target, while
    the
LLM sequence missed 26 distinct byte values, suggesting limited
entropy.

The expected gzip compression ratio is around 1.0. Ratios higher
than 1.0 indicate data that is effectively incompressible, aside
from minor gzip metadata overhead.

In summary, an incautious practitioner might be misled if relying
only on a few superficial randomness tests. Comprehensive
evaluation across multiple metrics is essential to detect subtle
patterns or biases. Cryptographic applications should always rely
on cryptographically secure random generators (such as the secrets
module in Python) rather than sequences produced by non-random
sources like LLMs.
'''
```

Listing 6.1 Random Number Generation

□

Exercise 6.2 Generate two sets of 1 000 random numbers using the following methods:

1. The first set uses a linear congruential random number generator (LCG) with the following parameters:
   `a=5`
   `x=2`
   `c=3`
   `m=1009`
   `x=(a*x+c) % m`

2. The second set uses `os.urandom`, a cryptographically secure pseudo-random number generator (CSPRNG).

Then plot each of the two sets of random numbers on different scatter plots (with a sequential number in the x-axis and the random number in the y-axis), compare the graphs, and report your naked-eye observations. Also compare compression rates using `gzip`.

Solution. Listing 6.2 offers a possible solution. Output graphs are shown in figures 6.1 and 6.2. Observing both scatter plots for a moment would reveal visible patterns in the LCG graph, whereas the CSPRNG graph shows no apparent signs of any patterns (the

sequence is indeed cryptographically secure). The compression rate of the LCG is expected to be larger than that of the CSPRNG. The `gzip`-compressed files turned out to be 979 bytes (`rand_strd.gz`) and 2 055 bytes (`rand_crypto.gz`) long.

```
"""
Visual and Compression-Based Comparison of Random Number Generators

Created on: February 2026
Author: Prof. Miguel Vargas Martin

This program compares:
  1) A cryptographically secure random number generator (CSPRNG),
    and
  2) A simple linear congruential generator (LCG).

The comparison is performed using:
  - Scatter plots (visual structure),
  - Gzip compression (empirical compressibility).

       IMPORTANT:
- This code is for educational purposes only.
- LCGs are NOT cryptographically secure.
- Visual randomness does NOT imply cryptographic security.
"""

import random
from sympy import nextprime
import matplotlib.pyplot as plt
import gzip

# Parameters
iter = 1000          # Number of random values generated
fill = 4             # Number of decimal digits retained per value

# Byte strings used for compression comparison
rand_crypto = b''
rand_strd = b''

# Cryptographically secure RNG
# SystemRandom draws entropy directly from the OS (CSPRNG)
rng = random.SystemRandom()

for i in range(iter):
    x = rng.random()                    # Uniform float in [0, 1)
    plt.scatter(i, x, color='b', alpha=0.3)# Visual inspection of
    randomness

    # Keep only the last few decimal digits to create a byte stream
    # (This is intentionally lossy and pedagogical, not secure.)
    rand_crypto += str(x)[-fill:].encode('utf-8')

plt.title("CSPRNG output (SystemRandom)")
plt.xlabel("Iteration")
plt.ylabel("Random value")
```

```
plt.show()

# Linear Congruential Generator (LCG)
# LCG parameters (chosen for simplicity, not quality)
a = 5           # multiplier
c = 3           # increment
x = 2           # seed

# Modulus chosen as a prime slightly larger than the number of
    samples
m = nextprime(iter)

# First iteration
x = (a * x + c) % m

for i in range(iter):
    # Normalize to [0, 1) for visual comparison with CSPRNG
    plt.scatter(i, x / iter, color='r', alpha=0.3)

    x = (a * x + c) % m

    # Store fixed-width decimal representation
    # This preserves structure and makes compression effects
    visible
    rand_strd += str(x).zfill(fill).encode('utf-8')

plt.title("LCG output")
plt.xlabel("Iteration")
plt.ylabel("Normalized value")
plt.show()

# Compression-based comparison

# Both byte streams should have comparable size
print(len(rand_strd), len(rand_crypto))

# Compress both outputs using gzip
# Structured data (LCG) should compress significantly better
with gzip.open('rand_crypto.gz', 'wb') as c:
    c.write(rand_crypto)

with gzip.open('rand_strd.gz', 'wb') as s:
    s.write(rand_strd)
```

Listing 6.2 Random number generation.

□

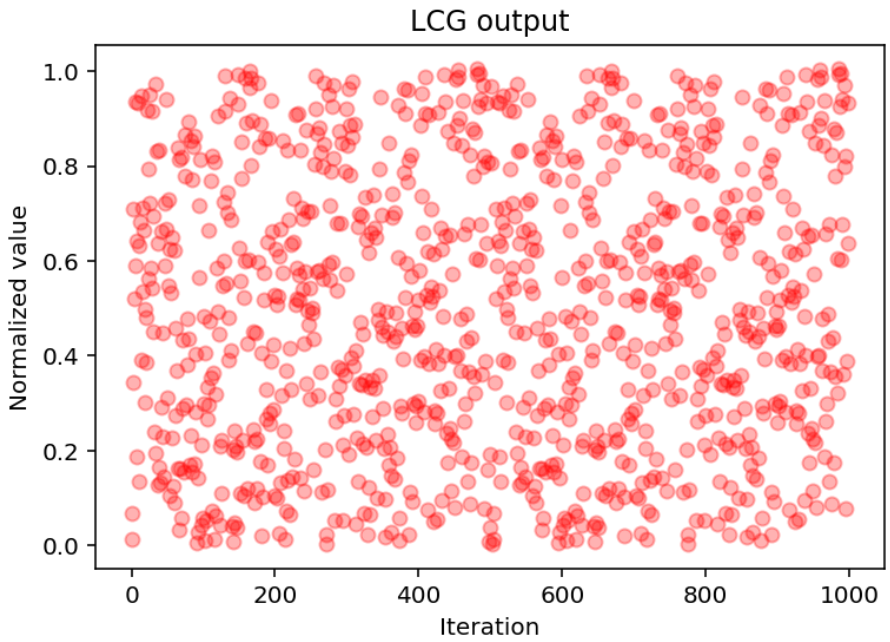


Fig. 6.1 Random numbers from a linear congruential generator.

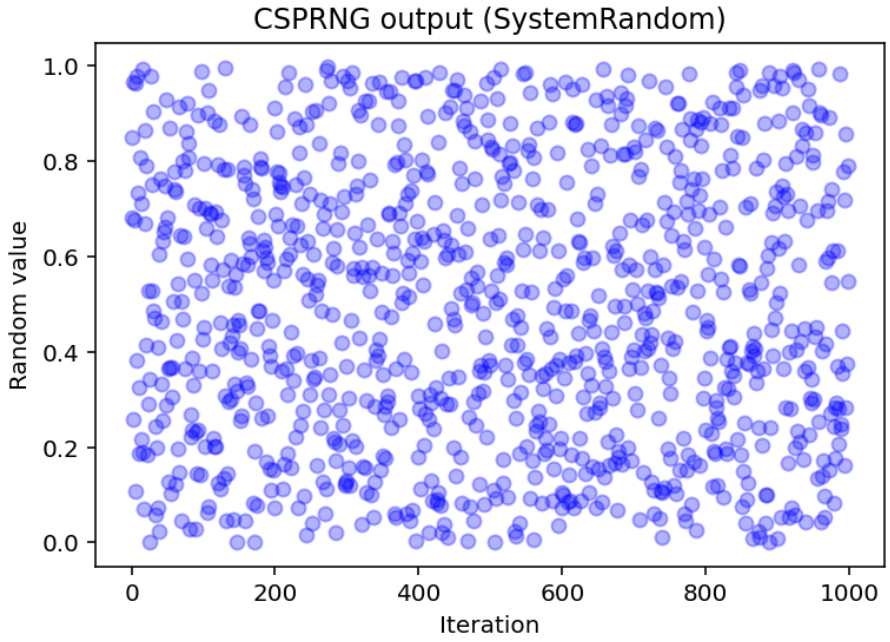


Fig. 6.2 Random numbers from SystemRandom.

6.2 Speed of Cryptographically Secure Generators

Exercise 6.3 This may be counterintuitive, but generating cryptographically secure pseudo-random numbers is generally fast. Write a Python program that compares times of a LCG and a CSPRNG.

Solution. Listing 6.3 offers a possible solution using Python's standard generator[2] for comparison. This solution also includes counts of the number of 1s in the sequences, just for kicks. In my conventional off-the-shelf machine, CSPRNG takes 3.311 ms, while the standard generator takes 125.231 ms.

```
"""
Cryptographically Secure vs. Standard Randomness:
Timing and Bit Statistics

Last modified: February 2026
Author: Dr. Miguel Vargas Martin

This program compares randomness obtained from:
  1) os.urandom()      cryptographically secure entropy source
  2) random.randint()     deterministic PRNG (Mersenne Twister)

The comparison is based on:
  - Average number of 1-bits in 128-bit values
  - Generation time
  - Compressibility of the generated byte streams
  - Visual inspection via scatter plots

       IMPORTANT:
- Passing simple statistical tests does NOT imply cryptographic
    security.
- The standard ‘random‘ module is NOT suitable for cryptography.
"""

import os
import random
import math
import time

# Bit statistics and timing (128-bit values)

iter = 15000                # Number of random samples
random.seed()               # Seed PRNG from system time (default
    behavior)

rand_crypto = b''           # Byte stream from os.urandom()
rand_strd = b''             # Byte stream from random.randint()

# --- Cryptographically secure random numbers ---
ones = 0                    # Total number of 1-bits observed
```

[2] Python, among other languages, uses the *Mersenne Twister* as the default random number generator.

```
t = 0                          # Total time spent generating
    randomness

for i in range(iter):
    tt1 = time.perf_counter()
    iv = os.urandom(16)      # 16 bytes = 128 bits of cryptographic
    randomness
    tt2 = time.perf_counter()

    t += (tt2 - tt1)

    # Convert to integer for bit-level analysis
    serial = int.from_bytes(iv, byteorder="big")

    # Count number of 1s in the 128-bit value
    ones += bin(serial)[2:].count('1')

    # Accumulate raw bytes for compression testing
    rand_crypto += iv

print(
    "Avg # of 1's in {:,} 128-bit crypto random numbers: {:.3f}; "
    "time = {:.3f} ms".format(iter, ones / iter, 1000 * t)
)

# --- Standard PRNG (Mersenne Twister) ---
ones = 0
t = 0
val = math.pow(2, 128)        # Upper bound for 128-bit values

for i in range(iter):
    tt1 = time.perf_counter()
    iv = random.randint(0, val)
    tt2 = time.perf_counter()

    t += (tt2 - tt1)

    ones += bin(iv)[2:].count('1')

    # Store as fixed-length 16-byte value
    rand_strd += iv.to_bytes(16, byteorder='big')

print(
    "Avg # of 1's in {:,} 128-bit std random numbers: {:.3f}; "
    "time = {:.3f} ms".format(iter, ones / iter, 1000 * t)
)
```

Listing 6.3 An illustration of the efficiency of CSPRNG.

□

Chapter 7
Public-Key Cryptography

"Hi, my name is Matthew, although you may know me by another name. My friends call me Matty. And I should be dead."

—Matthew Perry, "Friends, Lovers, and the Big Terrible Thing"

The advent of cryptographically relevant quantum computers would break the most pervasive trapdoor functions (e.g., prime factorization, discrete logarithms) underpinning modern public-key cryptography (PKC). This includes all algorithms covered in this chapter. Consequently, Part III explores quantum-resistant alternatives, specifically Kyber and Dilithium.

This chapter explores the versatility of the Rivest–Shamir–Adleman (RSA) algorithm for both asymmetric encryption and digital signatures.

7.1 RSA for Encryption

Exercise 7.1 This is a chosen ciphertext attack. You are given the RSA public key and one ciphertext:

- Public key: $PU = \{e = 5, n = 735\,984\,598\,043\}$.
- A ciphertext C which is the encryption of an unknown message M:
  $C \equiv M^e \pmod{n}, C = 650\,404\,468\,666$.

You may query a decryption oracle that returns the plaintext of any chosen ciphertext C', except when $C' = C$ (i.e., it returns $\text{Dec}(C')$ for any C' you supply, except when $C' = C$).

Choose two appropriate odd values s_1 and s_2 and submit the ciphertexts $C_1 \equiv s_1^e \cdot C \pmod{n}$ and $C_2 \equiv s_2^e \cdot C \pmod{n}$. The oracle returns $\text{Dec}(C_1)$ and $\text{Dec}(C_2)$. Use either one of $\text{Dec}(C_1)$ and/or $\text{Dec}(C_2)$ to find the original message M.

Solution. Using `sympy` (or a modular calculator) we obtain: $PU = \{d = 441\,589\,636\,901, n = 735\,984\,598\,043\}$.

We pick $s_1 = 3, s_2 = 7$. So,

$C_1 \equiv s_1^e \cdot C \pmod{n} = 35 \times 650\,404\,468\,666 \pmod{735\,984\,598\,043} = 547\,581\,904\,636$, and $C_2 \equiv s_2^e \cdot C \pmod{n} = 75 \times 650\,404\,468\,666 \pmod{735\,984\,598\,043} =$

504 654 734 826.

Let's now use C_1 to find the value of M.

Let $Y = C_1^d \pmod{n} = 547\,581\,904\,636^{441\,589\,636\,901} \bmod n = 573\,036$.
$M = (Y \times 3^{-1}) \bmod n = (573\,036 \times 245\,328\,199\,348) \bmod 735\,984\,598\,043 = 191\,012$.

Had we chosen C_2 to find the value of M:

Let $Y = C_2^d \bmod n = 504\,654\,734\,826^{441\,589\,636\,901} \bmod n = 1\,337\,084$.
$M = (Y \times 7^{-1}) \bmod n = (1\,337\,084 \times 315\,421\,970\,590) \bmod 735\,984\,598\,043 = 191\,012$.

□

Exercise 7.2 Write a program that compares RSA key generation and encryption times against AES-OFB encryption time. This exercise requires serialization and proper handling of `pem`-encoded keys.

Solution. A sample solution is presented in Listing 7.1, which also includes 'TODO' challenges for deeper exploration.

```
"""
RSA vs AES Performance and Key Serialization Demo

Last modified: February 2026
Author: Original code by Miguel Vargas Martin, TODOs by ChatGPT

This program demonstrates:

1) RSA key generation and serialization (PEM format)
2) RSA encryption using OAEP padding
3) AES symmetric encryption (OFB mode)
4) Basic hashing (MD5 example)
5) (Commented) RSA signature example

The goal is to compare:
- Public-key vs symmetric-key performance
- Proper padding schemes
- Key storage formats

       Educational purposes only.
       MD5 is cryptographically broken and included for
    demonstration.
"""

# Key Serialization Utilities
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import
    load_pem_private_key
from cryptography.hazmat.backends import default_backend
```

```
def save_key(pk, filename, passw):
    """
    Serialize and store a private key in PEM format.

