Applied Information Security and Cryptography

Tiziano Bianchi
Politecnico di Torino

A.A. 2025/2026

Abstract

These notes provide a comprehensive summary of the course "Applied Information Security and Cryptography" held by Professor Tiziano Bianchi at Politecnico di Torino for the academic year 2025/2026. The course offers an in-depth exploration of the fundamental principles of information security and cryptography. Starting from the basic concepts and the historical context, the course delves into modern cryptographic techniques, including symmetric and asymmetric encryption, hashing algorithms, and digital signatures. Key topics include the mathematical foundations of cryptography, the analysis of security protocols, and the practical application of cryptographic methods to ensure confidentiality, integrity, and authenticity of information in modern communication systems. Particular attention is given to the security models used to formally define and verify the security of cryptographic systems.

Contents

1 The Basics

1.1 Why Security?

In our increasingly interconnected world, the need for information security is paramount. We rely on a multitude of digital services for our daily activities, from online banking and social media to e-commerce and cloud storage. These services handle a vast amount of sensitive data, which represent valuable assets that need to be protected. The digital environment in which these services operate is inherently hostile, with malicious actors constantly seeking to compromise the security of our data for personal gain. Therefore, it is crucial to implement robust security measures to safeguard our digital assets and ensure the trustworthiness of our online interactions.

1.2 The Problems of Security

The complexity of modern digital systems makes it challenging to ensure their security. Simple solutions are no longer sufficient to protect against the sophisticated attacks employed by malicious actors. The proliferation of connected devices, including PCs, tablets, smartphones, smart TVs, and IoT devices, has expanded the attack surface, creating more opportunities for vulnerabilities to be exploited. These devices communicate over a variety of networks, such as wired, mobile, Wi-Fi, and Bluetooth, each with its own security challenges. Furthermore, the widespread adoption of distributed services, like cloud storage and web services, introduces additional complexities in securing our data.

1.3 Computer Security Concepts

Computer security is built upon a set of fundamental concepts that define the goals of a secure system. These concepts are often represented by the CIA triad, which stands for Confidentiality, Integrity, and Availability.

1.3.1 Confidentiality

Confidentiality ensures that information is not disclosed to unauthorized individuals, entities, or processes. It is about protecting the privacy of our data and ensuring that only authorized users can access it. This can be achieved through various mechanisms, such as encryption, access control, and data masking.

1.3.2 Integrity

Integrity ensures that data is not altered or destroyed in an unauthorized manner. It is about maintaining the accuracy and completeness of our data and ensuring that it has not been tampered with. This can be achieved through mechanisms such as hashing, digital signatures, and version control.

1.3.3 Availability

Availability ensures that systems and data are accessible and usable upon demand by an authorized entity. It is about ensuring that our services are available when we need them and that we are not denied access to our data. This can be achieved through mechanisms such as redundancy, failover, and disaster recovery.

1.4 Additional Security Concepts

In addition to the CIA triad, there are other important security concepts that are essential for building secure systems. These include:

- **Authenticity:** Verifying that users are who they claim to be and that data originates from a trusted source.
- **Accountability:** The ability to trace actions to a specific entity, ensuring non-repudiation.

1.5 Security Attacks

Security attacks are actions that compromise the security of information. They can be broadly classified into two categories: passive attacks and active attacks.

1.5.1 Passive Attacks

Passive attacks involve monitoring and eavesdropping on communications without altering the data. The goal of a passive attacker is to obtain information without being detected. Examples of passive attacks include:

- **Eavesdropping:** Intercepting communications to obtain sensitive information.
- **Traffic analysis:** Analyzing communication patterns to infer information about the communicating parties.

Passive attacks are difficult to detect, but they can be prevented using techniques such as encryption.

1.5.2 Active Attacks

Active attacks involve modifying, fabricating, or interrupting communications. The goal of an active attacker is to disrupt the normal operation of a system or to gain unauthorized access to information. Examples of active attacks include:

- **Masquerade:** An attacker impersonates an authorized user to gain access to a system.
- **Replay:** An attacker intercepts and retransmits a valid data transmission to deceive the recipient.
- **Modification of content:** An attacker alters the content of a message to mislead the recipient.
- **Denial of service:** An attacker prevents legitimate users from accessing a service.

Active attacks can be detected using techniques such as authentication and integrity checks, but they are difficult to prevent.

1.6 Attack Surfaces

The attack surface of a system is the set of all possible points where an attacker can try to enter or extract data from the system. It can be divided into three main categories:

- **Network attack surface:** Vulnerabilities in the network infrastructure, such as open ports, weak protocols, and insecure network devices.
- **Software attack surface:** Vulnerabilities in the software applications, such as bugs, buffer overflows, and injection flaws.
- **Human attack surface:** Vulnerabilities related to human behavior, such as social engineering, phishing, and weak passwords.

1.7 Security Services and Mechanisms

To counter security attacks, we use a variety of security services and mechanisms. Security services are services that enhance the security of a system, while security mechanisms are the techniques and tools used to implement those services.

1.7.1 Security Services

The main security services are:

- **Authentication:** Ensures that a user is who they claim to be.
- **Access control:** Restricts access to resources to authorized users.
- **Data confidentiality:** Protects data from unauthorized disclosure.
- **Data integrity:** Ensures that data has not been altered or destroyed.
- **Non-repudiation:** Prevents a user from denying that they have performed a certain action.

1.7.2 Security Mechanisms

The main security mechanisms are:

- **Encipherment:** The process of converting plaintext into ciphertext to protect its confidentiality. It can be either symmetric (private key) or asymmetric (public key).
- **Data integrity mechanisms:** Techniques used to ensure the integrity of data, such as message digests.
- **Digital signatures:** A mechanism used to provide authentication, integrity, and non-repudiation.
- **Authentication exchange:** A protocol used to exchange authentication information between two parties.
- **Notarization:** The use of a trusted third party to verify the authenticity of a document or transaction.

1.8 Security Design Principles

To design secure systems, it is important to follow a set of established security design principles. These include:

- **Economy of mechanism:** Keep the design of the system as simple as possible.
- **Open design:** The security of a system should not depend on the secrecy of its design or implementation.
- **Psychological acceptability:** The security mechanisms should not be overly complex or burdensome for the users.

2 Introduction to Number Theory

2.1 Why Number Theory?

Modern cryptography is deeply rooted in number theory. While some mathematical problems are easy to solve using real numbers, they become significantly harder when solutions are constrained to be integers. Cryptography leverages this difficulty. Practical algorithms for encryption, decryption, and key exchange rely heavily on computations with integers.

Two core ideas illustrate this dependency:

- **Integer Solutions:** Consider the equation $243 \times x = 1$. If x can be a real number, the solution is trivial ($x = 1/243$). However, finding an integer solution for x in a modular arithmetic context is a fundamental problem in cryptography.
- **Hard Problems:** The security of many modern cryptographic systems is based on the computational difficulty of solving certain problems in number theory. A prime example is the integer factorization problem: given a large composite number N, it is computationally infeasible to find its prime factors p and q such that $N = pq$.

2.2 Divisibility

The concept of divisibility is fundamental to number theory.

Definition: We say that a non-zero integer b **divides** an integer a if there exists an integer m such that $a = mb$. This is also stated as "a is a multiple of b" or "b is a divisor of a". The notation $b|a$ is used to denote that b divides a.

2.2.1 Properties of Divisibility

The divisibility relation has several important properties:

- If $a|1$, then $a = \pm 1$.
- If $a|b$ and $b|a$, then $a = \pm b$.
- Any non-zero integer b divides 0.
- If $a|b$ and $b|c$, then $a|c$ (transitivity).
- If $b|g$ and $b|h$, then for any integers m and n, $b|(mg + nh)$.

2.3 The Division Algorithm

The division algorithm states that for any integer a (the dividend) and a positive integer n (the divisor), there exist unique integers q (the quotient) and r (the remainder) such that:

$$a = qn + r \quad \text{where} \quad 0 \leq r < n$$

The quotient q is the integer part of the division, which can be expressed as $q = \lfloor a/n \rfloor$.

2.4 Greatest Common Divisor (GCD)

The greatest common divisor of two integers is the largest integer that divides both of them.

Definition: The greatest common divisor of two integers a and b, denoted as $\gcd(a, b)$, is the largest positive integer k such that $k|a$ and $k|b$.

For convenience, we define $\gcd(0, 0) = 0$.

An integer c is the GCD of a and b if it satisfies two conditions:

1. c is a divisor of both a and b.
2. Any divisor of a and b is also a divisor of c.