    Currently uses:
        - TraditionalOpenSSL format
        - No encryption (NoEncryption())

           The passw parameter is unused.
    TODO 1:
        Modify this function to encrypt the private key using:
            serialization.BestAvailableEncryption(passw)
        Explain why storing unencrypted private keys is dangerous.
    """
    pem = pk.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )

    with open(filename, 'wb') as pem_out:
        pem_out.write(pem)

def load_key(filename):
    """
    Load a PEM-encoded private key from disk.

    If the key is encrypted, a password must be supplied.
    """
    with open(filename, 'rb') as pem_in:
        pemlines = pem_in.read()

    private_key = load_pem_private_key(
        pemlines,
        password=None,
        backend=default_backend()
    )

    return private_key


# RSA Key Generation
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
import time, os

backend = default_backend()

# Measure RSA key generation time
t0 = time.time()
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=backend
```

```
)
t1 = time.time()

# Save and reload private key
save_key(private_key, 'private_key.pem', b'yetanotherpassword')
public_key = load_key('private_key.pem').public_key()
# (Equivalent to: private_key.public_key())

print("RSA key generation time:", t1 - t0, "seconds")

# TODO 2:
# Measure how key generation time changes for:
#   - 1024-bit
#   - 3072-bit
#   - 4096-bit
# Explain why key generation time increases.

# RSA Encryption (OAEP)
message = b'secret message' * 14

t2 = time.time()
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA224()),
        algorithm=hashes.SHA224(),
        label=None
    )
)
t3 = time.time()

print('RSA-OAEP:',
      'key gen:', t1 - t0, 'sec,',
      'enc:', t3 - t2, 'sec')

# OAEP provides semantic security (IND-CPA).
# TODO 3:
# Replace SHA224 with SHA256 in OAEP.
# Explain why OAEP is necessary instead of "raw RSA".

# AES Symmetric Encryption
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes

# 256-bit AES key (hardcoded for reproducibility)
k = b'abcdefghijklmnop' * 2

iv = os.urandom(16)     # 128-bit IV
mode = modes.OFB(iv)    # OFB = stream-like block mode

cipher1 = Cipher(algorithms.AES(k), mode, backend=backend)
encryptor1 = cipher1.encryptor()

t1 = time.time()
ct1 = encryptor1.update(message)
```

```
t2 = time.time()
encryptor1.finalize()

print('AES-OFB encryption time:', t2 - t1, 'sec')

# TODO 4:
# Compare RSA encryption time with AES encryption time.
# Explain why symmetric encryption is dramatically faster.
# What does this imply about hybrid encryption schemes?

# Hash Example (MD5)
digest = hashes.Hash(hashes.MD5(), backend=default_backend())
digest.update(b"abc")
digest.update(b"123")
dg = digest.finalize()

#        MD5 is collision-broken and should not be used for
    security.
# TODO 5:
# Replace MD5 with SHA256.
# Demonstrate how two different inputs can collide under MD5
# (research-based exercise).


# (Commented) RSA Signature Example
'''
signature = private_key.sign(
     message,
     padding.PSS(
         mgf=padding.MGF1(hashes.SHA256()),
         salt_length=padding.PSS.MAX_LENGTH
     ),
     hashes.SHA256()
 )
'''

# TODO 6:
# Uncomment the signature code and:
#   - Measure signing time
#   - Measure verification time
# Compare with encryption time.
# Why is signing typically slower than verification?
```

Listing 7.1 Performance comparison of RSA vs AES.

□

7.2 RSA for Digital Signatures

Exercise 7.3 Write a program that:

1. Generates a 2048-bit RSA key pair using the standard `public_exponent=65537`.

2. Saves the private key to disk and then extract the public key from the stored private key.
3. Encrypts a message using RSA-OAEP (Optimal Asymmetric Encryption Padding) with SHA224.
4. Uses RSA-PSS (Probabilistic Signature Scheme) to sign a message.
5. Verifies the RSA-PSS signature.

Solution. Listing 7.2 shows a simple program that demonstrates RSA for both encryption and digital signatures using the cryptography library. The program first generates a 2048-bit RSA key pair with the standard public exponent $e = 65537$. The private key is then serialized and stored on disk in PEM format, after which it is reloaded and the corresponding public key is extracted.

Next, a message is encrypted using RSA-OAEP with `SHA-224`, which provides randomized padding and protects against deterministic encryption attacks. The program also creates a digital signature using RSA-PSS with `SHA-256`, and finally verifies the signature using the public key to confirm the message's authenticity and integrity.

```
"""
RSA Encryption and Digital Signature Demo

Modified: February 2026
Author: Miguel Vargas Martin

This program demonstrates:

1) RSA key generation
2) Private key serialization (PEM format)
3) RSA encryption using OAEP padding
4) RSA digital signatures using PSS
5) Signature verification

       Educational demonstration only.
       Private key is stored without encryption (not secure in
    practice).
"""

# Key Serialization Utilities
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import
    load_pem_private_key

def save_key(pk, filename, passw):
    """
    Save a private RSA key to disk in PEM format.

    Current configuration:
        - TraditionalOpenSSL format
        - No encryption (NoEncryption())

           The 'passw' parameter is not used.
           Storing private keys without encryption is insecure.
    """
```

```
    pem = pk.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )

    with open(filename, 'wb') as pem_out:
        pem_out.write(pem)

def load_key(filename):
    """
    Load a private RSA key from a PEM file.
    """
    with open(filename, 'rb') as pem_in:
        pemlines = pem_in.read()

    private_key = load_pem_private_key(
        pemlines,
        password=None,
        backend=default_backend()
    )
    return private_key


# RSA Key Generation

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

backend = default_backend()

# Generate a 2048-bit RSA key pair
# public_exponent=65537 is the standard secure choice
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=backend
)

# Save private key to disk
save_key(private_key, 'private_key.pem', b'yetanotherpassword')

# Extract public key from stored private key
# (Equivalent to private_key.public_key())
public_key = load_key('private_key.pem').public_key()

# RSA Encryption (Confidentiality)

message = b'secret message' * 2

# Encrypt using RSA-OAEP
# OAEP provides semantic security (IND-CPA)
# SHA224 is used here for both MGF1 and hash function
```

```
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA224()),
        algorithm=hashes.SHA224(),  # Possible: SHA1, 224, 256,
    384, 512
        label=None
    )
)

# NOTE:
# - Raw RSA must never be used.
# - OAEP is required to prevent deterministic encryption attacks.

# RSA Digital Signature (Authenticity + Integrity)

signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

# PSS is the recommended modern signature padding.
# It provides provable security in the random oracle model.

# Signature Verification

# Recreate message (for clarity)
message = b'secret message' * 2

# Verify signature using the public key
# If verification fails, an exception is raised.
public_key.verify(
    signature,
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

# If no exception is raised, the signature is valid.

# Security Notes
"""
1) Encryption vs Signature:

   - Encryption uses the PUBLIC key.
   - Signing uses the PRIVATE key.
   - Verification uses the PUBLIC key.
```

```
2) OAEP vs PSS:

   - OAEP     used for encryption.
   - PSS      used for digital signatures.

3) Hash choices:

   - SHA256 is recommended.
   - SHA1 is deprecated.
   - SHA224 is acceptable but less common than SHA256.

4) Private Key Storage:

   - This example stores the private key without encryption.
   - In practice, always use:
        serialization.BestAvailableEncryption(password)

5) RSA is slow compared to symmetric encryption.
   In real systems, hybrid encryption is used:
       RSA encrypts an AES key,
       AES encrypts the data.
"""
```

Listing 7.2 RSA for encryption and signature.

Exercise 7.4 A number of RSA-PSS signatures have been generated for message `b'A heartfelt thank you to Galois and Fermat!'` using the same private key, with the following code snippet:

```
message = b'A heartfelt thank you to Galois and Fermat!'
signature = private_key.sign(
   message,
   padding.PSS(
      mgf=padding.MGF1(hashes.SHA256()),
      salt_length=padding.PSS.MAX_LENGTH
   ),
   hashes.SHA256()
)
```

The corresponding public key in `pem` format is as follows[1]:

```
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCpeGfrPlftac6CbipqMFde1YMm\nT8dPyosJ742KeZMeUqTDO5BsOV6Un
5Ab8tM5o6VEVumOsg4D6rTO6HhCpouzXchV\na38fOgKwQ5t9HblTCLM3C/3brOJ1mGGeOUQAAElK7NzoMlufJFKChCOg5VkuyjJX\n1iYn5aOcbVAXiAsQgwIDAQAB\n
-----END PUBLIC KEY-----\n'
```

To use the public key above you will need to serialize the `pem` bytes as follows:

[1] To optimize page space, a smaller font is used for the key (and signatures); I assume most readers will copy and paste the string rather than transcribe it manually.

```
public_key = serialization.load_pem_public_key(pem_bytes)
```

From the following 10 signatures in `Base64`, find the ones that successfully verify the message. (The key and signatures are also available as `exercise_7_4_data.py` in the book's GitHub repository.)

Signature 1: `PvfyQ0IXkl4TOqrHv+noDGV5X9MN0cxBScVJBFbBE30FSmkqP6dJ/1SxWdhE8xKdjy02lDLTZgHGGuhtPG0Rr73fEJlkqoUgs3uLfiIPCW`
`zriy6Ebuen9yQ7X4qlVQF8SfcROkW5KUgGe28lYT0StiUlQLdyQfbpJtU4spasQDQ=`

Signature 2: `gqp9EaLFVeBJRKO6+kUW5yiknsI1t2pL+FecagZWKxNOu6GgLnJTT8ntcbLfAY7db2iB74YK0eo6C0r2VRXBvbivQGMRRG5xSiFIYpKZ2Y`
`wLDOjYbCcb4ioRwyFijzbugLvNFUDbSGtA7DeZvlYiG8uZa9n0bxyQtcvHlUdHinY=`

Signature 3: `ZGjiYA9Ic4bl6ccxDtbGeGWh+Lv2xoVizxdPwi/7boKihbONU+g8kYBZOyP3vjr1xB/55BZuC8B9x2ONlyrcqS4U+7Za7G8ik8WY`
`GXn+sSqQugrM94Cbbw8HV/LQdDkkzWX3XmV0GNQ+ZQ5fAwmnMtpboT0Bod9WbObi90LRTQc=`

Signature 4: `kvPY+Q07oim687QqcP2erEOFa+eR9TWxRN4Aa0QjJkn+ZfmaFUtZGf13o4btdz+lHXTTigpFSC38s0IhH+tM5JzSLGl8NscOkQi0`
`BGnQz06tCK66fh7I2jafv4gE4HrqDDcrLhMtXbtjlGe3yESTSAWgCTDp4V6FTIDAyTCltpk=`

Signature 5: `a+gE2Rw4VgNWBbt2M1++9l6ilivB9utX6Bv6TPADAzZwR1mtQXwqXPX8Ov0fqKsO/w0ENcWwvz8Tplt9HtTBRw4ksEXrZTW/7f6hs`
`VmHdryyRB0iOozVSyn522UIU8CaN8PDGWzAWp7GKwiFuvz6G2OYwAkd92IfzgvMH6yQPrI=`

Signature 6: `ZC6COa99eLTchMESpki2mgtJx3/L/1yCAQyOlY3mkznB3+EM+bQgLhRy9q7Nky5221PHmmA80Zaeg4gz3b1qY8RqYNIY2eS8jeU`
`P4XcUervw4hmq2bIhrb/3YRGglujGldwmhwzENEcHRBOmUivJI5JFfoFW4QlfMSIVkiSNndw=`

Signature 7: `M9ll5wk2lsKtzwkqij+TJFO6cu8avIe0tx3iy10t45U9MQAA/ziog7qBFbmkgdFq2h6mKGyFCpCkEaPel9Mj8yvRoDTwVDKhWPae`
`Jh0dFW7EbK53LCFqSDiAr/4xSc2gsLX1OlVlF6Lf6TOAQgATznOXGMWr6nc78Y1ret0TF00=`

Signature 8: `ZLMF5sM2RaraKnRbU2XNoaOKXqhimaPSsWbqZhMpk4KnOXQhCbRlBWE2WgSMr1OxJDVnfszLGVLy/N+PhprVuD9XTqX0VMNdcK3Y`
`Ysx4F/MB0jNR2NSB0XYGim0A4zCJIqvDk6EOOoBvf5ZnKEsUh9bdCJhdTqmP8uA6RIke7WQ=`

Signature 9: `Aes4f/Rm/WXlv9WVSwf+nlRYi+TC/bTfKy0NXOXBtdXEiQtN8JHHpRWJ1iHtAa3D4v907rmHy+vdTgeBe3EItEJT5WyrKDTVTgKS`
`+EA+2OPFtkuJcbzwOvnd3FpV47wGFFR1uOPmR6RzZ5+ZySMDZ21+6eGALWMfNjJxEwJYj/8=`

Signature 10: `NNB43VvC3MW8qwRxdA1NR+556OSV4lU5PBNnKlTU6kuSSUTOy3GUHqU60M24zehWWnEjwnq4XcfTudmwkPwm7TF0C11cfy1eAsUu`
`WeWRjeC/DtU8byPoRj7M1bHSAunSXLPMvrTKVvb8gCbbcSudJ/ZYrTow3l14LUUgUjFuvwk=`

Solution. Listing 7.3 shows a snippet of a possible solution along with a list of valid signatures and their first few bytes for reference.