Relatively Prime: Two integers a and b are said to be **relatively prime** if their greatest common divisor is 1, i.e., $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime because their only common positive divisor is 1.

2.5 Euclidean Algorithm

The Euclidean algorithm is a highly efficient method for computing the greatest common divisor (GCD) of two integers. It is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number. This can be extended to replacing the larger number with its remainder when divided by the smaller number.

The core identity of the algorithm is:

$$\gcd(a, b) = \gcd(b, a \pmod b)$$

The algorithm works by repeatedly applying this identity. Let's assume we want to compute $\gcd(a, b)$ where $a > b$:

1. We start with $a = q_1 b + r_1$, where $r_1 = a \pmod b$. The identity tells us that $\gcd(a, b) = \gcd(b, r_1)$.
2. We repeat the process: $b = q_2 r_1 + r_2$. Now we have $\gcd(b, r_1) = \gcd(r_1, r_2)$.
3. This process continues until the remainder is 0. The last non-zero remainder is the GCD of the original two numbers.

2.5.1 Example: GCD(710, 310)

We apply the division steps as follows:

$$\begin{aligned}
710 &= 2 \times 310 + 90 \\
310 &= 3 \times 90 + 40 \\
90 &= 2 \times 40 + 10 \\
40 &= 4 \times 10 + 0
\end{aligned}$$

The last non-zero remainder is 10. Therefore, $\gcd(710, 310) = 10$.

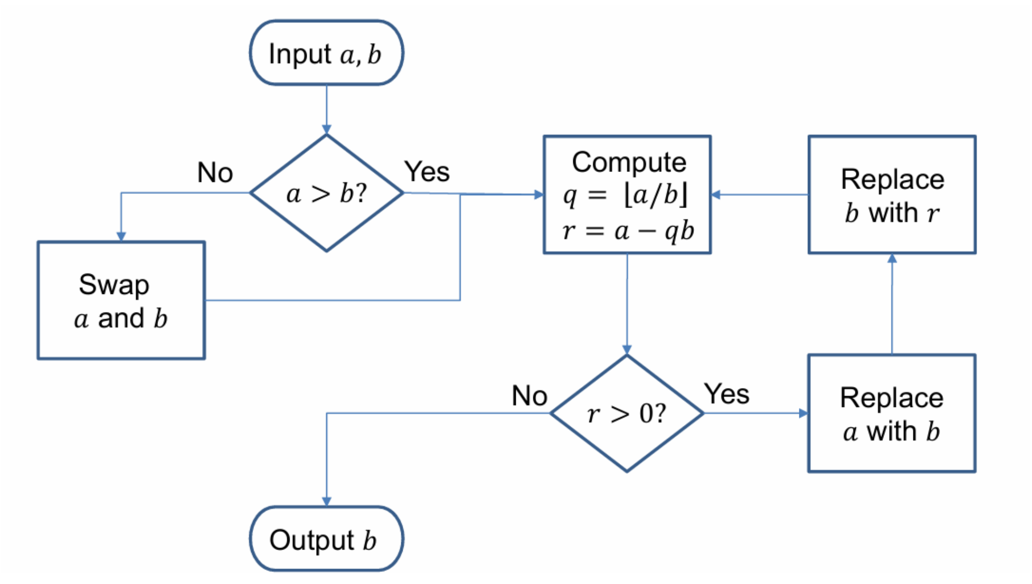


Figure 1: Euclidean algorithm flow-chart

2.6 Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" upon reaching a certain value—the **modulus**. It is a cornerstone of modern number theory and cryptography.

2.6.1 The Modulus and Congruence

The **modulus** operation finds the remainder of a division. If a is an integer and n is a positive integer, we define $a \pmod n$ to be the remainder r when a is divided by n. From the division algorithm, we have:

$$a = qn + r \quad \text{where} \quad 0 \leq r < n$$

This remainder r is the result of $a \pmod n$. For example, $11 \pmod 7 = 4$. For negative numbers, the remainder must be non-negative, so $-11 \pmod 7 = 3$ since $-11 = (-2) \times 7 + 3$.

Two integers a and b are said to be **congruent modulo n** if they have the same remainder when divided by n. This is written as:

$$a \equiv b \pmod n$$

This is equivalent to saying $(a \pmod n) = (b \pmod n)$.

An important property is that $a \equiv b \pmod n$ if and only if $n|(a-b)$. For example, $73 \equiv 4 \pmod{23}$ because $23|(73-4)$, since $69 = 3 \times 23$.

2.6.2 Properties of Congruences

Congruences have the following key properties:

1. $a \equiv b \pmod n$ if $n|(a-b)$.
2. $a \equiv b \pmod n$ implies $b \equiv a \pmod n$ (Symmetry).
3. $a \equiv b \pmod n$ and $b \equiv c \pmod n$ imply $a \equiv c \pmod n$ (Transitivity).

2.6.3 Properties of Modular Arithmetic

Modular arithmetic exhibits the following properties, which allow for consistent calculations:

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

These properties are extremely useful, as they allow us to reduce intermediate results modulo n at any step of a calculation, keeping the numbers small and manageable.

2.7 Residue Classes

The concept of congruence partitions the set of all integers $\mathbb{Z}$ into n distinct sets, known as **residue classes** modulo n.

The set of integers $\{0, 1, \ldots, n-1\}$ is called the **set of residues** modulo n, and is denoted by $\mathbb{Z}_n$.

$$\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$$

Each element $r \in \mathbb{Z}_n$ represents a residue class $[r]$, which contains all integers that have a remainder of r when divided by n.

$$[r] = \{a \in \mathbb{Z} \mid a \equiv r \pmod n\}$$

For example, the residue classes for modulus 4 are:

- $[0] = \{\ldots, -8, -4, 0, 4, 8, \ldots\}$
- $[1] = \{\ldots, -7, -3, 1, 5, 9, \ldots\}$
- $[2] = \{\ldots, -6, -2, 2, 6, 10, \ldots\}$
- $[3] = \{\ldots, -5, -1, 3, 7, 11, \ldots\}$

When performing modular arithmetic, we can replace any integer with any other integer from the same residue class without changing the result. Typically, we use the canonical representative, which is the smallest non-negative integer in the class (i.e., an element of $\mathbb{Z}_n$). This process is called **reducing modulo n**.

2.7.1 Inverses in Modular Arithmetic

Additive Inverse: For any integer $a \in \mathbb{Z}_n$, its additive inverse is an integer $b \in \mathbb{Z}_n$ such that $a + b \equiv 0 \pmod{n}$. The additive inverse is unique for each a and is typically denoted as $-a$. In $\mathbb{Z}_n$, the additive inverse of a (for $a \neq 0$) is $n - a$. This guarantees that subtraction is always possible.

Multiplicative Inverse: For an integer $a \in \mathbb{Z}_n$, its multiplicative inverse is an integer $a^{-1} \in \mathbb{Z}_n$ such that $a \times a^{-1} \equiv 1 \pmod{n}$. A multiplicative inverse for a modulo n exists **if and only if** $\gcd(a, n) = 1$.

For example, in $\mathbb{Z}_8$, the integer 3 has a multiplicative inverse because $\gcd(3, 8) = 1$. By testing, we find $3 \times 3 = 9 \equiv 1 \pmod{8}$, so $3^{-1} \equiv 3 \pmod{8}$. However, 2 has no multiplicative inverse modulo 8 because $\gcd(2, 8) = 2 \neq 1$. This implies that division by any number not relatively prime to the modulus is not well-defined.

2.8 Extended Euclidean Algorithm

The Extended Euclidean Algorithm is an extension of the Euclidean algorithm. In addition to finding the greatest common divisor (GCD) of two integers a and b, it also finds a pair of integers x and y that satisfy **Bézout's identity**:

$$ax + by = \gcd(a, b)$$

This algorithm is particularly important in cryptography for **finding multiplicative inverses in modular arithmetic**.

If $\gcd(a, b) = 1$, then the equation becomes $ax + by = 1$. Taking this equation modulo b, we get:

$$ax \equiv 1 \pmod{b}$$

This shows that x (**what we want to find**) is the **multiplicative inverse** of a modulo b. Similarly, y is the multiplicative inverse of b modulo a.

The algorithm extends the standard Euclidean process by keeping track of two auxiliary coefficients, x and y, for each remainder. Initially, we can express the two input values as:

$$a = 1 \cdot a + 0 \cdot b$$
$$b = 0 \cdot a + 1 \cdot b$$

At each step of the Euclidean algorithm, we perform the division:

$$r_{i-2} = q_i \, r_{i-1} + r_i$$

where q_i is the quotient and r_i is the remainder. For the extended version, we also express r_i as a linear combination of a and b:

$$r_i = x_i a + y_i b$$

The coefficients (x_i, y_i) are updated recursively using the following relations:

$$\begin{cases} x_i = x_{i-2} - q_i x_{i-1} \\ y_i = y_{i-2} - q_i y_{i-1} \end{cases}$$

This ensures that each remainder maintains the form $r_i = x_i a + y_i b$. The process continues until the remainder becomes zero; the last non-zero remainder is the GCD, and the corresponding (x, y) pair provides Bézout's coefficients.

2.8.1 Example: Finding the inverse of 550 mod 1759

We need to solve the equation $550y \equiv 1 \pmod{1759}$. We use the Extended Euclidean Algorithm with $a = 1759$ and $b = 550$. That is, we seek integers x and y such that:

$$1759x + 550y = \gcd(1759, 550).$$

For clarity we index remainders and coefficients starting from -1:

$$r_{-1} = a = 1759, \qquad r_0 = b = 550,$$

and compute the division

$$r_{i-2} = q_i\, r_{i-1} + r_i,$$

while keeping

$$r_i = x_i a + y_i b.$$

The recurrence for the coefficients is

$$\boxed{x_i = x_{i-2} - q_i x_{i-1}, \qquad y_i = y_{i-2} - q_i y_{i-1}}$$

with initial values $x_{-1} = 1,\ y_{-1} = 0,\ x_0 = 0,\ y_0 = 1$.

i	$\mathbf{q}_i$	$\mathbf{r}_i$	$\mathbf{x}_i$	$\mathbf{y}_i$	**Computation / comment**
-1	–	1759	1	0	initial: $r_{-1} = 1 \cdot a + 0 \cdot b$
0	–	550	0	1	initial: $r_0 = 0 \cdot a + 1 \cdot b$
1	3	109	1	-3	$x_1 = x_{-1} - 3x_0 = 1 - 3 \cdot 0 = 1$; $y_1 = y_{-1} - 3y_0 = 0 - 3 \cdot 1 = -3$.
2	5	5	-5	16	$x_2 = x_0 - 5x_1 = 0 - 5 \cdot 1 = -5$; $y_2 = y_0 - 5y_1 = 1 - 5 \cdot (-3) = 16$.
3	21	4	106	-339	$x_3 = x_1 - 21x_2 = 1 - 21 \cdot (-5) = 106$; $y_3 = y_1 - 21y_2 = -3 - 21 \cdot 16 = -339$.
4	1	1	-111	355	$x_4 = x_2 - 1x_3 = -5 - 1 \cdot 106 = -111$; $y_4 = y_2 - 1y_3 = 16 - 1 \cdot (-339) = 355$.
5	4	0	–	–	termination: $r_5 = 0$; last non-zero remainder $r_4 = 1$

Table 1: Extended Euclidean table with step index i (starting at -1).

The last non-zero remainder is $r_4 = 1$, hence $\gcd(1759, 550) = 1$. The corresponding Bézout coefficients are

$$x = x_4 = -111, \qquad y = y_4 = 355,$$

and indeed

$$1759(-111) + 550(355) = 1.$$

Taking the congruence modulo 1759 yields

$$550 \cdot 355 \equiv 1 \pmod{1759},$$

so the multiplicative inverse of 550 modulo 1759 is

$$\boxed{550^{-1} \equiv 355 \pmod{1759}.}$$

Remark. If you prefer a positive representative for x or y, reduce modulo the appropriate modulus; e.g. $x = -111 \equiv 1648 \pmod{1759}$.

2.9 Prime Numbers

A **prime number** is an integer greater than 1 that has no positive divisors other than 1 and itself. Integers greater than 1 that are not prime are called composite.

Prime numbers are central to number theory and cryptography. A key property is that for any prime p and any integer a such that $0 < a < p$, we have $\gcd(p, a) = 1$.

Fundamental Theorem of Arithmetic: Any integer greater than 1 can be uniquely factored into a product of prime numbers (up to the order of the factors).

$$x = p_1^{a_1} \cdot p_2^{a_2} \cdot \cdots \cdot p_t^{a_t}$$

where $p_1 < p_2 < \cdots < p_t$ are prime numbers and each a_i is a positive integer.

3 Some Abstract Algebra

Abstract algebra provides the formal language to describe the structures on which cryptographic systems are built. The main structures we are interested in are groups, rings, and fields.

3.1 Groups

A **group** is a set of elements, G, together with a binary operation $\circ$, that satisfies the following four axioms:

Closure: For all $a, b \in G$, the result of the operation, $a \circ b$, is also in G.

Associativity: For all $a, b, c \in G$, $(a \circ b) \circ c = a \circ (b \circ c)$.

Identity element: There exists an element $e \in G$, such that for every element $a \in G$, the equation $e \circ a = a \circ e = a$ holds.

Inverse element: For each element $a \in G$, there exists an element $a^{-1} \in G$, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

If a group is also commutative (i.e., $a \circ b = b \circ a$ for all $a, b \in G$), it is called an **abelian group**. The number of elements in a group is called its **order**. A group can be finite or infinite.

3.1.1 Cyclic Groups

A group G is called **cyclic** if every element in the group can be generated by repeatedly applying the group operation to a single element, called the **generator**. For an element $g \in G$, the set of all powers of g forms a cyclic subgroup. If this subgroup is the entire group G, then g is a generator of G. Cyclic groups are always abelian.

3.2 Rings

A **ring** is a set R equipped with two binary operations, typically addition $(+)$ and multiplication $(\times)$, satisfying the following axioms:

1. $(R, +)$ is an abelian group. The identity element for addition is denoted as 0.
2. Multiplication is associative: $(a \times b) \times c = a \times (b \times c)$ for all $a, b, c \in R$.
3. Multiplication is distributive over addition:
   - $a \times (b + c) = (a \times b) + (a \times c)$
   - $(b + c) \times a = (b \times a) + (c \times a)$

If multiplication is also commutative, the ring is called a **commutative ring**. An **integral domain** is a commutative ring with a multiplicative identity (1) and no zero divisors (if $a \times b = 0$, then $a = 0$ or $b = 0$).

3.3 Fields

A **field** is a set F with two operations, addition and multiplication, that satisfies the following:

1. $(F, +)$ is an abelian group (with identity 0).
2. $(F \setminus \{0\}, \times)$ is an abelian group (with identity 1).
3. Multiplication is distributive over addition.

In simpler terms, a field is a set where you can perform addition, subtraction, multiplication, and division (by non-zero elements) without leaving the set. Examples include the set of rational numbers ($\mathbb{Q}$), real numbers ($\mathbb{R}$), and complex numbers ($\mathbb{C}$). The set of integers ($\mathbb{Z}$) is not a field because not every element has a multiplicative inverse.

3.4 Finite Fields (Galois Fields)

In cryptography, we are particularly interested in fields with a finite number of elements. These are known as **finite fields** or **Galois Fields** (GF).

A key theorem in abstract algebra states that the order (number of elements) of a finite field must be a power of a prime number, p^n, where p is a prime and n is a positive integer. A finite field of order p^n is denoted as $\mathrm{GF}(p^n)$.

3.4.1 Finite Fields of the Form GF(p)

When $n = 1$, we get the field $\mathrm{GF}(p)$, which has p elements. This field is the set of integers $\{0, 1, \ldots, p-1\}$ with addition and multiplication performed modulo p. This set is denoted as $\mathbb{Z}_p$.

For $\mathbb{Z}_p$ to be a field, every non-zero element must have a multiplicative inverse. This is true if and only if p is a prime number. If p is prime, then for any integer a in $\{1, 2, \ldots, p-1\}$, we have $\gcd(a, p) = 1$. This guarantees the existence of a multiplicative inverse for a modulo p. The inverse can be found efficiently using the Extended Euclidean Algorithm.

For example, $\mathrm{GF}(7) = \mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ is a field. Every non-zero element has a multiplicative inverse modulo 7. For instance, $3 \times 5 = 15 \equiv 1 \pmod 7$, so $3^{-1} \equiv 5$. On the other hand, $\mathbb{Z}_8$ is not a field because not all non-zero elements have a multiplicative inverse (e.g., 2, 4, 6).

3.4.2 Finite Fields of the Form GF(2^n)

Finite fields of the form $\mathrm{GF}(p^n)$ with $p = 2$ are crucial in computer science and cryptography because their elements can be naturally represented by n-bit words. This avoids the inefficiency of mapping n-bit data into a field of prime order, which might leave some bit patterns unused or require more than n bits. For instance, mapping 8-bit bytes to GF(257) (a prime) would require 9 bits, while using GF(251) would leave the values from 251 to 255 unused.