```
signature = base64.b64decode('NNB43...=')
public_key = serialization.load_pem_public_key(pem_bytes)
public_key.verify(
    signature,
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
#Answer: 3(ZGji),4(kvPY+),8(ZLMF5),9(Aes4f),10(NNB43)
```

Listing 7.3 Sketch of a possible solution.

□

Part III
Post-Quantum Cryptography

Chapter 8
More Number Theory

> *"But know this, that if the master of the house had known what hour the thief would come, he would have watched and not allowed his house to be broken into."*
>
> —Matthew 24:43

Cryptographic ciphers aren't usually designed with an expiration date, but they all carry an implicit one. This lifespan is a function of computational power; as hardware becomes more efficient, the cost of breaking a cipher—measured in time, money, and effort—eventually drops below the value of the information it protects.

In 1994, Peter W. Shor[1] introduced polynomial-time quantum algorithms capable of solving the prime factorization and discrete logarithm problems (cf. Section 3.6). While "Q-Day", the point at which a quantum computer can reliably execute these algorithms, remains a moving target, the threat of "harvest now, decrypt later" attacks has made the transition urgent. Consequently, the global migration to Post-Quantum Cryptography (PQC) is already underway. NIST finalized its first three standards (FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM), based on the CRYSTALS-Kyber algorithm, FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA), based on the Dilithium algorithm, and FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA), based on the SPHINCS$^+$ algorithm) in August 2024, with full organizational migration targets largely set for the early 2030s.[2]

A primary hurdle in building a *cryptographically relevant quantum computer* (CRQC) is achieving fault-tolerance (i.e., the ability to correct the inherent fragility of qubits). IBM's quantum roadmap projects delivering its first large-scale, fault-tolerant system, *Quantum Starling*, by 2029, featuring approximately 200 logical qubits (each encoded across many physical qubits: 288 physical qubits for every 12 logical ones in IBM's "gross" code).[3] Although breaking a 2048-bit RSA modulus currently requires an estimated 1 000+ logical qubits, the gap between hardware capability and cryptographic collapse is narrowing rapidly. For immediate experimentation, IBM al-

---

[1] Wikipedia Contributors. Peter Shor — Wikipedia, The Free Encyclopedia

[2] For more information on migration to PQC, frequently asked questions, etc., visit NIST's "Post-Quantum Cryptography", the "Canadian National Quantum-Readiness: Best Practices and Guidelines," and "The Commercial National Security Algorithm Suite 2.0 and Quantum Computing FAQ."

[3] The IBM Quantum roadmap (also watch their video): `https://www.ibm.com/quantum/hardware#roadmap`.

ready offers public cloud access to its quantum systems, ranging from a free tier for light research (10 minutes/month) to a pay-as-you-go model starting at USD$96 per minute.[4]

8.1 Basic Number Theory for PQC[5]

To grasp the mechanics of Kyber and Dilithium, one must first examine the *Decisional-Module Learning With Errors* (D-MLWE) problem. Given that the hardness of D-MLWE is derived from the *Module Learning With Errors* (MLWE) problem, this section begins with a description of MLWE. Before describing these problems, we need some extra number theory.

Definition 8.1 Let q be a prime and let $\mathbb{Z}_q$ denote the finite field of integers modulo q. The *polynomial ring* $\mathbb{Z}_q[x]$ is the set of all polynomials in x whose coefficients are taken from $\mathbb{Z}_q$, with the usual addition and multiplication of polynomials. For a fixed n that is a power of 2, we denote by $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ the ring obtained by reducing every polynomial in $\mathbb{Z}_q[x]$ modulo $x^n + 1$. Its elements are the polynomials of degree at most $n - 1$ with coefficients in $\mathbb{Z}_q$, and it is the ring used throughout Part III.

Remark 8.1 The polynomial $x^n + 1$ is irreducible over the integers when n is a power of 2, but it is *not* irreducible over $\mathbb{Z}_q$ for the moduli used in the standards: for $n = 256$ it factors into 128 quadratic polynomials modulo $q = 3329$ (ML-KEM) and into 256 linear polynomials modulo $q = 8\,380\,417$ (ML-DSA). Consequently R_q is a commutative ring but not a field: not every non-zero element has a multiplicative inverse. This factorization is not a weakness; it is precisely what makes the Number Theoretic Transform of Section 9.3.3 possible.

For reference, and in simple terms, in a finite field you can add, subtract, multiply, and divide by any non-zero element, and multiplication must also be commutative. In contrast, while in a ring you can always add, subtract, and multiply, you cannot always divide, and furthermore, the order of multiplication might not matter (in which case, it's a *commutative ring*). Reducing a polynomial modulus $x^n + 1$ means that if the maximum degree of a polynomial (e.g., after multiplying two polynomials) is $\geq n$, we reduce the polynomial by obtaining the remainder of dividing it by $x^n + 1$.

Example 8.1 Consider $R_{13} = \mathbb{Z}_{13}[x]/\langle x^2 + 2\rangle$. Let $f(x) = 5 + 2x$ and $g(x) = 4 + 3x$. $f(x)g(x) = 20 + 23x + 6x^2$; reducing the coefficients modulo 13 yields $7 + 10x + 6x^2$, and then reducing the polynomial modulo $x^2 + 2$, we get $8 + 10x$. Thus, the result of multiplying $f(x) = 5+2x$ and $g(x) = 4+3x$ in the polynomial ring $R_{13} = \mathbb{Z}_{13}[x]/\langle x^2+2\rangle$ is $8 + 10x$.

[4] Explore products & services: `https://www.ibm.com/quantum/products#access-plans`.

[5] For didactical courses on PQC, visit Alfred Menezes' (`https://en.wikipedia.org/wiki/Alfred_Menezes`) Cryptography101.ca web page, which includes excellent courses, including a YouTube course on Kyber and Dilithium.

Definition 8.2 For any integer a and a positive integer n, the *symmetric modulo* (mods) $r = a$ mods n is the unique integer such that $r \equiv a \bmod n$ and $-\frac{n}{2} < r \leq \frac{n}{2}$.

In simple terms, the symmetric modulo is the "closest to zero" version of a remainder. Or you can think of it this way: In standard math, if you are at 12 on a clock of 13, you say you are at "12." But in symmetric modulo, you'd say you are at "−1" because you are only one step away from the start; that is, pick the smallest number of steps away from 13. Listing 8.1 gives a programmatic definition of mods.

```
def mods(a, n):
    r = a % n
    if r > n / 2:
        r -= n
    elif r <= -n / 2:
        r += n
    return r
```

Listing 8.1 Symmetric modulo.

Example 8.2 7 mods 13 = −6; 6 mods 13 = 6; 7 mods 14 = 7; 8 mods 14 = −6; 15 mods 13 = 2; 25 mods 13 = −1.

Definition 8.3 The *infinity norm* of integer $r \in \mathbb{Z}_q$, denoted $\| r \|_\infty$, is $\| r \|_\infty = |r \text{ mods } q|$.

Example 8.3 In $\mathbb{Z}_{13}$: $\| 7 \|_\infty = \| 6 \|_\infty = 6$.

Moving into polynomial rings $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. The infinity norm of a polynomial is the coefficient with the maximum absolute value when all coefficients have been reduced via the symmetric modulus. That is, let $f(x) = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ in the ring R_q, $\| f \|_\infty = \max_i(|c_i \text{ mods } q|)$.

Example 8.4 In R_{13}, $\| 3x + 4 \|_\infty = \max(|3|, |4|) = 4$.

Now we turn our attention to polynomials in R_q with "small" infinity norm, and we will say that a polynomial is small if its infinite norm is $\leq \eta \ll q/2$. We will refer to the set containing all these small polynomials as S_η.

At this point, and before describing the MLWE problem, we need one more fact about polynomial rings: the product of two small polynomials is still "small-ish" relative to the modulus:

Claim Consider polynomials in $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. Let $f, g \in R_q$. The size of their product is bounded by $\| f \cdot g \|_\infty \leq n \cdot \| f \|_\infty \cdot \| g \|_\infty$.

And finally, the claim also holds for k-dimensional vectors of polynomials. That is, consider $R_q^k = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, for $\boldsymbol{f}, \boldsymbol{g} \in R_q^k$, $\| \boldsymbol{f} \cdot \boldsymbol{g} \|_\infty \leq k \cdot n \cdot \| \boldsymbol{f} \|_\infty \cdot \| \boldsymbol{g} \|_\infty$.

8.2 PQC One-Way Trapdoor Functions

The MLWE problem is a variant of the *Learning With Errors* (LWE) problem.

Definition 8.4 The *Learning With Errors (LWE) problem* in $\mathbb{Z}_q$ consists of solving a system of linear equations ($b = \boldsymbol{A} \cdot s$) that have been made noisy by adding a small error term (e) to each equation ($b = \boldsymbol{A} \cdot s + e$). It is computationally hard (i.e., there is no known polynomial-time algorithm) to find s given $\boldsymbol{A}$ and b.

Definition 8.5 The *Module Learning With Errors (MLWE) problem* (a.k.a. *Search-Module Learning With Errors (Search-MLWE) problem*) in the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ consists of solving a noisy system of linear equations ($\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$) where both the secret vector ($\mathbf{s}$) and the error term ($\mathbf{e}$) are sampled from a small-norm distribution. It is computationally hard to find $\mathbf{s}$ given $\mathbf{A}$ and $\mathbf{b}$.

A critical variant of MLWE is the Decision-MLWE (D-MLWE) problem, which challenges an attacker to distinguish between a valid MLWE sample and a uniformly random distribution. These two problems are computationally equivalent under the parameters used in Kyber, meaning an efficient solution for one would imply a solution for the other.

Definition 8.6 The *Decision-Module Learning With Errors (D-MLWE) problem* in the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ consists of distinguishing a noisy system of linear equations ($\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$) where $\mathbf{s}$ and $\mathbf{e}$ are small-norm terms, from a system where $\mathbf{b}$ is a vector of uniformly random polynomials. It is computationally hard to determine if $\mathbf{b}$ was generated using a short secret $\mathbf{s}$ or if it is merely uniform noise, given $\mathbf{A}$ and $\mathbf{b}$.

In the MLWE and D-MLWE problems, $\mathbf{A} \in R_q^{k \times k}$ is a matrix of k^2 uniformly random polynomials, while $\mathbf{s}, \mathbf{e} \in R_q^k$ are short vectors sampled from a specific distribution. The term "small" in Definitions 8.5 and 8.6 (as well as 8.4) specifically refers to a low infinity norm ($\| \cdot \|_\infty$). This "smallness trap" represents a critical security trade-off in Kyber: if the error $\mathbf{e}$ is too small (e.g. $\eta = 1$) the system becomes vulnerable to lattice reduction attacks;[6] however, if $\mathbf{e}$ is too large, the accumulated noise will exceed the threshold for correctness, leading to decryption failures. More of this to come in Chapter 9.

[6] This book doesn't cover lattices, but suffice it to say that the security of MLWE and D-MLWE is based on the hardness of certain lattice problems. Specifically, these lattice problems provide a lower bound of computational complexity, ensuring that finding the secret is infeasible for even the most powerful computers.

Chapter 9
ML-KEM Encapsulation

"Train yourself to let go of everything you fear to lose."

——Yoda, Star Wars: Episode III – Revenge of the Sith

As mentioned earlier, the Kyber algorithm is standardized in FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM). While Kyber and ML-KEM are often used interchangeably, it is important to distinguish them. ML-KEM is a key encapsulation mechanism that generates or wraps a 256-bit shared secret. Therefore, when referring to ML-KEM, "encapsulation" and "decapsulation" are more technically precise than "encryption" and "decryption."

A crucial element of ML-KEM is the "compression function," which uses modular rounding to discard least-significant bits, considerably reducing the size of the ciphertext.

Definition 9.1 The *closest integer round function*, denoted $\lceil x \rfloor$, is the closest integer to x, with ties broken "upwards" (i.e., $\lceil x \rfloor = \lfloor x + 1/2 \rfloor$).

Definition 9.2 The *modular rounding (a.k.a. compression) function*,[1] denoted $\text{Compress}_q(x, d)$ scales down integer $x \pmod q$ to an integer in the range $[0, 2^d - 1]$ as follows:

$$\text{Compress}_q(x, d) = \left\lceil \frac{2^d}{q} \cdot x \right\rfloor \pmod{2^d}.$$

Definition 9.3 The *inverse modular rounding (a.k.a. decompression) function*, denoted $\text{Decompress}_q(x, d)$ scales back up integer x as follows:

$$\text{Decompress}_q(x, d) = \left\lceil \frac{q}{2^d} \cdot x \right\rfloor.$$

Finally, it is helpful to understand the Centered Binomial Distribution (CBD) as ML-KEM picks the polynomials' η-small coefficients from a CBD.

To sample a coefficient c from CBD_η, take 2η random bits, divide them into two groups of η bits (a and b), and subtract the number of set bits in b from the number of set bits in a as follows:

[1] To an electrical engineer, this may sound like *quantization*.

$$c = \sum_{i=1}^{\eta} a_i - \sum_{i=1}^{\eta} b_i.$$

Listing 9.1 illustrates the sampling of 256 coefficients from a CBD_2, and adds a few TODO ideas for the curious reader.

```
"""
Last modified: February 2026

@author: Miguel Vargas Martin (adapted from ChatGPT iterations)

Secure implementation of Centered Binomial Distribution (CBD)
sampling using a cryptographically secure RNG.

Used in lattice-based cryptography schemes such as CRYSTALS-Kyber.
"""

import secrets

def cbd_sample(eta):
    """
    Sample a single integer coefficient from a Centered Binomial
    Distribution (CBD).

    X = sum_{i=1..eta}(a_i) - sum_{i=1..eta}(b_i)
    where a_i, b_i     {0,1} are independent uniform random bits.

    Output range: [-eta, +eta]
    """

    # Generate 2*eta bits at once (efficient + closer to real
    implementations)
    r = secrets.randbits(2 * eta)

    a_sum = 0
    b_sum = 0

    for i in range(eta):
        a_sum += (r >> i) & 1
        b_sum += (r >> (i + eta)) & 1

    return a_sum - b_sum

def sample_small_polynomial(n, eta):
    """
    Generate a polynomial with n coefficients sampled from CBD(eta)
    .
    """
    return [cbd_sample(eta) for _ in range(n)]

# Example usage (Kyber-768 uses eta = 2)
n = 256
```

```
eta = 2

small_poly = sample_small_polynomial(n, eta)

print(small_poly)
print(f"Max absolute value: {max(abs(c) for c in small_poly)}")

# TODO 1 (Basic):
# Empirically verify that coefficients are always in [-eta, +eta].
# Raise an exception if a value falls outside the expected range.

# TODO 2 (Statistics):
# Generate 100,000 samples from cbd_sample(eta) and:
#   - Plot the empirical distribution.
#   - Compare it with the theoretical CBD distribution.
#   - Is it symmetric? Is it centered at 0?

# TODO 3 (Theory):
# Derive the expected value and variance of CBD(eta).
# Verify empirically that:
#   E[X]     0
#   Var[X]     eta / 2

# TODO 4 (Security Engineering):
# Replace secrets.randbits with random.randint and:
#   - Measure performance difference.
#   - Discuss why random is insecure in cryptographic contexts.
#   - Explain how predictability could break a KEM scheme.

# TODO 5 (Cryptographic Context):
# Research how CRYSTALS-Kyber actually samples noise.
#   - Does it sample bits individually?
#   - How does it ensure constant-time behavior?
#   - Why is constant-time important here?

# TODO 6 (Advanced     Parameter Exploration):
# Experiment with eta = 1, 2, 3, 4.
#   - How does increasing eta affect:
#       * variance?
#       * coefficient spread?
#       * potential decryption failure probability?
```

Listing 9.1 Sampling of coefficients from a Centered Binomial Distribution.

9.1 Encapsulation

ML-KEM encapsulates a message m that is 256 bits (32 bytes) long, i.e., $m \in \{0,1\}^{256}$, into a ciphertext $(\mathbf{u}, v)$ using the recipient's public key $(\mathbf{A}, \mathbf{t})$. We will see later the public-key generation algorithm, but for now, let's assume the encryptor/sender (Bob) knows the recipient's (Alice's) public key.

Bob generates a random "encapsulation secret" vector $\mathbf{r} \in S^k_{\eta_1}$ and two small error terms: a vector $\mathbf{e_1} \in S^k_{\eta_2}$ and a scalar $e_2 \in S_{\eta_2}$. These are all sampled from a CBD (cf., Listing 9.1) using pseudorandom bytes generated by *Extendable Output Function* (XOF) SHAKE-256 (cf. Listing 5.4).

The next step is mapping message m into the polynomial ring R_q using the function $\text{Decompress}_q(m, d = 1)$. This scales each bit of m to either 0 or $\lceil q/2 \rfloor$, creating the "error tolerance gaps" necessary for the recipient (Alice) to successfully recover the message even in the presence of noise. For example, consider FIPS 203 modulus (cf. Table 9.1) $q = 3329$, each bit of m is converted according to the decompression rule:

$$\text{Decompress}_q(m_i, 1) = \begin{cases} 0, & \text{if } m_i = 0 \\ \left\lceil \frac{q}{2} \right\rfloor = 1665, & \text{if } m_i = 1 \end{cases}$$

At this point, the ciphertext pair $(\mathbf{u} \in R^k_q, v \in R_q)$ is generated as follows:

$$\begin{aligned} \mathbf{u} &= \mathbf{A}^T\mathbf{r} + \mathbf{e}_1 \\ v &= \mathbf{t}^T\mathbf{r} + e_2 + \text{Decompress}_q(m, 1) \end{aligned} \tag{9.1}$$

Finally, to reduce ciphertext size before sending it to Alice, $\mathbf{u}$ and v are passed through compression functions $\text{Compress}_q(\mathbf{u}, d_u)$ and $\text{Compress}_q(v, d_v)$.

9.2 Decapsulation

Alice performs decapsulation using her secret key $\mathbf{s}$ to strip away the noise and recover the original message. The first step for Alice is to decompress the received ciphertext $(\mathbf{u}, v)$: $\text{Decompress}_q(\mathbf{u}, d_u)$ and $\text{Decompress}_q(v, d_v)$. From here, Alice obtains a value y that is $\approx \text{Decompress}_q(m, 1)$ + noise, and finally performs $\text{Compress}_q(y, d = 1)$ to "most certainly" recover m.

It remains to see how the value y is obtained from the decompressed $(\mathbf{u}, v)$:

$$y = v - \mathbf{s}^T\mathbf{u}$$

Substituting $\mathbf{u}$ and v from the formulas in Equation 9.1, we obtain:

$$y = \left(\mathbf{t}^T\mathbf{r} + e_2 + \text{Decompress}_q(m, 1)\right) - \mathbf{s}^T\left(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1\right)$$

Since $\mathbf{t} \approx \mathbf{As}$, the terms $\mathbf{s}^T\mathbf{A}^T\mathbf{r}$ and $\mathbf{t}^T\mathbf{r}$ "mostly" cancel out, yielding[2]

$$y \approx \text{Decompress}_q(m, 1) + \text{noise}.$$

[2] The noise term collects every small quantity that does not cancel: noise $= \mathbf{e}^T\mathbf{r} - \mathbf{s}^T\mathbf{e}_1 + e_2 + \Delta v - \mathbf{s}^T\Delta\mathbf{u}$, where $\Delta\mathbf{u}$ and Δv are the rounding errors introduced by compressing $\mathbf{u}$ and v. Every term is small: e_2 and Δv directly, and the remaining three because they are products of small polynomials (cf. the Claim in Section 8.1).

The reader may be wondering why the words "most certainly" and "mostly" in the previous paragraphs. It is possible for the coefficients of the noise polynomials (cf. Equation 9.1) or the rounding errors from compression to be large enough that they "bleed" into the message bits. However, the odds of this occurring in ML-KEM are statistically negligible (e.g., $\approx 2^{-164}$ for ML-KEM-768).

While the mathematical steps above recover the message, ML-KEM adds a final layer of security known as the *Fujisaki-Okamoto (FO) transform*. After recovering the candidate message, Alice immediately re-encrypts it using her own public key $(\mathbf{A}, \mathbf{t})$. If the resulting ciphertext does not perfectly match the one she received, she assumes the ciphertext was tampered with, rejects the result, and returns a random value so that the attacker learns nothing. This re-encryption check ensures that even if an attacker tampered with the ciphertext (e.g., in a chosen-ciphertext attack), they cannot gain information about Alice's secret key.

9.3 Key Generation

ML-KEM's key generation process consists of creating a link between a public matrix of random polynomials ($\mathbf{A}$) and a secret "short" vector ($\mathbf{s}$). This process uses 32-byte random values, called *seeds*, which serve as the system's "blueprints" for its larger mathematical structures. So, instead of storing or transmitting entire vectors of polynomials, ML-KEM generates them on-the-fly from the seeds.

9.3.1 Seed generation and vector of polynomials A

The seed generation starts by sampling a random 32-byte seed d. The value d is passed through a hash function, such as SHA3-512 (cf. Exercise 5.7), to produce two separate values: a public $\rho \in \{0,1\}^{256}$ and a secret $\sigma \in \{0,1\}^{256}$. Then ρ is passed through XOF SHAKE-128, to "expand" it into a $k \times k$ matrix of polynomials, $\mathbf{A} \in R_q^{k \times k}$. Note that Bob (i.e., a user of Alice's public value $\mathbf{A}$) is able to derive $\mathbf{A}$ from ρ (which is much smaller than $\mathbf{A}$), making the transmission of $\mathbf{A}$ unnecessary.[3]

The values of the public matrix $\mathbf{A} \in R_q^{k \times k}$ are derived using $\mathbf{A} = \texttt{ExpandA}(\rho)$. Each entry $a_{i,j}$ is a polynomial with $n = 256$ coefficients, filled from the output stream of $\texttt{SHAKE-128}(\rho \,\|\, j \,\|\, i)$. In essence, `ExpandA` takes the next three bytes (24 bits) of the stream and splits them, with bit shifts and masks, into two 12-bit values; since $q < 2^{12}$, each value that is $\geq q$ is rejected, and the process continues with the next three bytes until

[3] For the curious reader, the size of $\mathbf{A}$ for, say $k = 3$ (cf. Table 9.1), is $k^2 = 9$ polynomials of degree 255 (i.e., 256 coefficients). Each coefficient of the polynomial is 12 bits ($\lceil \log_2 3329 \rceil$). Thus, the size of $\mathbf{A}$ is $9 \times 256 \times 12 = 27\,648$ bits (or 3 456 bytes). A huge difference compared to the 256 bits (32 bytes) of ρ.

all 256 coefficients are accepted. Because $q = 3329$ is well below $2^{12} = 4096$, about 19% of candidates are rejected, so each entry consumes about 473 bytes of SHAKE-128 output on average.[4] Compare this description with the one offered in Section 10.5.2 for ML-DSA.

9.3.2 Small vectors of polynomials s and e

As per the secret σ, this value is used as the seed to generate samples of two "short" vectors from a CBD: the secret key vector $\mathbf{s} \in S_{\eta_1}^k$, and the error vector $\mathbf{e} \in S_{\eta_1}^k$, where both have coefficients with a very low infinity norm (e.g., between -2 and 2, cf. Table 9.1).

9.3.3 Public key

At this point Alice can calculate the noisy linear equation to complete her public key $(\mathbf{A}, \mathbf{t})$:

$$\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} \pmod{q} \tag{9.2}$$

Alice can now share her public key as $(\mathbf{A}, \mathbf{t})$ or its much shorter form $(\rho, \mathbf{t})$. It is not hard to see that the most expensive computations in Equations 9.1, and 9.2 are $\mathbf{A}^T\mathbf{r}$ and $\mathbf{A}\mathbf{s}$, since multiplying two polynomials of degree $n = 256$ via standard convolution requires $O(n^2)$ operations. To accelerate these, ML-KEM performs multiplications in the Number Theoretic Transform (NTT) domain. While the transformation itself costs $O(n \log n)$, once in the NTT domain, polynomial multiplication is reduced to a simple point-wise product requiring only $O(n)$ multiplications.

9.4 ML-KEM Security Levels

Table 9.1 shows the three security levels specified in FIPS 203. The table also shows the equivalent strength compared with AES.

[4] FIPS 203 calls this step SampleNTT, and the coefficients it produces are interpreted directly in the NTT domain (Section 9.3.3).

Table 9.1 ML-KEM security levels and parameters in FIPS 203. Modulus $q = 3329$ and polynomial degree $n = 256$ are fixed across the three levels. Security levels represent a standardized measure of computational resistance against attacks, and in relation to AES as benchmark.

	Security	AES	Module	CBD noise		Compression	
Parameter set	level	equivalent	rank (k)	η_1	η_2	$d_{\mathbf{u}}$	d_v
ML-KEM-512	1	AES-128	2	3	2	10	4
ML-KEM-768	3	AES-192	3	2	2	10	4
ML-KEM-1024	5	AES-256	4	2	2	11	5

Example 9.1 The Java program of Listing 9.2 illustrates the generation of the polynomial seed ρ using the Bouncy Castle cryptographic API. Some compatibility comments are added to highlight API implementation changes as the library moved from `crypto.crystals.kyber` to `crypto.mlkem` to match the finalized FIPS 203 standard.

```
/*
 * Example: ML-KEM-768 (formerly Kyber-768) using Bouncy Castle
    1.83
 *
 * IMPORTANT:
 * Starting with Bouncy Castle 1.83, the original "Kyber" classes
 * were replaced with ML-KEM classes to match the finalized NIST
 * FIPS 203 standard.
 *
 * ML-KEM-768     Kyber-768 (k = 3).
 *
 * Author: Adapted by Miguel Vargas Martin from ChatGPT iterations
 * Last Modified: February 2026
 *
 * This example illustrates:
 *   1. Key generation
 *   2. Structure of the public key
 *   3. How the public seed (rho) is embedded
 */

import org.bouncycastle.pqc.crypto.mlkem.*;
import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.util.encoders.Hex;
import java.security.SecureRandom;

public class KyberMatrix {

    public static void main(String[] args) {
        /*
         * 1. Select Parameter Set
         *
         * ML-KEM defines three security levels:
         *   ml_kem_512   (k = 2)
         *   ml_kem_768   (k = 3)      recommended / balanced
         *   ml_kem_1024  (k = 4)
         *
         * For ML-KEM-768:
         *   - k = 3
```

```
         *   - Matrix A is 3    3
         *   - Each element is a polynomial of degree 255
         *   - Arithmetic is over Z_q[x]/(x^256 + 1)
         */
        MLKEMParameters params = MLKEMParameters.ml_kem_768;

        /*
         * 2. Key Generation
         *
         * Internally, the algorithm:
         *   - Generates 32 random bytes (seed d)
         *   - Expands d into:
         *         rho       public matrix seed
         *         sigma     noise seed
         *   - Uses rho to deterministically generate matrix A
         *   - Samples secret vector s and noise vector e
         *   - Computes:
         *         t = A  s + e
         * The public key is:
         *         (t, rho)
         * The private key contains:
         *         s and additional values for decapsulation
         */
        MLKEMKeyPairGenerator keyGen = new MLKEMKeyPairGenerator();
        keyGen.init(new MLKEMKeyGenerationParameters(
                new SecureRandom(), params));
        AsymmetricCipherKeyPair keyPair = keyGen.generateKeyPair();
        MLKEMPublicKeyParameters pubKey =
                (MLKEMPublicKeyParameters) keyPair.getPublic();
        MLKEMPrivateKeyParameters privKey =
                (MLKEMPrivateKeyParameters) keyPair.getPrivate();
        System.out.println("=== ML-KEM-768 Key Generation ===");

        /*
         * 3. Encoded Public Key Structure
         *
         * Bouncy Castle does NOT expose:
         *   - Matrix A
         *   - Vector t
         *   - Seed rho
         * Instead, it exposes a standardized byte encoding.
         * For ML-KEM-768, the public key encoding is:
         *     publicKey = t || rho
         * where:
         *     t   = compressed polynomial vector
         *     rho = 32-byte matrix seed
         */
        byte[] encodedPub = pubKey.getEncoded();
        System.out.println("Public key length: "
                + encodedPub.length + " bytes");

        /*
         * 4. Extracting the Matrix Seed (rho)
         *
         * In ML-KEM-768:
```

```
         *     rho is always the last 32 bytes
         *
         * rho is the seed that deterministically expands
         * (via SHAKE-128) into the full matrix A.
         *
         * Matrix A itself is never stored     it is generated
         * on-the-fly during encapsulation.
         */
        byte[] rho = new byte[32];
        System.arraycopy(
                encodedPub,
                encodedPub.length - 32,
                rho,
                0,
                32
        );

        System.out.println("Matrix seed (rho): "
                + Hex.toHexString(rho));

        /*
         * 5. Private Key
         *
         * The private key encoding contains:
         *   - Secret vector s
         *   - Hash of the public key
         *   - Randomness for CCA security
         * All internal lattice values remain hidden,
         * enforcing safe usage through encapsulation/decapsulation
    .
         */
        byte[] encodedPriv = privKey.getEncoded();
        System.out.println("Private key length: "
                + encodedPriv.length + " bytes");

        /*
         * Conceptual Summary
         *
         * ML-KEM does NOT store matrix A explicitly.
         * Instead:
         *   A = Expand(rho)
         * This design:
         *       reduces public key size
         *       ensures deterministic reconstruction
         *       simplifies interoperability
         *       avoids accidental misuse
         *
         * Bouncy Castle intentionally hides internal
         * polynomials to prevent cryptographic misuse.
         */
    }
}
```

Listing 9.2 Illustration of Kyber key generation.

□

Example 9.2 This example illustrates the encapsulation/decapsulation in Bouncy Castle's implementation of ML-KEM. Listing 9.3 illustrates key generation, encapsulation, and decapsulation of a 256-bit AES key. The program measures times to demonstrate the efficiency of the ML-KEM mechanisms. For a compelling follow-up left for the curious reader, compare these results with the times obtained in Exercise 7.2. Try lowering the bar for RSA by comparing times between ML-KEM 1024 against 512- and 1024-bit RSA.