The elements of $\mathrm{GF}(2^n)$ are not integers, but rather polynomials of degree strictly less than n with coefficients in GF(2) (i.e., coefficients are either 0 or 1). A polynomial $f(x)$ in this set has the form:

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

where each $a_i \in \{0, 1\}$. There are 2^n such polynomials, each corresponding to an n-bit string $(a_{n-1}a_{n-2}\ldots a_0)$.

Arithmetic in $\mathrm{GF}(2^n)$ is defined as follows:

- **Addition**: Polynomial addition is performed by adding the corresponding coefficients. In GF(2), addition is equivalent to the XOR operation. So, polynomial addition is a bitwise XOR of their corresponding bit strings.
- **Multiplication**: Polynomial multiplication is followed by a reduction modulo an **irreducible polynomial** of degree n over GF(2). An irreducible polynomial is a polynomial that cannot be factored into the product of two polynomials of lower degree (it is the polynomial equivalent of a prime number).

This structure forms a field. Addition is bitwise XOR, and multiplication is more complex but can be implemented efficiently in hardware and software (often using shifts and XORs). For example, the AES (Advanced Encryption Standard) algorithm heavily uses arithmetic in $\mathrm{GF}(2^8)$.

3.4.3 Example: Arithmetic in GF(2^3)

To construct $\mathrm{GF}(2^3)$, we need an irreducible polynomial of degree 3 over GF(2). An example is $p(x) = x^3 + x + 1$. The elements of the field are the $2^3 = 8$ polynomials of degree less than 3:

$$0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1$$

Let's add and multiply two elements, say $f(x) = x + 1$ and $g(x) = x^2 + 1$:

- **Addition**: $(x+1) + (x^2+1) = x^2 + (1+1)x + (1+1) = x^2$. In binary, this is $(011) \oplus (101) = (110)$.
- **Multiplication**: $(x+1)(x^2+1) = x^3 + x^2 + x + 1$. We must reduce this modulo $p(x) = x^3 + x + 1$. Since $x^3 \equiv x + 1 \pmod{x^3 + x + 1}$, we substitute:

  $$(x+1) + x^2 + x + 1 = x^2 + (1+1)x + (1+1) = x^2$$

  So, $(x+1)(x^2+1) \equiv x^2 \pmod{x^3 + x + 1}$.

4 Classical Encryption Techniques

This section explores historical ciphers. While not secure by modern standards, they are instructive as they introduce fundamental concepts of substitution and transposition, and their weaknesses highlight key principles of cryptography.

4.1 The Symmetric Cipher Model

A symmetric encryption scheme involves two parties (often called Alice and Bob) who share a secret key. This key is used for both encryption and decryption.

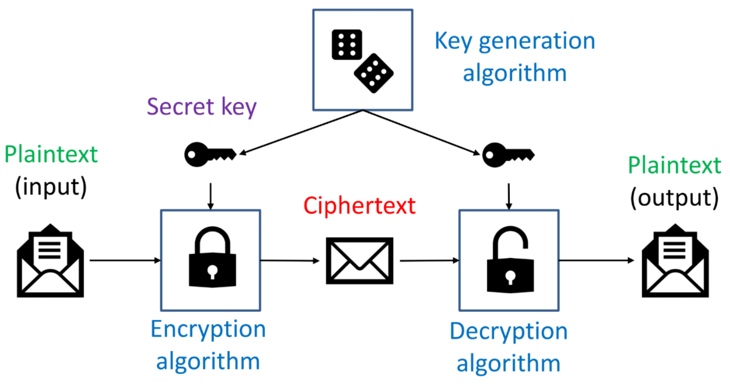


Figure 2: Symmetric cipher model.

The model consists of five components:

Plaintext (p or m): The original, intelligible message.

Secret Key (k): A piece of secret information shared by the communicating parties.

Encryption Algorithm (E): A procedure that transforms the plaintext into ciphertext using the secret key. The result is the ciphertext: $c = E(p, k)$.

Ciphertext (c): The scrambled, unintelligible message produced by the encryption algorithm.

Decryption Algorithm (D): A procedure that reverses the encryption, transforming the ciphertext back into the original plaintext using the same secret key: $p = D(c, k)$.

For the scheme to be correct, it must hold that for any key k and any plaintext p:

$$D(E(p, k), k) = p$$

The generation of the key itself is handled by a **Key Generation Algorithm (Gen)**, which is often a probabilistic algorithm that produces a random key based on some security parameters (e.g., key length).

4.2 Kerckhoffs' Principle

Proposed by Auguste Kerckhoffs in the 19th century, this principle is a cornerstone of modern cryptography. It states:

"The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience."

The modern interpretation is that the security of a cryptosystem should depend only on the secrecy of the key, not on the secrecy of the algorithm itself. This means that the encryption, decryption, and key generation algorithms (E, D, Gen) should be considered public knowledge.

This principle is still valid today for several reasons:

- It is much easier to keep a short key secret than to keep a complex algorithm secret.
- It is easier to change a compromised key than to replace an entire algorithm across all systems.
- Public algorithms can be scrutinized by the entire cryptographic community, leading to more robust and trustworthy systems. This avoids the fallacy of "security through obscurity".

4.3 Security of Encryption and Attack Scenarios

We can classify the security of an encryption scheme and the types of attacks an adversary might mount.

4.3.1 Levels of Security (Informal)

Unconditionally Secure: The ciphertext does not contain enough information to uniquely determine the corresponding plaintext, no matter how much computing power the adversary has. The One-Time Pad is the only known example.

Computationally Secure: Breaking the cipher would require an infeasible amount of computational resources (e.g., time, memory). Modern ciphers aim for this level of security.

An adversary can attempt two main types of attacks:

- **Brute-force attack:** Trying every possible key until a meaningful plaintext is found. This is defeated by having a sufficiently large key space.
- **Cryptanalysis:** Exploiting the properties of the algorithm or statistical patterns in the ciphertext to deduce the key or plaintext.

4.3.2 Attack Scenarios

The power of an adversary is often categorized by the type of information they have access to:

Ciphertext-Only Attack (COA): The adversary only has access to one or more ciphertexts.

Known-Plaintext Attack (KPA): The adversary has access to one or more plaintext/ciphertext pairs encrypted with the same key. This is a realistic scenario, for example, due to predictable message headers.

Chosen-Plaintext Attack (CPA): The adversary can choose arbitrary plaintexts and obtain their corresponding ciphertexts.

Chosen-Ciphertext Attack (CCA): The adversary can choose arbitrary ciphertexts and obtain their corresponding plaintexts. This is the strongest attack model.

A practical cryptosystem should be secure at least against KPA, and modern systems are often required to be secure against CPA and CCA.

4.4 Substitution Ciphers

In a substitution cipher, each character of the plaintext is replaced by another character. We will use the convention that plaintext is in lowercase and ciphertext is in UPPERCASE.

4.4.1 Caesar Cipher (Shift Cipher)

Julius Caesar used a simple cipher where each letter in the plaintext is shifted by a fixed number of places down the alphabet. The standard Caesar cipher uses a shift of 3.

- **Encryption:** $C = E(p, k) = (p + k) \pmod{26}$
- **Decryption:** $p = D(C, k) = (C - k) \pmod{26}$

For the original Caesar cipher, the key k is fixed at 3. A generalized version, the **Shift Cipher**, allows k to be any integer between 0 and 25.

Weakness: The key space is extremely small (only 26 possible keys). This makes it trivial to break with a brute-force attack.

4.4.2 Monoalphabetic Cipher

To counter the small key space of the shift cipher, a monoalphabetic cipher uses an arbitrary permutation of the alphabet as its key. Each letter in the plaintext is mapped to a unique letter in the ciphertext according to this permutation.

```
plain:  abcdefghijklmnopqrstuvwxyz
cipher: XEUADNBKVMROCQFSYHWGLZIJPT
```

Strength: The key space is massive. There are 26! (26 factorial) possible permutations, which is approximately 4×10^{26} or 2^{88}. This makes a brute-force attack computationally infeasible.

Weakness: This cipher preserves the underlying statistical properties of the plaintext language. Each plaintext letter is always mapped to the same ciphertext letter. This makes it vulnerable to **frequency analysis**. For example, in English, 'e' is the most frequent letter. By analyzing the frequency of letters in the ciphertext, an attacker can deduce the mapping. Similarly, patterns in digrams (e.g., 'th') and trigrams (e.g., 'the') can be used to break the cipher.

Lesson Learned: A large key space is necessary but not sufficient for security. The encryption must also hide the statistical properties of the plaintext.