```
/*
 * Modified: February 2026
 *
 * Author: Adapted by Miguel Vargas Martin from ChatGPT iterations
 *
 * Kyber (ML-KEM) Benchmark Example
 *
 * This program measures:
 *   1. Key pair generation time
 *   2. Encapsulation time (key wrapping)
 *   3. Decapsulation time (key unwrapping)
 *
 * IMPORTANT:
 * Kyber is a Key Encapsulation Mechanism (KEM), not a traditional
 * public-key encryption scheme. Therefore, in Bouncy Castle it
 * only supports WRAP_MODE and UNWRAP_MODE.
 *
 * The program performs multiple iterations and reports
 * average execution times in milliseconds.
 */

import java.security.*;
import javax.crypto.*;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.pqc.jcajce.provider.BouncyCastlePQCProvider
    ;
import org.bouncycastle.pqc.jcajce.spec.KyberParameterSpec;

public class Kyber {

    // Number of benchmark repetitions
    // Increase this for more stable timing results
    private static final int ITERATIONS = 100;

    public static void main(String[] args) throws Exception {
        /*
         * 1. Register Security Providers
         * BouncyCastleProvider  -> classical crypto algorithms
         * BouncyCastlePQCProvider -> post-quantum algorithms (
    Kyber, Dilithium, etc.)
         *
```

```
         * "BCPQC" is the provider name used later when requesting
    instances.
         */
        Security.addProvider(new BouncyCastleProvider());
        Security.addProvider(new BouncyCastlePQCProvider());

        // Accumulators for total execution time (in nanoseconds)
        long totalKeyGen = 0;
        long totalEncap = 0;
        long totalDecap = 0;

        /*
         * Main benchmark loop.
         * Each iteration performs a complete KEM cycle:
         *   - Generate Kyber keypair
         *   - Encapsulate (wrap a symmetric key)
         *   - Decapsulate (unwrap it)
         */
        for (int i = 0; i < ITERATIONS; i++) {
            // 2. Key Pair Generation
            /*
             * KyberParameterSpec.kyber768 selects the security
    level.
             * Available parameter sets:
             *   kyber512  -> NIST Level 1
             *   kyber768  -> NIST Level 3
             *   kyber1024 -> NIST Level 5
             * kyber768 is a good balanced choice for benchmarking.
             */
            long startKeyGen = System.nanoTime();

            KeyPairGenerator kpg =
                    KeyPairGenerator.getInstance("Kyber", "BCPQC");

            kpg.initialize(KyberParameterSpec.kyber768, new
    SecureRandom());

            KeyPair kp = kpg.generateKeyPair();

            long endKeyGen = System.nanoTime();
            totalKeyGen += (endKeyGen - startKeyGen);

            // 3. Encapsulation (WRAP_MODE)
            /*
             * Kyber encapsulates a randomly generated shared
    secret.
             * In JCA abstraction, this is modeled as wrapping a
    symmetric key.
             *
             * We generate a 256-bit AES key to simulate hybrid
    encryption.
             */
            KeyGenerator aesGen = KeyGenerator.getInstance("AES");
            aesGen.init(256);
            SecretKey secretKey = aesGen.generateKey();
```

```
            Cipher wrapCipher =
                    Cipher.getInstance("Kyber", "BCPQC");
            long startEncap = System.nanoTime();

            /*
             * WRAP_MODE triggers Kyber encapsulation.
             * Internally:
             *   - A shared secret is derived
             *   - A ciphertext (encapsulation) is produced
             */
            wrapCipher.init(Cipher.WRAP_MODE, kp.getPublic());

            byte[] wrappedKey = wrapCipher.wrap(secretKey);

            long endEncap = System.nanoTime();
            totalEncap += (endEncap - startEncap);

            // 4. Decapsulation (UNWRAP_MODE)
            /*
             * UNWRAP_MODE triggers Kyber decapsulation.
             * The private key reconstructs the shared secret
             * from the encapsulated ciphertext.
             */
            Cipher unwrapCipher =
                    Cipher.getInstance("Kyber", "BCPQC");
            long startDecap = System.nanoTime();
            unwrapCipher.init(Cipher.UNWRAP_MODE, kp.getPrivate());
            Key unwrappedKey =
                    unwrapCipher.unwrap(wrappedKey,
                                        "AES",
                                        Cipher.SECRET_KEY);
            long endDecap = System.nanoTime();
            totalDecap += (endDecap - startDecap);

            // 5. Correctness Verification
            /*
             * Ensure that encapsulation/decapsulation
             * correctly reconstructed the symmetric key.
             */
            if (!java.util.Arrays.equals(
                    secretKey.getEncoded(),
                    unwrappedKey.getEncoded())) {
                throw new RuntimeException("Decapsulation failed!")
;
            }
        }

        // 6. Print Average Timing Results
        System.out.println("Iterations: " + ITERATIONS);
        System.out.printf("Average Key Generation Time: %.3f ms%n",
                totalKeyGen / 1e6 / ITERATIONS);
        System.out.printf("Average Encapsulation Time: %.3f ms%n",
                totalEncap / 1e6 / ITERATIONS);
        System.out.printf("Average Decapsulation Time: %.3f ms%n",
```

```
                totalDecap / 1e6 / ITERATIONS);
    }
}
```

Listing 9.3 Illustration of ML-KEM key generation, encapsulation, and decapsulation.

□

Chapter 10
ML-DSA Signature Standard

"Our dilithium crystals represent awesome power. Wrongful use of that power, even to the extent of the taking of one life, would violate our history of total peace."

—Tharn, Star Trek: "Mirror, Mirror"

10.1 More Notation

As mentioned earlier, the Dilithium algorithm is standardized in FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA).

This section builds on top of Chapter 8. Recall that S_η denotes the set of polynomials in R_q with coefficients in the interval $[-\eta, \eta]$. Let $\tilde{S}_{\gamma_1}$ denote the set of polynomials in R_q with coefficients in the interval $(-\gamma_1, \gamma_1]$.

Now let B_τ be the set of polynomials in S_1 for which exactly τ of their coefficients are ± 1. ML-DSA uses a rejection threshold $\beta = \tau\eta$.

10.2 ML-DSA Decomposition Algorithms

Before describing ML-DSA signature and verification, we will study two powerful mechanisms that allow for compression of the public key and commitment vectors.

ML-DSA achieves compression of the public key vector $\mathbf{t}$ using a decomposition algorithm called `Power2Round` illustrated in Algorithm 4.

Algorithm 4 `Power2Round`(r, d)

Require: $r \in [0, q-1]$, $d \in [1, \log_2 q]$
Ensure: $(r_1, r_0) : r = r_1 2^d + r_0$ with $-2^{d-1} < r_0 \leq 2^{d-1}$ and $0 \leq r_1 \leq \lceil (q-1)/2^d \rceil$
1: $r_0 \leftarrow r \bmod 2^d$
2: $r_1 \leftarrow (r - r_0)/2^d$
3: **return** (r_1, r_0)

The exercises below illustrate `Power2Round` from its basic scalar form to the use of vectors of polynomials in R_q^k.

Exercise 10.1 Write a Python program that implements Algorithm 4.

Solution. Listing 10.1 implements `Power2Round` for a single integer. A small toy example is included to illustrate the computation and verify that the original value can be reconstructed correctly.

```
"""
Demo of power2round function for scalars

Last modified: November 2026
Author: Original code by Gemini and adapted by Miguel Vargas Martin
"""

import random

def power2round_scalar(r, d, q):
    """
    Standard Power2Round for a single integer.
    r_0 is the centered remainder: -2^(d-1) < r0 <= 2^(d-1)
    """
    r = r % q                       # ensure r is in [0, q-1]
    two_d = 1 << d
    half_two_d = 1 << (d - 1)

    # Standard centered modular reduction
    r0 = ((r + half_two_d - 1) % two_d) - half_two_d + 1
    r1 = (r - r0) // two_d
    return r1, r0

# --- Example using ML-DSA Parameters ---
#q = 8380417 # ML-DSA Q
#d = 13      # Standard d for ML-DSA

# --- Toy example ---
q = 23
d = 3

example = random.randint(0, q-1)

r1, r0 = power2round_scalar(example, d, q)

print(f"Original: {example}")
print(f"High bits (r1): {r1}")
print(f"Low bits (r0): {r0}")
print(f"Verification (r1 * 2^d + r0): {r1 * (2**d) + r0}")
```

Listing 10.1 Possible implementation of `Power2Round` from Algorithm 4.

□

Exercise 10.2 Extend your solution to Exercise 10.1 to polynomials in R_q.

Solution. Listing 10.2 offers a possible solution. Observe the use of `numpy` to handle array arithmetic.

```
"""
Demo of power2round function for polynomials in Rq

Last modified: March 2026
Author: Original code by Gemini and adapted by Miguel Vargas Martin
"""

import numpy as np

def power2round_poly_numpy(poly_coeffs, d, q):
    """
    Vectorized Power2Round for polynomials in Rq.
    poly_coeffs: a numpy array of coefficients
    """
    # Convert input to numpy array and ensure it is in [0, q-1]
    r = np.array(poly_coeffs) % q

    two_d = 1 << d
    half_two_d = 1 << (d - 1)

    # Vectorized centered modular reduction for r0:
    # -2^(d-1) < r0 <= 2^(d-1)
    r0 = ((r + half_two_d - 1) % two_d) - half_two_d + 1

    # Compute r1 based on r0
    r1 = (r - r0) // two_d

    return r1, r0

# --- ML-DSA Parameter Example ---
#q = 8380417
#d = 13
#n = 256 # Degree of polynomial in Rq

# --- Toy example ---
q = 23
d = 3
n = 5

# Create a random polynomial with n coefficients
coeffs = np.random.randint(0, q, n)

r1_poly, r0_poly = power2round_poly_numpy(coeffs, d, q)

# Verify that the decomposition is correct for the entire
    polynomial
# (r1 * 2^d + r0) should perfectly reconstruct the coefficients (
    mod q)
```

```
reconstructed = (r1_poly * (1 << d) + r0_poly) % q
is_correct = np.all(coeffs % q == reconstructed)

print(f"Full Polynomial Reconstruction Successful: {is_correct}")
```

Listing 10.2 Possible extension of Listing 10.1 to work with polynomials in R_q.

□

Exercise 10.3 Extend your solution to Exercise 10.2 to a vector of k polynomials in R_q^k.

Solution. Listing 10.3 offers a possible solution. Again, observe the seamless effort extending Listing 10.2 to work with vectors of polynomials by using `numpy`.

```
"""
Demo of power2round function for vector of polynomials in Rq^k

Last modified: November 2026
Author: Original code by Gemini and adapted by Miguel Vargas Martin
"""

import numpy as np

def power2round_vector_numpy(vector_coeffs, d, q):
    """
    Extends Power2Round to a vector of polynomials in Rq^k.
    vector_coeffs: numpy array of shape (k, n)
    """
    # Ensure input is a numpy array and reduced mod q
    r = np.array(vector_coeffs) % q
    two_d = 1 << d
    half_two_d = 1 << (d - 1)
    # The logic is identical; NumPy handles the (k, n) shape
    automatically
    r0 = ((r + half_two_d - 1) % two_d) - half_two_d + 1
    r1 = (r - r0) // two_d

    return r1, r0

# --- ML-DSA Parameter Example (e.g., ML-DSA-65/Dilithium3) ---
#q = 8380417
#d = 13
#n = 256
#k = 6    # Vector dimension for ML-DSA-65

# --- Toy example ---
q = 23
d = 3
n = 5
k = 2

# Create a random vector of k polynomials, each of degree n-1
# Shape: (6, 256)
```

```
vector_r = np.random.randint(0, q, size=(k, n))

# Process the entire vector at once
v1, v0 = power2round_vector_numpy(vector_r, d, q)

# --- Verification ---
# Reconstruct the vector
reconstructed = (v1 * (1 << d) + v0) % q

# Check if the entire matrix matches
all_match = np.array_equal(vector_r % q, reconstructed)

print(f"Vector Shape: {vector_r.shape}")
print(f"All {k*n} coefficients correctly decomposed: {all_match}")

# Check specific bounds for ML-DSA
two_d = 1 << d
half_two_d = 1 << (d - 1)
print(f"Max r0: {v0.max()} (Should be <= {half_two_d})")
print(f"Min r0: {v0.min()} (Should be > -{half_two_d})")
```

Listing 10.3 Possible extension of Listing 10.2 to work with a vector of k polynomials in R_q^k.

□

A second decomposition algorithm used by ML-DSA is called `Decompose`, which splits the coefficients of the commitment vector $\mathbf{w}$ into `HighBits` ($\mathbf{w_1}$) and `LowBits` ($\mathbf{w_0}$). The potential "bleeding" or carries between these bits during verification is managed by two auxiliary routines: `MakeHint`, which the signer uses to identify bit-flips, and `UseHint`, which the verifier uses to recover the original high bits. `Decompose` is illustrated in Algorithm 5.

Algorithm 5 `Decompose`(r, α)

Require: $r \in [0, q-1]$, α is an even integer such that $\alpha | (q-1)$
Ensure: (r_1, r_0) such that $r \equiv r_1\alpha + r_0 \pmod{q}$
1: $r_0 \leftarrow r \bmod \alpha$
2: **if** $r - r_0 = q - 1$ **then**
3: $\quad r_1 \leftarrow 0$
4: $\quad r_0 \leftarrow r_0 - 1$
5: **else**
6: $\quad r_1 \leftarrow (r - r_0)/\alpha$
7: **end if**
8: **return** (r_1, r_0)

Exercise 10.4 Write Algorithm 5 in Python, but use `NumPy`'s vectorization (cf. Exercise 10.3) to make it work with vectors of k polynomials in R_q^k.

Solution. Listing 10.4 offers a solution. Note the use of `NumPy`'s *Boolean masks* to replace the `if` statements in the algorithm.