4.4.3 Polyalphabetic Ciphers

To overcome the weakness of monoalphabetic ciphers, polyalphabetic ciphers use different substitution rules for each letter of the plaintext, effectively flattening the letter frequency distribution of the ciphertext.

The **Vigenère cipher** is a well-known example. It uses a keyword of length m. To encrypt a plaintext, the keyword is repeated, and each plaintext letter is shifted by the value of the corresponding key letter.

$$C_i = (p_i + k_{i \pmod m}) \pmod{26}$$

Weakness: The cipher is still periodic. If two identical plaintext segments are separated by a multiple of the key length m, they will be encrypted to identical ciphertext segments. An attacker can use this property (e.g., via Kasiski examination) to find the key length m. Once m is known, the ciphertext can be broken into m separate monoalphabetic ciphers, each of which can be solved by frequency analysis.

Lesson Learned: The encryption should not reveal information about the key, including its length or any periodic patterns.

4.5 One-Time Pad (Vernam Cipher)

The One-Time Pad (OTP) is a variant of the Vigenère cipher that achieves perfect, unconditional security. It requires the following strict conditions:

1. The key must be a truly random sequence.
2. The key must be at least as long as the message.
3. The key must be used only once (hence the name).

Encryption is typically done using bitwise XOR:

$$C_i = p_i \oplus k_i$$

Strength: The OTP is provably secure. Given a ciphertext, any plaintext of the same length is equally possible, because for any plaintext p^* and ciphertext c, there exists a key $k^* = c \oplus p^*$ that decrypts c to p^*. The ciphertext conveys no statistical information about the plaintext.

Weakness: The OTP is highly impractical for most applications. The need for a long, truly random key that must be securely shared in advance and can never be reused makes it very difficult to manage.

5 Security Models

The examples from classical cryptography demonstrate that designing a secure cryptosystem is not a trivial task. Ad-hoc solutions are often insecure, and even seemingly robust systems can have subtle flaws. To build secure systems, we need a formal and precise way to define what "secure" means.

5.1 What is Secure?

Before designing a cryptosystem, we need a formal security definition. For an encryption scheme, the core security property is **confidentiality** or **secrecy**. This can be defined in two main ways:

1. **Unconditional (Perfect) Secrecy:** Security holds regardless of the adversary's computational power.
2. **Computational Secrecy:** Security holds against adversaries with limited (polynomial-time) computational resources. This is the standard model for modern cryptography.

5.2 Perfect Secrecy

The concept of perfect secrecy was formalized by Claude Shannon. It is a statistical notion of security based on probability theory.

5.2.1 Probabilistic Setting

To define perfect secrecy, we model the plaintext, key, and ciphertext as random variables:

- **Key Distribution $P(K = k)$:** This distribution is determined by the key generation algorithm. Typically, we assume the key is chosen uniformly at random from the set of all possible keys $\mathcal{K}$.
- **Plaintext Distribution $P(M = m)$:** This is the *a priori* probability that a specific message m will be transmitted. It depends on the context, language, and communicating parties. The adversary is assumed to have knowledge of this distribution.
- **Ciphertext Distribution $P(C = c)$:** This distribution is determined by the distributions of the key and the plaintext, along with the encryption algorithm E.

We assume that the choice of the key is independent of the choice of the plaintext, as the key is generated before the message is known.

5.2.2 Definition of Perfect Secrecy

An encryption scheme is **perfectly secret** if the observation of the ciphertext provides no additional information about the plaintext. Formally, for every possible plaintext distribution, every message m, and every ciphertext c for which $P(C = c) > 0$:

$$P(M = m \mid C = c) = P(M = m)$$

This definition states that the *a posteriori* probability of the message being m given that the ciphertext is c is the same as the *a priori* probability of the message being m. The ciphertext leaks no information about the plaintext.

5.2.3 Alternative Formulations

The definition of perfect secrecy can be expressed in equivalent ways:

Perfect Indistinguishability: An adversary cannot distinguish between the encryptions of two different plaintexts. Formally, for every possible plaintext distribution, every pair of messages m_0, m_1, and every ciphertext c:

$$P(C = c \mid M = m_0) = P(C = c \mid M = m_1)$$

This means that the probability distribution of the ciphertext is independent of the plaintext that was encrypted.

Adversarial Indistinguishability Game: This provides a more interactive way to think about security.

1. The adversary chooses two distinct messages, m_0 and m_1.
2. We choose a random bit $b \in \{0, 1\}$. We generate a random key k and compute the challenge ciphertext $c = E(m_b, k)$.
3. We send c to the adversary.
4. The adversary must guess the value of b. The adversary wins if their guess is correct.

An encryption scheme achieves perfect secrecy if and only if, for **every** possible adversary, the probability of winning this game is exactly $\frac{1}{2}$.

$$\Pr(\text{adversary wins}) = \frac{1}{2}$$

This implies that the adversary can do no better than randomly guessing. The ciphertext gives them no advantage.

5.3 The One-Time Pad and Perfect Secrecy

The One-Time Pad (OTP) is the canonical example of a perfectly secret encryption scheme. Let's consider the binary version:

- The message space $\mathcal{M}$ and key space $\mathcal{K}$ are the set of all binary strings of length n.
- The key k is chosen uniformly at random from $\mathcal{K}$. So, $P(K = k) = \frac{1}{2^n}$.
- Encryption is bitwise XOR: $c = m \oplus k$.

To prove perfect secrecy, we show that $P(C = c \mid M = m)$ is independent of m. For any message m and any ciphertext c:

$$\begin{aligned} P(C = c \mid M = m) &= P(M \oplus K = c \mid M = m) \\ &= P(m \oplus K = c) \\ &= P(K = c \oplus m) \end{aligned}$$

Since the key $k = c \oplus m$ is a specific n-bit string, and the key is chosen uniformly at random, the probability of it being chosen is:

$$P(K = c \oplus m) = \frac{1}{2^n}$$

This probability is constant for any m and c. Thus, $P(C = c \mid M = m)$ is independent of m, which proves that the One-Time Pad is perfectly secret.

5.4 Limitations of Perfect Secrecy: Shannon's Theorem

While perfect secrecy is the strongest possible notion of security, it comes at a very high cost. Claude Shannon proved a fundamental limitation of perfectly secret cryptosystems.

Shannon's Theorem: In any perfectly secret encryption scheme, the size of the key space must be at least as large as the size of the message space.

$$|\mathcal{K}| \geq |\mathcal{M}|$$

Proof Sketch:

1. For an encryption scheme to be secure, every plaintext must be able to be decrypted from a given ciphertext, but the specific plaintext cannot be uniquely determined. This means that for any given ciphertext c, there must be a possible key that maps it back to *any* possible plaintext $m \in \mathcal{M}$.
2. Let's fix a ciphertext c. For each plaintext $m_i \in \mathcal{M}$, there must exist at least one key $k_i \in \mathcal{K}$ such that $D(c, k_i) = m_i$.
3. All these keys $(k_1, k_2, \ldots, k_{|\mathcal{M}|})$ must be distinct. If two different plaintexts m_i and m_j were decrypted from the same ciphertext c with the same key k, this would violate the basic property that decryption with a given key is a well-defined function.
4. Therefore, for a single ciphertext c, there must be at least $|\mathcal{M}|$ distinct keys. This implies that the total number of keys must be at least the total number of possible messages, so $|\mathcal{K}| \geq |\mathcal{M}|$.

For a system like the One-Time Pad, this theorem is satisfied with equality: $|\mathcal{K}| = |\mathcal{M}| = |\mathcal{C}|$. This is a defining characteristic of "perfect systems".

This theorem reveals the main drawback of perfect secrecy: to encrypt a message of n bits, we need a key of at least n bits. This is impractical for most real-world applications where vast amounts of data are transmitted.

5.5 Unicity Distance

What happens if the key space is smaller than the message space ($|\mathcal{K}| < |\mathcal{M}|$)? Shannon's theorem tells us perfect secrecy is impossible. However, this doesn't mean the system is completely broken.

With a small amount of ciphertext, an adversary might find multiple plausible plaintexts corresponding to different keys. For example, if the ciphertext is "PHHW", a brute-force attack on a shift cipher might yield:

- Key 3: "MEET" (plausible)
- Key 19: "WOOD" (plausible)

However, as the amount of available ciphertext increases, the number of plausible plaintexts decreases. Eventually, with enough ciphertext, only one key will produce a meaningful plaintext.

The **unicity distance** is a concept from information theory that estimates the minimum length of ciphertext required for an adversary to uniquely determine the key (and thus the plaintext) with high probability, assuming the adversary has infinite computational power.