```
"""
Implementation of Decompose for coefficients in Rq^k

Last modified: November 2026
Author: Original code by Gemini and adapted by Miguel Vargas Martin
"""

import numpy as np

def decompose_vector_numpy(vector_coeffs, alpha, q):
    """
    Extends Decompose to a vector of polynomials in Rq^k.
    vector_coeffs: numpy array of shape (k, n)
    """
    # 1. Ensure input is a numpy array and reduced mod q
    r = np.array(vector_coeffs) % q

    # 2. Initial remainder r0 in [0, alpha-1]
    r0 = r % alpha

    # 3. Center the remainder: -alpha/2 < r0 <= alpha/2
    mask_centered = r0 > (alpha // 2)
    r0[mask_centered] -= alpha

    # 4. Handle the overflow condition (r - r0 == q - 1)
    mask_overflow = (r - r0 == q - 1)

    # 5. Calculate r1
    # Initialize r1 with standard division
    r1 = (r - r0) // alpha

    # Apply special case logic for overflow coefficients
    r1[mask_overflow] = 0
    r0[mask_overflow] = r0[mask_overflow] - 1

    return r1.astype(np.int64), r0.astype(np.int64)

# --- ML-DSA Parameter Example (ML-DSA-65) ---
#q = 8380417
#k, n = 6, 256
#gamma2 = (q - 1) // 32
#alpha = 2 * gamma2

# Create a random vector of k polynomials, each of degree n-1
#vector_r = np.random.randint(0, q, size=(k, n))

# --- Toy example ---
q = 41
k, n = 2, 4
gamma2 = (q - 1) // 8
alpha = 2 * gamma2
```

```
vector_r = np.array([
    [3, 14, 24, 38],   # First polynomial
    [9, 21, 33, 5]   # Second polynomial
])

# Process the entire vector at once
v1, v0 = decompose_vector_numpy(vector_r, alpha, q)
is_handled = np.any((v1 == 0) & (vector_r % q > q - 1 - alpha // 2)
    )

# --- Verification ---
# Reconstruct the vector: r = r1 * alpha + r0 (mod q)
reconstructed = (v1 * alpha + v0) % q
all_match = np.array_equal(vector_r % q, reconstructed)

print(f"Vector Shape: {vector_r.shape}")
print(f"All {k*n} coefficients correctly decomposed: {all_match}")
print(f"Overflow cases handled: {is_handled}")
```

Listing 10.4 Possible implementation of `Decompose` from Algorithm 5.

□

In Listing 10.4, Boolean mask `mask_overflow` handles the case when adding small noise z to r (i.e., $r + z$) results in a bit overflow from low bits to high bits. ML-DSA signing keeps a record of these cases into a *hint* vector $\mathbf{h} \in R_q^k$ with coefficients $h_{i,j} \in \{0, 1\}$ using algorithm `MakeHint`. Then, the verifier uses the hint $\mathbf{h}$ to reconstruct the original high bits using algorithm `UseHint`. `MakeHint` and `UseHint` are described in Algorithms 6 and 7.

Algorithm 6 `MakeHint`(z, r, α)

Require: $r \in [0, q-1]$, $-\alpha/2 \leq z \leq \alpha/2$, α even
Ensure: $h \in \{0, 1\}$
1: $r_1 \leftarrow$ `HighBits`(r, α)
2: $v_1 \leftarrow$ `HighBits`$(r + z \pmod q, \alpha)$
3: **if** $r_1 \neq v_1$ **then**
4: **return** 1
5: **else**
6: **return** 0
7: **end if**

Algorithm 7 `UseHint`(h, r, α)

Require: $h \in \{0,1\}$, $r \in [0, q-1]$, α even, $m = (q-1)/\alpha$
Ensure: r_1' (The reconstructed high bits)
1: $(r_1, r_0) \leftarrow$ `Decompose`(r, α)
2: **if** $h = 0$ **then**
3: **return** r_1
4: **else if** $r_0 > 0$ **then**
5: **return** $(r_1 + 1) \pmod m$
6: **else**
7: **return** $(r_1 - 1) \pmod m$
8: **end if**

Exercise 10.5 This exercise and the next preview, with toy numbers, how ML-DSA uses `Decompose` and hints when signing and verifying (Sections 10.3 and 10.4). Two vectors of polynomials play a role:

- $\mathbf{w} \in R_q^k$ is Alice's *commitment*, computed when she signs (Section 10.3.1). Only its high bits $\mathbf{w}_1$ matter: they are what gets hashed into the signature.
- $\mathbf{z} \in R_q^k$ is a small shift. Bob, the verifier, cannot recompute $\mathbf{w}$ exactly; he obtains $\mathbf{w} + \mathbf{z}$ instead. In ML-DSA this shift is $-c\mathbf{s}_2 + c\mathbf{t}_0$ (Sections 10.4.2 and 10.5.5), which Alice can compute because she knows $\mathbf{s}_2$ and $\mathbf{t}_0$. (This $\mathbf{z}$ is not the signature vector $\mathbf{z}$ of Section 10.3.3; we keep the letter used by Algorithm 6.)

Consider the following toy parameters:

- $q = 41$, $n = 4$, $k = 2$
- $\gamma_2 = (q-1)/8 = 5$, $\alpha = 2\gamma_2 = 10$, so there are $m = (q-1)/\alpha = 4$ windows
- $\mathbf{w} = \begin{pmatrix} 3 & 14 & 24 & 38 \\ 9 & 21 & 33 & 5 \end{pmatrix}$

Suppose every coefficient of the shift $\mathbf{z}$ is +2, so that Bob will see $\mathbf{w} + \mathbf{z} \pmod q$. Alice anticipates this: for each coefficient she applies `MakeHint` (Algorithm 6), $h_{i,j} =$ `MakeHint`$(z_{i,j}, w_{i,j}, \alpha)$, which returns 1 exactly when adding $z_{i,j}$ moves $w_{i,j}$ into a different high-bit window. Determine the resulting hint vector $\mathbf{h}$. Note that the high bits $(\mathbf{w}_1)$ of $\mathbf{w}$ are computed via $(\mathbf{w}_1, \mathbf{w}_0) = \text{Decompose}(\mathbf{w}, \alpha)$, resulting in:[1]

$$\mathbf{w}_1 = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 1 & 2 & 3 & 0 \end{pmatrix}$$

Solution. The `MakeHint` algorithm identifies coefficients where adding $\mathbf{z}$ causes a "carry" into a different high-bit window. With $\alpha = 10$, the windows are $[0, 5]$, $[6, 15]$, $[16, 25]$ and $[26, 35]$ (high bits 0 to 3), while $[36, 40]$ wraps around to high bits 0 by the overflow rule of `Decompose` (Algorithm 5). Adding +2 moves $14 \to 16$, $24 \to 26$ and $5 \to 7$ into the next window, so these three coefficients need a hint. The coefficient 38 becomes 40; both values lie in the wrap-around zone, so its high bits stay 0 and no

[1] The low bits are not relevant for the sake of this exercise, but the reader can verify them to be $\mathbf{w}_0 = \begin{pmatrix} 3 & 4 & 4 & -3 \\ -1 & 1 & 3 & 5 \end{pmatrix}$.

hint is needed. All other coefficients remain in their windows. The resulting sparse hint vector is:

$$\mathbf{h} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

□

Exercise 10.6 Continuing from Exercise 10.5 with toy parameters ($q = 41$, $\alpha = 10$), Bob receives the signature, which includes the hint vector $\mathbf{h}$, and recomputes the shifted commitment $\mathbf{w}' = \mathbf{w} + \mathbf{z} \pmod{41}$; he never sees $\mathbf{w}$ itself:

$$\mathbf{w}' = \begin{pmatrix} 5 & 16 & 26 & 40 \\ 11 & 23 & 35 & 7 \end{pmatrix} \tag{10.1}$$

Using the hint vector $\mathbf{h}$ obtained previously,

$$\mathbf{h} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

apply the `UseHint` logic (Algorithm 7) to reconstruct the signer's original high-bit vector $\mathbf{w}_1$.[2]

Solution. The verifier applies `Decompose` to each coefficient of $\mathbf{w}'$. Where a hint bit $h_{i,j} = 1$ exists, the high bits are adjusted to recover the signer's original $\mathbf{w}_1$; here $m = (q-1)/\alpha = 4$.

For $w'_{0,1} = 16$: Decompose$(16, 10)$ yields $(w'_1 = 2, w'_0 = -4)$. Since $h_{0,1} = 1$ and $w'_0 \leq 0$, we calculate $(w'_1 - 1) \bmod m = 1$, which matches the original high bits of $w_{0,1} = 14$.

For $w'_{0,2} = 26$: Decompose$(26, 10)$ yields $(3, -4)$, and the hint gives $(3-1) \bmod m = 2$, matching the original high bits of $w_{0,2} = 24$.

For $w'_{1,3} = 7$: Decompose$(7, 10)$ yields $(1, -3)$, and the hint gives $(1-1) \bmod m = 0$, matching the original high bits of $w_{1,3} = 5$.

For coefficients where $h_{i,j} = 0$, the verifier adopts the calculated w'_1. For instance, $w'_{0,3} = 40$ decomposes to $(0, -1)$ by the overflow rule, which matches the original high bits of $w_{0,3} = 38$. The reconstructed high-bit vector is:

$$\mathbf{w}_1 = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 1 & 2 & 3 & 0 \end{pmatrix}$$

□

[2] We will see later how being able to reconstruct the high bits results in a successful signature verification.

10.3 Signature Generation

Describing ML-DSA using a prose narrative, as was done for ML-KEM in Chapter 9, poses some challenges due to the sheer number of interdependent steps and the algorithm's iterative nature. Nonetheless, in the interest of honouring the style of this book, I appeal to the reader's indulgence to allow me this venture. The signature process can be broken down into four main steps: commitment generation, challenge derivation, potential signature calculation, and rejection sampling. We will see the details of the key generation in a later section, but for now, assume that the signer, Alice, has generated 1) her verification public key: $pk = (\rho \parallel \mathbf{t})$, and 2) her private signing key: $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2)$, where K is a secret 32-byte seed generated together with ρ (Section 10.5.1). Table 10.1 lists the parameters for the three security levels of ML-DSA.

Table 10.1 ML-DSA parameters in FIPS 204. The modulus $q = 8\,380\,417$, the polynomial degree $n = 256$ and the number of bits dropped from $\mathbf{t}$, $d = 13$, are common to the three parameter sets. Level 2 is defined by collision search on SHA-256; its strength lies between AES-128 and AES-192.

Parameter	Description	ML-DSA-44	ML-DSA-65	ML-DSA-87
	Security level (NIST category)	2	3	5
	AES equivalent	≥ AES-128	AES-192	AES-256
λ	Collision strength of $\tilde{c}$ (bits)	128	192	256
k	Dimension of vector $\mathbf{t}$	4	6	8
ℓ	Dimension of vector $\mathbf{s}_1$	4	5	7
η	Secret key range bound	2	4	2
τ	Number of ±1 coefficients in c	39	49	60
β	Rejection threshold ($\tau\eta$)	78	196	120
γ_1	Range of coefficients in $\mathbf{y}$	2^{17}	2^{19}	2^{19}
γ_2	Low-order rounding range	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
α	Window size, $2\gamma_2$	$(q-1)/44$	$(q-1)/16$	$(q-1)/16$
m	Number of windows, $(q-1)/\alpha$	44	16	16
ω	Max number of 1s in hint $\mathbf{h}$	80	55	75
	Public key (bytes)	1 312	1 952	2 592
	Private key (bytes)	2 560	4 032	4 896
	Signature (bytes)	2 420	3 309	4 627
	Expected signing attempts	4.25	5.1	3.85

Furthermore, prior to these four steps, Alice would have computed $\mathbf{A}$ using an expansion function $\mathbf{A} = \texttt{ExpandA}(\rho)$, and $\mu = \texttt{SHAKE-256}(tr \parallel M', 64)$, where M' is the message M with a message prefix.[3]

[3] The prefix consists of two bytes, a mode byte and the length of an optional context string, followed by that string. The context is chosen by the application, so that a signature made for one purpose cannot be reused for another. Listings 10.5 and 10.6 use no context, so Bouncy Castle simply prepends the two bytes `0x00 0x00` to the message.

10.3.1 Commitment generation

Unlike ML-KEM, which relies on a CBD (see Listing 9.1) to introduce noise, ML-DSA utilizes a uniform distribution for its masking vector $\mathbf{y}$. This choice is foundational to the algorithm's security; it allows the signer to "dilute" the secret key in a sea of uniform randomness, provided the resulting sum stays within the strictly defined boundaries of the rejection sampling step.

Thus, to generate the commitment $\mathbf{w}$, Alice picks a random vector $\mathbf{y} \in \tilde{S}^{\ell}_{\gamma_1}$ by expanding, with SHAKE-256, a seed derived from her secret K, the message digest μ and 32 fresh random bytes; she then computes the commitment using the public matrix $\mathbf{A}$ as $\mathbf{w} = \mathbf{A}\mathbf{y} \pmod{q}$.

10.3.2 Challenge Derivation

In this step, Alice, the signer, computes the high bits ($\mathbf{w}_1$) of the commitment vector $\mathbf{w}$ using the `Decompose` algorithm (see Algorithm 5). These high bits are then hashed alongside the message digest μ and the public key to derive the challenge polynomial c. This derivation occurs in two stages.

First, Alice generates a seed, $\tilde{c}$, by computing $\tilde{c} = \texttt{SHAKE-256}(\mu \,\|\, \mathbf{w}_1, 64)$; in general $\tilde{c}$ is $\lambda/4$ bytes long, that is, 32, 48 and 64 bytes for ML-DSA-44, ML-DSA-65 and ML-DSA-87. Second, this seed $\tilde{c}$ is passed to a function called `SampleInBall`, which utilizes `SHAKE-256` as an XOF to stretch $\tilde{c}$ into a continuous stream of randomness. This stream is used to deterministically select the positions and signs of exactly τ coefficients to be assigned values of ± 1, while the remaining $n - \tau$ coefficients in the challenge polynomial c are set to 0.

10.3.3 Potential signature calculation

With the challenge c in hand, Alice computes a potential signature vector $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. The purpose of $\mathbf{z}$ is to mask the secret key $\mathbf{s}_1$, ensuring that $\mathbf{z}$ appears as a set of random values within the permitted range.

10.3.4 Rejection sampling

This step is crucial as it makes sure the coefficients of $\mathbf{z}$ are not too large as to leak information about the secret $\mathbf{s}_1$, and that the low bits $\mathbf{w}_0$ of the shifted commitment $(\mathbf{w} - c\mathbf{s}_2)$ are small enough to ensure verification remains unambiguous. Specifically, Alice checks if $\| \mathbf{z} \|_{\infty} < \gamma_1 - \beta$ and if the low bits $\mathbf{w}_0$ from $\texttt{Decompose}(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$ are less than $\gamma_2 - \beta$. If either condition fails, the potential signature is "aborted" to prevent

security leaks or verification errors. Alice must then restart the entire process with a fresh random vector $\mathbf{y}$. If both conditions are met, the hint vector $\mathbf{h}$ is computed. In practice, ML-DSA requires an average of three to five iterations before a valid signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$ is successfully generated.[4]

10.4 Signature Verification

The verification process can be described in three steps: initial validation and derivations, challenge derivation, and verification. Recall that, Bob, the verifier, has knowledge of the following:

- The message M along with its signature produced by Alice, $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$.
- Alice's verification public key: $pk = (\rho \parallel \mathbf{t})$. (In fact, the public key includes $\mathbf{t}_1$, not $\mathbf{t}$, but this discussion is deferred to Section 10.5.5.)
- ML-DSA's public global parameters of Table 10.1.

10.4.1 Initial validation and derivations

As the reader may suspect, the first thing Bob verifies is that $\| \mathbf{z} \|_\infty < \gamma_1 - \beta$. If this condition is not met, the signature is rejected.

Bob proceeds to derive $\mathbf{A}$ using expansion function $\texttt{ExpandA}(\rho)$. He also derives $tr = \texttt{SHAKE-256}(pk, 64)$, and $\mu = \texttt{SHAKE-256}(tr \parallel M', 64)$.

10.4.2 Challenge derivation

Bob derives the challenge c using $c = \texttt{SampleInBall}(\tilde{c})$. Then he recovers the high bits of the commitment, $\mathbf{w}_1$, using the hint vector $\mathbf{h}$ as $\mathbf{w}_1 = \texttt{UseHint}(\mathbf{h}, \mathbf{Az} - c\mathbf{t}, 2\gamma_2)$.

To see why this recovers Alice's $\mathbf{w}_1$, substitute $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ and $\mathbf{t} = \mathbf{As}_1 + \mathbf{s}_2$:

$$\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} + c\mathbf{As}_1 - c\mathbf{As}_1 - c\mathbf{s}_2 = \mathbf{w} - c\mathbf{s}_2.$$

Bob thus obtains the commitment $\mathbf{w}$ shifted by the small term $c\mathbf{s}_2$, and the rejection sampling of Section 10.3.4 ensures that this shift does not alter the high bits, so both parties arrive at the same $\mathbf{w}_1$.

[4] FIPS 204 adds two further checks: the signature is also rejected if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or if $\mathbf{h}$ has more than ω ones.

10.4.3 Verification

What's left for Bob is to verify that $\tilde{c} = \texttt{SHAKE-256}(\mu \parallel \mathbf{w}_1)$. If this is true, the signature is accepted as valid. If not, the signature is rejected.

10.5 Key Generation

The key derivation process can be broken down into five steps: sampling seeds, expanding the public matrix, generating the secret vectors, computing the public vector, and compressing the public key.

10.5.1 Sampling seeds

Alice begins by sampling a random 32-byte master seed ξ from a uniform distribution. This master seed is then used to obtain three values $(\rho, s, K) = \texttt{SHAKE-256}(\xi, 128)$, where ρ is assigned the first 32 bytes, s is assigned the following 64 bytes, and the remaining 32 bytes are assigned to K.

10.5.2 Expanding the public matrix A

With ρ in hand, Alice derives the public matrix $\mathbf{A} \in R_q^{k \times \ell}$ using $\mathbf{A} = \texttt{ExpandA}(\rho)$. Each entry $a_{i,j}$ is a polynomial with $n = 256$ coefficients, filled from the output stream of $\texttt{SHAKE-128}(\rho \parallel j \parallel i)$. In essence, `ExpandA` applies bit shifts to the next three bytes (24 bits) of the stream and masks the result to 23 bits; since $q < 2^{23}$, values $\geq q$ are rejected and the next three bytes are used instead. This continues until all 256 coefficients are accepted. Because q is very close to 2^{23}, only about 0.1% of candidates are rejected, so each entry consumes about 769 bytes of `SHAKE-128` output on average.

10.5.3 Generating the secret vectors $(\mathbf{s_1}, \mathbf{s_2})$

ML-DSA derives $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k$ from seed s through an expansion function: $(\mathbf{s}_1, \mathbf{s}_2) = \texttt{ExpandS}(s)$. To derive the coefficients for polynomial i, `ExpandS` processes the byte stream of $\texttt{SHAKE-256}(s \parallel i)$ to extract 4-bit chunks, rejecting any values that fall outside the required range to ensure the resulting coefficients are uniformly distributed within $[-\eta, \eta]$.

10.5.4 Computing the public vector t

Alice then computes the public vector $\mathbf{t} \in R_q^k$ as $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. At this point Alice has her public key $pk = (\rho \parallel \mathbf{t})$, and she is able to compute a compact digest of the public key as $tr = \texttt{SHAKE-256}(pk, 64)$.

10.5.5 Compressing the public key t

Finally, Alice compresses $\mathbf{t}$ using function $(\mathbf{t}_1, \mathbf{t}_0) = \texttt{Power2Round}(\mathbf{t}, d)$, where only $\mathbf{t}_1$ is included in the actual public key. So, effectively, the public key is $pk = (\rho \parallel \mathbf{t}_1)$, and the private signing key is $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$. But how then will Bob be able to recover the high bits $\mathbf{w}_1$ of the commitment if he only has $\mathbf{t}_1$, not the whole value $\mathbf{t}$ (see Section 10.4.2)?

The answer is in the hint vector $\mathbf{h}$. Bob uses the public $\mathbf{t}_1$ to compute an approximate value of $\mathbf{A}\mathbf{z} - c\mathbf{t}$ as $\mathbf{w}' = \mathbf{A}\mathbf{z} - c(\mathbf{t}_1 \cdot 2^d)$. This value $\mathbf{w}'$ differs from $\mathbf{A}\mathbf{z} - c\mathbf{t}$ by $c\mathbf{t}_0$, the contribution of the missing low bits $\mathbf{t}_0$. To fix this, Bob applies the function `UseHint`, which uses the hint bits to determine if a "carry" occurred into the high bits during the signer's original calculation, allowing Bob to recover the exact high bits of Alice's $\mathbf{w}$, denoted as $\mathbf{w}_1$, without ever knowing $\mathbf{t}_0$.

10.6 One More One-Way Trapdoor Function: I-MSIS

While the MLWE problem ensures that the public key $(\mathbf{A}, \mathbf{t})$ does not leak the secret key, we require a different "hard problem" to ensure a signer cannot forge a signature. Specifically, we must prevent an adversary from finding a vector $\mathbf{z}$ and a hint $\mathbf{h}$ that convincingly produce the high bits $(\mathbf{w}_1)$ required by the challenge.

In the lattice world, this difficulty is rooted in the hardness of finding short vectors in a collection of linear equations. There is no known polynomial-time solution to the following two problems:

Definition 10.1 Given a random matrix $\mathbf{A} \in R_q^{k \times \ell}$, find a non-zero vector $\mathbf{v} \in R_q^{k+\ell}$ such that:

$$[\mathbf{A} \mid \mathbf{I}_k]\mathbf{v} = \mathbf{0} \pmod{q},$$

where the infinity norm $\|\mathbf{v}\|_\infty$ is bounded by a small threshold $B \ll q$. This is known as the *Module Short Integer Solution (MSIS) problem*.

Definition 10.2 Given a random matrix $\mathbf{A} \in R_q^{k \times \ell}$ and a random target vector $\mathbf{b} \in R_q^k$, find a vector $\mathbf{v} \in R_q^{k+\ell}$ such that:

$$[\mathbf{A} \mid \mathbf{I}_k]\mathbf{v} = \mathbf{b} \pmod{q},$$

where $\|\mathbf{v}\|_\infty$ is bounded by a small threshold $B \ll q$. This is known as the *Inhomogeneous MSIS (I-MSIS) problem.*

In the context of ML-DSA, I-MSIS is the one-way wall that protects the signature from forgery. If an attacker could solve I-MSIS, they could "work backwards" from a challenge $\tilde{c}$ to find a short vector $\mathbf{z}$ that satisfies the verification relation:

$$\mathbf{Az} - \mathbf{w}_0 = c\mathbf{t} + 2\gamma_2\mathbf{w}_1$$

Recall that $(\mathbf{w}_1, \mathbf{w}_0) = \texttt{Decompose}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$. Therefore $\mathbf{Az} - c\mathbf{t} = 2\gamma_2\mathbf{w}_1 + \mathbf{w}_0$, which can be expressed as $\mathbf{Az} - \mathbf{w}_0 = c\mathbf{t} + 2\gamma_2\mathbf{w}_1$. So, the security of the scheme relies on the fact that while it is easy for Alice to compute a commitment $\mathbf{w}$ from a short vector $\mathbf{y}$, it is computationally infeasible for an attacker to find a short vector $\mathbf{z}$ that maps to a specific $\mathbf{w}_1$ and c without solving the I-MSIS problem.

10.7 ML-DSA Security Levels

Table 10.1 lists the parameters of the three security levels standardized in FIPS 204. ML-DSA-44 is the fastest and has the smallest keys and signatures; ML-DSA-87 offers the highest security level at roughly twice the key and signature size.

Exercise 10.7 Write a Java program using Bouncy Castle to sign the same document using the three ML-DSA security levels. Compare the time required for the generation of the public key as well as the signing process.

Solution. Listing 10.5 offers a solution. Note the loop of key generations at the beginning, a warm-up so that the time measured is fair across the three levels. In particular, we want to disregard the startup overhead that would otherwise fall on the first security level, while the later ones run on an already optimized JVM.