Take-home message:

- Perfect secrecy is possible but requires a key space as large as the message space, making it impractical.
- A finite key space can only unconditionally protect very short or low-redundancy messages.
- Practical cryptography must be based on a different, more relaxed security model.

5.6 Computational Security

Since perfect secrecy is impractical, modern cryptography relies on **computational security**. This model relaxes the assumptions about the adversary:

1. The adversary has **limited computational resources**.
2. The adversary can succeed with a very small, **negligible probability**.

5.6.1 The Asymptotic Approach

To formalize these concepts, we use an asymptotic approach. Security is analyzed in terms of a **security parameter**, denoted by n. Typically, n represents the length of the key in bits.

- **Polynomial-Time Adversary:** We assume the adversary can only run algorithms whose running time is a polynomial function of the security parameter n. We denote this as $\text{poly}(n)$. This is our model for a "computationally bounded" or "efficient" adversary. Algorithms with exponential running time, like $O(2^n)$, are considered infeasible for large n. For example, a brute-force attack on an n-bit key takes $O(2^n)$ time.
- **Negligible Probability:** A function $\epsilon(n)$ is considered negligible if it decreases faster than the inverse of any polynomial in n. Formally, for every constant $c > 0$, there exists an integer N such that for all $n > N$, $\epsilon(n) < n^{-c}$. We denote this as $\text{negl}(n)$. Examples include 2^{-n} and $n^{-\log n}$. This is our model for a "negligible probability of success".

This asymptotic framework is powerful because it allows us to make robust statements about security that hold up even as computers get faster. Doubling the key length (e.g., from 128 to 256 bits) typically results in a small, polynomial increase in the cost of encryption, but an exponential increase in the cost of a brute-force attack, rendering it infeasible.

5.6.2 Computationally-Secure Encryption: Revised Definition

We now revise our definition of a private-key encryption scheme in this computational context.

- A scheme is defined by three polynomial-time algorithms: $\text{Gen}(n)$, $\text{E}(m, k)$, and $\text{D}(c, k)$.
- We revisit the adversarial indistinguishability game, but now with a computationally bounded adversary.

An encryption scheme has **indistinguishable encryptions in the presence of an eavesdropper** (a formal notion of computational security, often abbreviated as EAV-IND) if, for every probabilistic, polynomial-time adversary:

$$\Pr(\text{adversary wins}) \leq \frac{1}{2} + \text{negl}(n)$$

This definition means that any efficient adversary can only gain a negligible advantage over random guessing. For all practical purposes, this is as good as perfect secrecy. An adversary with reasonable resources cannot distinguish the encryption of one message from another.

6 Private-Key Cryptography

In the realm of computational security, we build secure encryption schemes upon fundamental building blocks known as **cryptographic primitives**. For private-key (or symmetric) cryptography, two main primitives are considered:

1. **Pseudorandom Generators (PRGs):** These are used to construct *stream ciphers*.
2. **Pseudorandom Permutations (PRPs):** These are used to construct *block ciphers*.

6.1 Pseudorandomness

The concept of pseudorandomness is fundamental to modern cryptography. Loosely speaking, a string of bits is considered pseudorandom if a computationally bounded observer cannot distinguish it from a truly random string of the same length (i.e., a string chosen with uniform probability from all possible strings). Pseudorandomness is therefore a property of the *distribution* of strings generated by an algorithm.

6.2 Pseudorandom Generators (PRGs) and Stream Ciphers

6.2.1 Pseudorandom Generators (PRGs)

A **Pseudorandom Generator (PRG)** is a deterministic algorithm that takes a short, truly random input, called a **seed**, and expands it into a much longer output string that appears random.

Let G be a deterministic algorithm that takes an n-bit seed s and produces an ℓ-bit output string $y = G(s, \ell)$, where $\ell \gg n$. While the output distribution of G is far from uniform (it can only produce 2^n distinct strings out of the 2^ℓ possible ones), its strength lies in its computational indistinguishability from a truly random string.

Formal Definition: An algorithm G is a pseudorandom generator if, for every probabilistic, polynomial-time distinguisher algorithm D:

$$|\Pr[D(r) = 1] - \Pr[D(G(s)) = 1]| \leq \text{negl}(n)$$

where r is a truly random ℓ-bit string and s is a truly random n-bit seed. This means no efficient algorithm can tell the difference between the output of the PRG and a truly random string with more than a negligible probability.

Assumption: We assume that well-known algorithms like **Salsa20** and **ChaCha20** behave as secure pseudorandom generators. Based on this assumption, we can build secure cryptosystems.

6.2.2 Stream Ciphers

A stream cipher is a direct application of a PRG. It mimics the One-Time Pad, but replaces the truly random key stream with a pseudorandom one.

Construction:

- **Key Generation:** Choose a secret key k uniformly at random from all n-bit strings. This key will be the seed for the PRG.
- **Encryption:** To encrypt a message m of length $|m|$, generate a pseudorandom string of the same length using the key: $G(k, |m|)$. The ciphertext is the XOR of the message and this *keystream*:

$$c = m \oplus G(k, |m|)$$

- **Decryption:** Decryption is identical:

$$m = c \oplus G(k, |c|)$$

Security: Under the assumption that G is a secure PRG, the resulting stream cipher has indistinguishable encryptions in the presence of an eavesdropper (is EAV-IND secure). A proof can be made by reduction: if an adversary could distinguish encryptions, they could be used to construct a distinguisher for the underlying PRG, which contradicts our assumption.

6.2.3 The Problem with Multiple Encryptions

A critical weakness arises if the same key (seed) is used to encrypt multiple messages. **Never use a stream cipher key more than once**. This is the same vulnerability as reusing a one-time pad. If an adversary has two ciphertexts $c_1 = m_1 \oplus G(k)$ and $c_2 = m_2 \oplus G(k)$, they can compute:

$$c_1 \oplus c_2 = (m_1 \oplus G(k)) \oplus (m_2 \oplus G(k)) = m_1 \oplus m_2$$

XORing the two ciphertexts cancels out the keystream, revealing the XOR of the two plaintexts. This leaks significant information and can lead to a full recovery of both messages if the language has enough redundancy.

To safely encrypt multiple messages, we need a different keystream for each message. This can be achieved in practice using an **Initialization Vector (IV)**.

- The PRG is modified to take a key k and an IV: $G(k, \text{IV})$.
- For each message, a new, unique, and random IV is generated.
- The encryption becomes: $E(m, k, \text{IV}) = m \oplus G(k, \text{IV})$.
- The IV is not secret and is sent along with the ciphertext, typically as a prefix: $c = \text{IV}||(m \oplus G(k, \text{IV}))$.

This ensures that even with the same key, a different keystream is used for each message, provided the IV is not reused.

6.3 Security Against Chosen-Plaintext Attacks (CPA)

A stronger notion of security is required for many real-world applications. An adversary might be able to trick a system into encrypting messages of their choice. This is modeled by the **Chosen-Plaintext Attack (CPA)**.

In the CPA indistinguishability game, the adversary has access to an **encryption oracle**, which will encrypt any message the adversary submits with the secret key. The adversary can use this oracle before and after receiving the challenge ciphertext.

An encryption scheme is **CPA-secure** (or IND-CPA) if any polynomial-time adversary wins the CPA game with a probability at most $\frac{1}{2} + \text{negl}(n)$.

A key takeaway is that any **deterministic** encryption scheme is **not CPA-secure**. If encryption is deterministic, an adversary can win the CPA game with probability 1. They simply query the oracle with their two chosen messages, m_0 and m_1, to get c_0 and c_1. When they receive the challenge ciphertext c, they just check if $c = c_0$ or $c = c_1$ to determine the bit b.

Therefore, CPA-security requires **probabilistic encryption**: encrypting the same message multiple times must produce different, unpredictable ciphertexts. A stream cipher used with a fresh, random IV for each message is an example of a probabilistic scheme that is believed to be CPA-secure.

6.4 Pseudorandom Functions (PRFs) and Permutations (PRPs)

While PRGs are useful, a more powerful and versatile primitive is the pseudorandom function.

Random Function: A truly random function is a function chosen uniformly at random from the set of all possible functions that map n-bit inputs to n-bit outputs. It can be imagined as a giant lookup table where each output value is chosen completely randomly.

Pseudorandom Function (PRF): A PRF is a keyed, deterministic function, denoted F_k, that is computationally indistinguishable from a true random function. An adversary with oracle access to either F_k (for a random key k) or a true random function f cannot tell which one they are interacting with.