```
/*
* Example: ML-DSA levels comparison using Bouncy Castle 1.83
*
* Author: Adapted by Miguel Vargas Martin from ChatGPT iterations
* Last Modified: November 2026
*
* This example compares, for ML-DSA-44, ML-DSA-65 and ML-DSA-87:
*   1. Key generation time
*   2. Signing and verification time
*   3. Signature size
*/

package MLDSABenchmark;
import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
```

```
//import org.bouncycastle.pqc.jcajce.spec.MLDSAParameterSpec;

public class MLDSABenchmark {

    private static final byte[] MESSAGE =
            "This is a document that will be signed using ML-DSA.".
    getBytes();

    public static void main(String[] args) throws Exception {

        Security.addProvider(new BouncyCastleProvider());

        testMLDSA("ML-DSA-44", MLDSAParameterSpec.ml_dsa_44);
        testMLDSA("ML-DSA-65", MLDSAParameterSpec.ml_dsa_65);
        testMLDSA("ML-DSA-87", MLDSAParameterSpec.ml_dsa_87);
    }

    private static void testMLDSA(String name, MLDSAParameterSpec
    spec) throws Exception {

        System.out.println("----- " + name + " -----");

        KeyPairGenerator kpg = KeyPairGenerator.getInstance("MLDSA"
    , "BC");
        kpg.initialize(spec, new SecureRandom());

        int ITERATIONS = 10;

        for(int i=0;i<ITERATIONS;i++){
            kpg.generateKeyPair();
        }

        // Key generation
        long startKey = System.nanoTime();
        KeyPair kp = kpg.generateKeyPair();
        long endKey = System.nanoTime();
        double keyGenTime = (endKey - startKey) / 1e6;

        // Signing
        Signature signer = Signature.getInstance("MLDSA", "BC");
        signer.initSign(kp.getPrivate());

        long startSign = System.nanoTime();
        signer.update(MESSAGE);
        byte[] signature = signer.sign();
        long endSign = System.nanoTime();
        double signTime = (endSign - startSign) / 1e6;

        // Verification
        Signature verifier = Signature.getInstance("MLDSA", "BC");
        verifier.initVerify(kp.getPublic());

        long startVerify = System.nanoTime();
        verifier.update(MESSAGE);
```

```
        boolean valid = verifier.verify(signature);
        long endVerify = System.nanoTime();
        double verifyTime = (endVerify - startVerify) / 1e6;

        System.out.println("Public Key Generation Time: " +
    keyGenTime + " ms");
        System.out.println("Signing Time: " + signTime + " ms");
        System.out.println("Verification Time: " + verifyTime + "
    ms");
        System.out.println("Signature Size: " + signature.length +
    " bytes");
        System.out.println("Signature Valid: " + valid);
        System.out.println();
    }
}
```

Listing 10.5 Possible solution.

□

Exercise 10.8 Compare the times obtained from your solution to Exercise 10.7 with a 2048-bit RSA signature.

Solution. Listing 10.6 extends the timing experiment of Exercise 10.7 by comparing the three ML-DSA security levels against a 2048-bit RSA signature. The program measures key generation, signing, and verification times for the same message in each case, allowing the results to be compared directly. In addition, it reports the signature size, which is an important practical consideration. In general, RSA-2048 tends to produce much smaller signatures, while ML-DSA provides post-quantum security. The exact timings depend on the implementation, provider, and hardware, so the results should be interpreted as an empirical comparison rather than as a universal ranking. Note, too, that the listing uses Bouncy Castle for ML-DSA, but because Bouncy Castle is added at the end of the provider list, the RSA calls, which name no provider, run on Java's built-in SunRsaSign provider; the comparison therefore also compares two implementations. For a like-for-like comparison, name Bouncy Castle in all three RSA calls: `KeyPairGenerator.getInstance("RSA", "BC")` for key generation, and `Signature.getInstance("RSASSA-PSS", "BC")` for both the signer and the verifier.