Pseudorandom Permutation (PRP): A PRP is a PRF that is also a permutation (i.e., it is a one-to-one and onto mapping). This means it is efficiently invertible.

6.5 Block Ciphers and Modes of Operation

6.5.1 Block Ciphers

A **block cipher** is a practical implementation of a pseudorandom permutation (PRP). It is a deterministic, invertible algorithm that operates on fixed-size blocks of bits.

- It takes an n-bit block of plaintext and an n-bit key as input.
- It produces an n-bit block of ciphertext as output.

Crucial point: A block cipher, by itself, is a primitive, **not** a secure encryption scheme. It is deterministic, and therefore not CPA-secure.

6.5.2 Modes of Operation

To securely encrypt messages longer than a single block, block ciphers must be used in a specific **mode of operation**. These are cryptographic constructions that define how to repeatedly apply the block cipher's single-block operation.

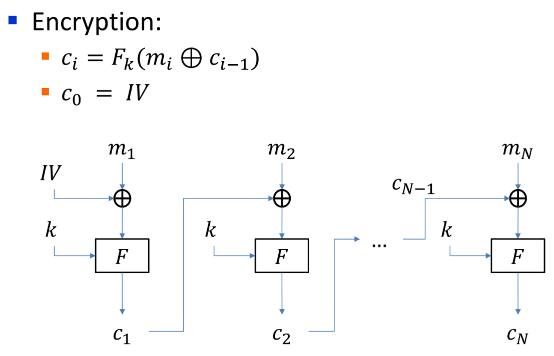
Electronic Codebook (ECB) Mode: This is the simplest but most insecure mode. The message is divided into blocks, and each block is encrypted independently.

$$c_i = F_k(m_i)$$

Flaw: It is deterministic. Identical plaintext blocks will always result in identical ciphertext blocks, leaking significant information about the message structure. **ECB should not be used in practice**.

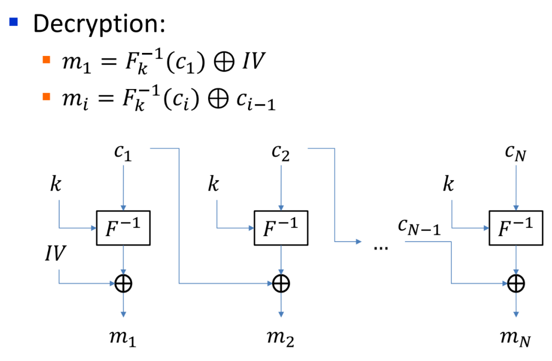
Cipher Block Chaining (CBC) Mode: To fix the problem with ECB, CBC introduces chaining. Each plaintext block is XORed with the *previous* ciphertext block before being encrypted.

$$c_i = F_k(m_i \oplus c_{i-1})$$



- Encryption:
  - $c_i = F_k(m_i \oplus c_{i-1})$
  - $c_0 = IV$

(a) CBC encryption scheme.



- Decryption:
  - $m_1 = F_k^{-1}(c_1) \oplus IV$
  - $m_i = F_k^{-1}(c_i) \oplus c_{i-1}$

(b) CBC decryption scheme.

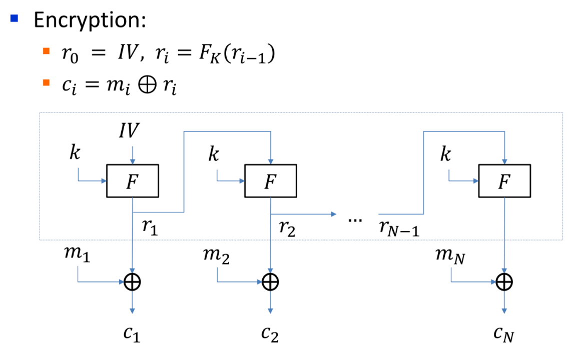
For the first block, a random **Initialization Vector (IV)** is used as c_0. CBC is CPA-secure **ONLY** if the IV is random and unpredictable, and the underlying block cipher is a secure PRP. Decryption can be parallelized, but encryption is sequential.

Output Feedback (OFB) Mode: This mode effectively turns a block cipher into a stream cipher. It generates a keystream by repeatedly encrypting an IV.

$$r_i = F_k(r_{i-1}) \quad \text{(where } r_0 = \text{IV)}$$

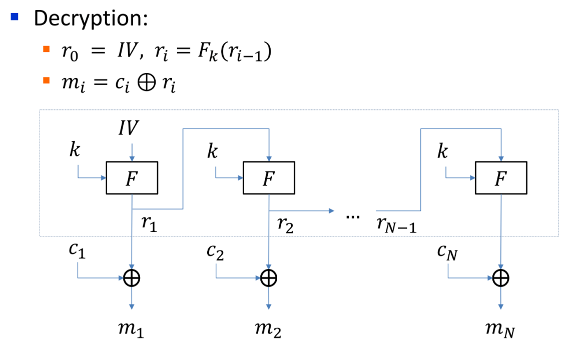
$$c_i = m_i \oplus r_i$$

OFB is CPA-secure. Since it's a stream cipher, the block cipher primitive does not need to be invertible.



- Encryption:
  - $r_0 = IV, \; r_i = F_K(r_{i-1})$
  - $c_i = m_i \oplus r_i$

(a) OFB encryption scheme.



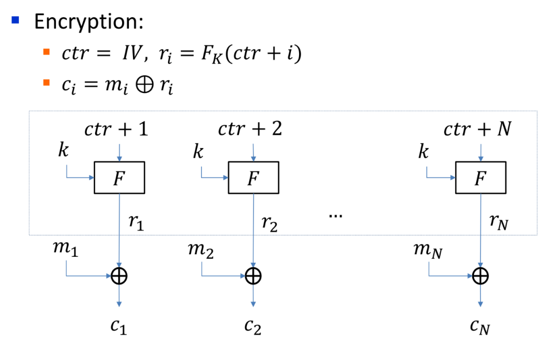
- Decryption:
  - $r_0 = IV, \; r_i = F_k(r_{i-1})$
  - $m_i = c_i \oplus r_i$

(b) OFB decryption scheme.

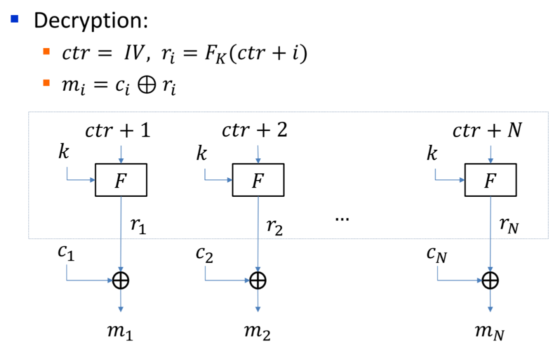
Counter (CTR) Mode: This is another mode that turns a block cipher into a stream cipher, and it is highly popular due to its performance and flexibility. It generates the keystream by encrypting the values of a counter.

$$r_i = F_k(\mathrm{IV} + i)$$

$$c_i = m_i \oplus r_i$$



(a) CTR encryption scheme.



(b) CTR decryption scheme.

Advantages: CTR mode is CPA-secure, fully parallelizable for both encryption and decryption, and allows for random access to any block of the message.

7 Practical Symmetric Ciphers

This section delves into the design of modern symmetric ciphers, focusing on the principles behind block ciphers and the structure of widely used algorithms like AES. We will also explore practical stream ciphers that are not based on block cipher modes of operation.

7.1 Practical Block Cipher Design

The ideal block cipher would be a true random permutation mapping n-bit inputs to n-bit outputs. This can be implemented as a lookup table, but is only practical for very small block sizes (n), such as 4 to 8 bits. These small, fixed permutations are called **Substitution Boxes (S-boxes)**.

For modern block sizes (e.g., 128 bits), a lookup table is infeasibly large (a key of 128×2^{128} bits would be needed). Therefore, practical block ciphers are constructed using iterative structures that approximate the behavior of a random permutation.

7.1.1 Substitution-Permutation (S-P) Networks

A common design for modern block ciphers is the **Substitution-Permutation (S-P) Network**. This design, proposed by Claude Shannon, iteratively applies rounds of substitution and permutation operations to the data.