```
/*
 * Exercise 10.8
 *
 * Compare the times obtained from Exercise 10.7 against
 * a 2048-bit RSA signature.
 *
 * Modified: November 2026
 * Author: ChatGPT and adapted by Miguel Vargas Martin
 *
 * This program benchmarks:
 * 1. ML-DSA-44
 * 2. ML-DSA-65
 * 3. ML-DSA-87
```

```
 * 4. RSA-2048 with RSA-PSS and SHA-256
 *
 * The same message is signed in all four cases.
 *
 * IMPORTANT:
 * - Times are implementation- and hardware-dependent.
 * - A short warm-up loop is used before timing so that
 *   JVM startup effects do not unfairly affect the first test.
 */

import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import java.security.spec.MGF1ParameterSpec;
import java.security.spec.PSSParameterSpec;

public class SignatureComparison {
    public static void main(String[] args) throws Exception {

        Security.addProvider(new BouncyCastleProvider());

        byte[] message =
            "This is a document that will be signed using ML-DSA
    and RSA."
            .getBytes();

        int ITERATIONS = 10;

        long startKey, endKey;
        long startSign, endSign;
        long startVerify, endVerify;

        double keyGenTime, signTime, verifyTime;

        KeyPairGenerator kpg;
        KeyPair kp;
        Signature signer;
        Signature verifier;
        byte[] signature;
        boolean valid;

        System.out.println("
    ======================================");
        System.out.println("ML-DSA-44");
        System.out.println("
    ======================================");

        kpg = KeyPairGenerator.getInstance("MLDSA", "BC");
        kpg.initialize(MLDSAParameterSpec.ml_dsa_44, new
    SecureRandom());

        for (int i = 0; i < ITERATIONS; i++) {
            kpg.generateKeyPair();
        }
```

```
        startKey = System.nanoTime();
        kp = kpg.generateKeyPair();
        endKey = System.nanoTime();
        keyGenTime = (endKey - startKey) / 1e6;

        signer = Signature.getInstance("MLDSA", "BC");
        signer.initSign(kp.getPrivate());
        startSign = System.nanoTime();
        signer.update(message);
        signature = signer.sign();
        endSign = System.nanoTime();
        signTime = (endSign - startSign) / 1e6;

        verifier = Signature.getInstance("MLDSA", "BC");
        verifier.initVerify(kp.getPublic());
        startVerify = System.nanoTime();
        verifier.update(message);
        valid = verifier.verify(signature);
        endVerify = System.nanoTime();
        verifyTime = (endVerify - startVerify) / 1e6;

        System.out.println("Public Key Generation Time: " +
    keyGenTime + " ms");
        System.out.println("Signing Time: " + signTime + " ms");
        System.out.println("Verification Time: " + verifyTime + "
    ms");
        System.out.println("Signature Size: " + signature.length +
    " bytes");
        System.out.println("Signature Valid: " + valid);
        System.out.println();

        System.out.println("
    ========================================");
        System.out.println("ML-DSA-65");
        System.out.println("
    ========================================");

        kpg = KeyPairGenerator.getInstance("MLDSA", "BC");
        kpg.initialize(MLDSAParameterSpec.ml_dsa_65, new
    SecureRandom());

        for (int i = 0; i < ITERATIONS; i++) {
            kpg.generateKeyPair();
        }

        startKey = System.nanoTime();
        kp = kpg.generateKeyPair();
        endKey = System.nanoTime();
        keyGenTime = (endKey - startKey) / 1e6;

        signer = Signature.getInstance("MLDSA", "BC");
        signer.initSign(kp.getPrivate());
        startSign = System.nanoTime();
        signer.update(message);
        signature = signer.sign();
```

```
        endSign = System.nanoTime();
        signTime = (endSign - startSign) / 1e6;

        verifier = Signature.getInstance("MLDSA", "BC");
        verifier.initVerify(kp.getPublic());
        startVerify = System.nanoTime();
        verifier.update(message);
        valid = verifier.verify(signature);
        endVerify = System.nanoTime();
        verifyTime = (endVerify - startVerify) / 1e6;

        System.out.println("Public Key Generation Time: " +
    keyGenTime + " ms");
        System.out.println("Signing Time: " + signTime + " ms");
        System.out.println("Verification Time: " + verifyTime + "
    ms");
        System.out.println("Signature Size: " + signature.length +
    " bytes");
        System.out.println("Signature Valid: " + valid);
        System.out.println();

        System.out.println("
    ======================================");
        System.out.println("ML-DSA-87");
        System.out.println("
    ======================================");

        kpg = KeyPairGenerator.getInstance("MLDSA", "BC");
        kpg.initialize(MLDSAParameterSpec.ml_dsa_87, new
    SecureRandom());

        for (int i = 0; i < ITERATIONS; i++) {
            kpg.generateKeyPair();
        }

        startKey = System.nanoTime();
        kp = kpg.generateKeyPair();
        endKey = System.nanoTime();
        keyGenTime = (endKey - startKey) / 1e6;

        signer = Signature.getInstance("MLDSA", "BC");
        signer.initSign(kp.getPrivate());
        startSign = System.nanoTime();
        signer.update(message);
        signature = signer.sign();
        endSign = System.nanoTime();
        signTime = (endSign - startSign) / 1e6;

        verifier = Signature.getInstance("MLDSA", "BC");
        verifier.initVerify(kp.getPublic());
        startVerify = System.nanoTime();
        verifier.update(message);
        valid = verifier.verify(signature);
        endVerify = System.nanoTime();
        verifyTime = (endVerify - startVerify) / 1e6;
```

```
        System.out.println("Public Key Generation Time: " +
    keyGenTime + " ms");
        System.out.println("Signing Time: " + signTime + " ms");
        System.out.println("Verification Time: " + verifyTime + "
    ms");
        System.out.println("Signature Size: " + signature.length +
    " bytes");
        System.out.println("Signature Valid: " + valid);
        System.out.println();

        System.out.println("
    ========================================");
        System.out.println("RSA-2048");
        System.out.println("
    ========================================");

        kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048, new SecureRandom());

        for (int i = 0; i < ITERATIONS; i++) {
            kpg.generateKeyPair();
        }

        startKey = System.nanoTime();
        kp = kpg.generateKeyPair();
        endKey = System.nanoTime();
        keyGenTime = (endKey - startKey) / 1e6;

        PSSParameterSpec pssSpec = new PSSParameterSpec(
            "SHA-256",
            "MGF1",
            MGF1ParameterSpec.SHA256,
            32,
            1
        );

        signer = Signature.getInstance("RSASSA-PSS");
        signer.setParameter(pssSpec);
        signer.initSign(kp.getPrivate());
        startSign = System.nanoTime();
        signer.update(message);
        signature = signer.sign();
        endSign = System.nanoTime();
        signTime = (endSign - startSign) / 1e6;

        verifier = Signature.getInstance("RSASSA-PSS");
        verifier.setParameter(pssSpec);
        verifier.initVerify(kp.getPublic());
        startVerify = System.nanoTime();
        verifier.update(message);
        valid = verifier.verify(signature);
        endVerify = System.nanoTime();
        verifyTime = (endVerify - startVerify) / 1e6;
```

```
        System.out.println("Public Key Generation Time: " +
    keyGenTime + " ms");
        System.out.println("Signing Time: " + signTime + " ms");
        System.out.println("Verification Time: " + verifyTime + "
    ms");
        System.out.println("Signature Size: " + signature.length +
    " bytes");
        System.out.println("Signature Valid: " + valid);
        System.out.println();
    }
}
```

Listing 10.6 Possible solution.

□

Part IV
Requiem

Chapter 11
Beyond

¿Cuix oc nelli nemohua in tlalticpac?
An nochipa tlalticpac: zan achica ye nican.
Tel ca chalchihuitl no xamani,
no teocuitlatl in tlapani,
no quetzalli poztequi.

"Do we truly live with roots in the earth?
Not forever on earth: only a little while here.
Even jade breaks,
even gold shatters,
even quetzal feathers tear apart."

—Nezahualcóyotl *(Nahuatl text after M. León-Portilla)*

At the time of writing, during a biting, long, and snowy winter in 2026 that has, if nothing else, driven me to my most active season of indoor tennis yet, Large Language Models (LLMs) have thoroughly permeated software development, including the sensitive domain of cryptographic code. This shift was the focus of a paper I presented last summer at the International Conference on Security and Cryptography in Bilbao, titled "Beyond Rules: How Large Language Models are Redefining Cryptographic Misuse Detection."[1]

Our research highlights a paradigm shift. While traditional static analysis tools such as CryptoGuard[2] rely on rigid rule-based detection mechanisms, LLMs demonstrate a superior ability to identify nuanced security flaws. We found that LLMs not only outperform these legacy tools in accuracy, but also provide more actionable guidance when faced with the complex and nonstandard code variations that often confuse traditional systems.

Naturally, LLMs, as paradigm shifters and accelerators of engineering progress, have prompted researchers across the globe to ask provocative questions. Some are framed with optimism, such as "Can AI outperform the state of the art for a given task?" Others reflect deeper concerns about the future of the field, for example "Is this the end of software developers?" or even broader questions about the fate of entire disciplines.

These concerns are not entirely new. Long before the current wave of AI systems, popular imagination had already explored scenarios in which machines acquired strategic autonomy with unsettling consequences. In the early 1980s, films such as *WarGames* and *The Terminator* captured public anxieties about computer systems that might escalate conflicts or act beyond human control. I remember watching these movies as a child.

[1] Z. Masood and M. Vargas Martin. "Beyond Rules: How Large Language Models Are Redefining Cryptographic Misuse Detection," 2025 International Conference on Security and Cryptography (SECRYPT), Bilbao, Spain, pp. 179–194, doi: 10.5220/0013524100003979.

[2] S. Rahaman, Y. Xiao et al. "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," 2019 ACM SIGSAC Conference on Computer and Communications Security, New York, USA, pp. 2455–2472, doi: 10.1145/3319535.3345659.

While they were intended as cautionary tales, they also sparked a lingering fascination with the technological future they imagined. Those fictional worlds felt distant at the time, yet they planted a quiet expectation that the coming decades would bring profound changes in how humans interact with machines.

Today, some of those imagined futures no longer feel quite so distant. In the narrative timeline of *The Terminator*, the year 2029 marks the height of a war between humans and machines. In reality, 2029 is now just around the corner. It will almost certainly not arrive with cybernetic soldiers resembling those fictional machines. Unfortunately, however, it is already arriving with something that echoes the same underlying concerns: the rapid emergence of AI-enabled warfare. Autonomous and semi-autonomous drone systems are increasingly capable of identifying, tracking, and engaging targets with limited human intervention. The technological landscape is evolving in ways that earlier generations might have recognized in spirit, even if the details differ from fiction.

At the same time, another transformative technology may soon reach practical relevance for security. Within the same time horizon, we may witness the emergence of a cryptographically relevant quantum computer, often referred to as a CRQC, capable of threatening widely deployed public-key cryptographic systems.

Taken together, these developments highlight an enduring reality: technological progress rarely unfolds exactly as imagined, yet its consequences can be just as profound. The machines of the future may not resemble those depicted in science fiction, but they will nevertheless challenge our assumptions about security, trust, and control.

For example, the theme of the 2026 New Security Paradigms Workshop (NSPW) is "The End of Security?" This deliberately provocative framing captures the tension between rapid advances in AI capabilities and the long-standing assumptions that underpin security practice. Whether the question is rhetorical or prophetic remains to be seen. What is certain, however, is that the practice of security, and cryptography in particular, will continue to evolve alongside the technologies it seeks to protect.

Looking across the history of cryptography reveals a recurring pattern: each generation of technology reshapes both the tools available to defenders and the capabilities available to adversaries. From mechanical ciphers to digital cryptosystems, from classical public-key infrastructures to emerging post-quantum designs, cryptography has continually adapted to new computational realities. The coming decades will likely be no different. Quantum computing may challenge long-standing assumptions about computational hardness, while artificial intelligence is already reshaping how vulnerabilities are discovered, analyzed, and mitigated.

Yet one element has remained constant throughout this evolution. Cryptography has never been solely about mathematics, algorithms, or machines. It is ultimately about trust between people operating in uncertain environments. Even as increasingly powerful computational systems assist us in designing protocols, analyzing implementations, and identifying weaknesses, the responsibility for understanding their limitations and ensuring their correct use remains a human one.

In this sense, the future of cryptography may not belong exclusively to new algorithms or faster machines, but to the practitioners who can thoughtfully combine them. The next generation of researchers and engineers will inherit tools of extraordinary capability: artificial intelligence systems that can reason about code and protocols, and potentially

quantum computers that redefine the boundaries of computation itself. Navigating this landscape will require not only technical mastery, but also careful judgment.

If history offers any guidance, cryptography will continue to evolve alongside the systems it protects. The questions may change, and the tools will certainly change, but the underlying challenge remains the same: how to build systems that deserve trust in a world where certainty is always provisional. In that respect, the story of cryptography is far from finished. It is merely entering its next chapter.

Index

acronyms, list of, xi
Advanced Encryption Standard, 29
ARC4, 41
avalanche effect, 29

birthday paradox, 51
Boolean masks, 106
Bouncy Castle API, 31

Centered Binomial Distribution, 87
ChaCha20, 41
chosen ciphertext attack, 71
cipher block chaining, 29, 41
cipher feedback, 29, 41
cipher modes of operation, 29
closest integer round function, 87
compression function, 87
congruency, 16
counter mode of operation, 29, 41
cryptographically relevant quantum computer, 83
CRYSTALS-Dilithium, 83
CRYSTALS-Kyber, 83

D-MLWE, 84
Data Encryption Standard, 5
Decision-Module Learning With Errors, 84, 86
Decompose, 105
decompression function, 87
Diffie-Hellman key exchange, 22
Digital Signature Algorithm, 57
Dilithium, 71, 83
discrete logarithm problem, 22
divisibility, 15
Division Algorithm, 16

electronic codebook, 29, 41
Elliptic Curve Cryptography, 22
Euclidean algorithm, 21
Extendable Output Function, 53, 90
Extended Euclidean algorithm, 22

Fujisaki-Okamoto transform, 91

Galois/Counter mode, 29, 41, 55
greatest common divisor, 20

hash function, 49
hint vector, 107

I-MSIS problem, 115
infinity norm, 85
Inhomogeneous MSIS problem, 115
inverse modular rounding function, 87

Kyber, 71, 83, 87

Learning With Errors Problem, 86
least common multiple, 21
least significant bit steganography, 8
Leonhard Euler, 19

MakeHint, 107
MD5, 49
message authentication codes, 49, 55
Miller-Rabin algorithm, 20
ML-DSA, 101
ML-KEM, 83, 87
ML-KEM decapsulation, 90
ML-KEM encapsulation, 89
ML-KEM security levels, 92
MLWE, 84
modular rounding function, 87
Module Learning With Errors, 84
Module Learning With Errors problem, 86
Module Short Integer Solution problem, 114
Module-Lattice-Based Digital Signature Standard, 83

Module-Lattice-Based Digital Signature Standard (ML-DSA), 101
Module-Lattice-Based Key-Encapsulation Mechanism Standard, 83, 87
MSIS problem, 114
multiplicative inverse, 21

network steganography, 9
Number Theoretic Transform, 92

one-way trapdoor function, 21
output feedback, 29, 41

Peter W. Shor, 83
Pierre de Fermat, 19
Pigpen cipher, 5
post-quantum cryptography, 83
Power2Round algorithm, 101
prime factorization problem, 22
public-key cryptography, 71

Q-Day, 83
quantum computer, 83

Radar cipher, 3
rejection sampling, 110
remainder, 16
RSA, 71
RSA-OAEP, 76
RSA-PSS, 76

SampleInBall, 111
Search-Module Learning With Errors problem, 86
SHA-3 Family, 53
steganography, 7
symbolic geometric ciphers, 3
symmetric modulo, 85

Triple DES, 5, 29

UseHint, 108

XTS-AES, 29, 41