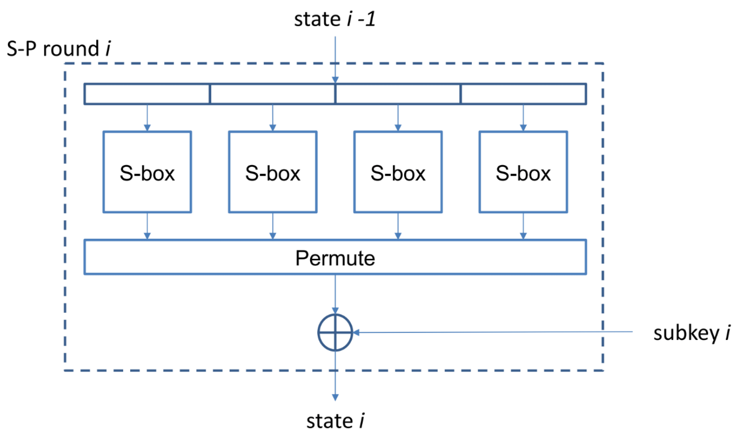
Substitution (S-boxes): A non-linear transformation that replaces small blocks of bits with other blocks. This is the core source of non-linearity and is crucial for security.

Permutation (P-boxes): A linear transformation that shuffles the bits. This spreads the output of one S-box across the inputs of multiple S-boxes in the next round.

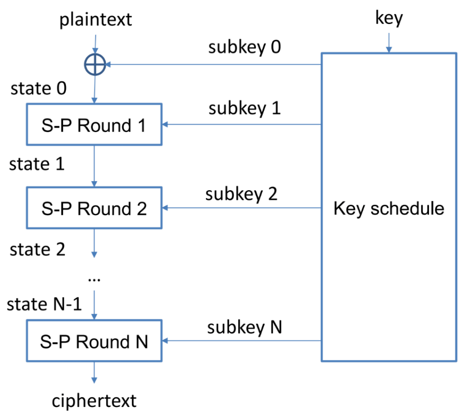
The goal of this design is to achieve Shannon's properties of **confusion** and **diffusion**:

- **Confusion:** To make the relationship between the key and the ciphertext as complex and involved as possible. This is primarily achieved by the S-boxes.
- **Diffusion:** To ensure that a single change in a plaintext bit affects many ciphertext bits, and vice-versa. This is achieved by the P-boxes, which spread the influence of each bit.

A block cipher based on an S-P network consists of multiple rounds. Each round takes the output of the previous round as input and performs the S-box substitutions, the P-box permutation, and mixes in a **round key** (derived from the main key via a **key schedule**). The repeated application of these rounds ensures that after a sufficient number of iterations, every output bit depends on every input bit and every key bit, achieving a strong **avalanche effect**.



(a) S-P round scheme.



(b) Sub-keys obtaining scheme.

IMPORTANT! Note that the first step is to combine the plaintext with the first subkey. This is because the SP round function is public (according to Kerckhoffs' Principle). Therefore, if an adversary knows the plaintext, it can combine it with the SP function and XOR it with the obtained ciphertext to recover the subkey.

7.2 The Advanced Encryption Standard (AES)

The **Advanced Encryption Standard (AES)**, also known as Rijndael, is the most widely used block cipher today. It was selected by the U.S. National Institute of Standards and Technology (NIST) in 2001.

Specifications:

- **Block Size:** Fixed at 128 bits (16 bytes).
- **Key Sizes:** Can be 128, 192, or 256 bits.
- **Number of Rounds:** Depends on the key size: 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

7.2.1 AES Structure

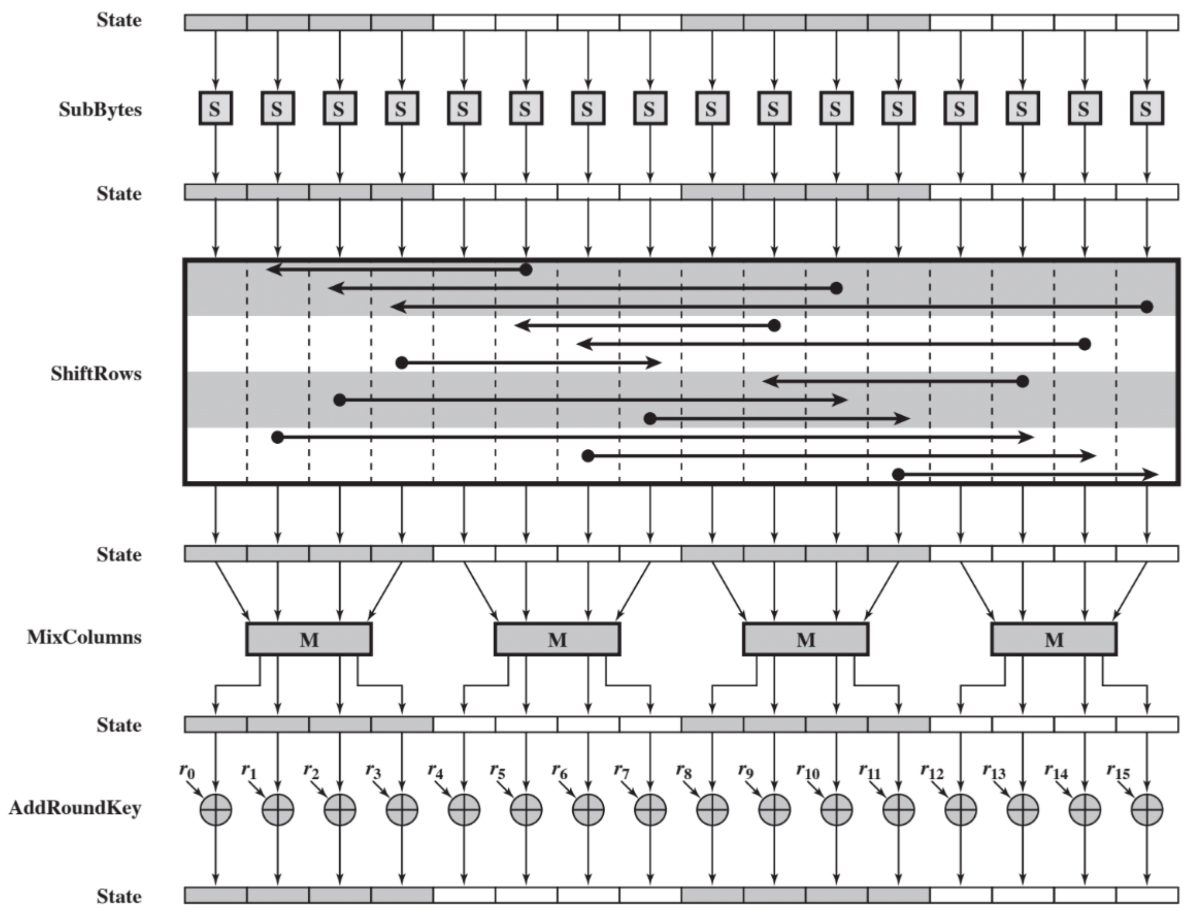

State
SubBytes
S
State
ShiftRows
State
MixColumns
M
State
AddRoundKey
r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8 r_9 r_{10} r_{11} r_{12} r_{13} r_{14} r_{15}
State

Figure 7: AES round.

AES operates on a 4x4 matrix of bytes called the **state**. The initial state is the 128-bit plaintext block. The algorithm proceeds as follows:

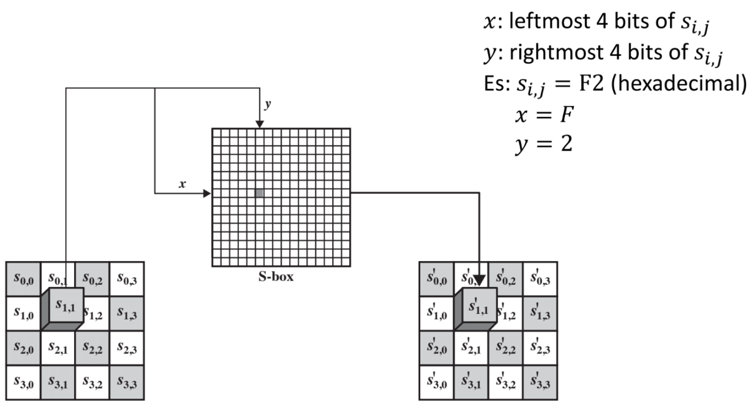
1. **Initial Key Addition:** The state is XORed with the first round key.
2. **Rounds:** The state is transformed by applying a round function for a specified number of rounds (e.g., 10 for AES-128). The final round is slightly different from the others.
3. **Final State:** The final state after the last round is the 128-bit ciphertext block.

7.2.2 The AES Round Structure in Detail

Each standard round of AES (except for the last one) is composed of four distinct, invertible transformations that operate on the *state*, a 4x4 matrix of bytes. These layers provide the confusion and diffusion properties essential for the cipher's security.

1. SubBytes (Substitution) This is a **non-linear** substitution step and is the only non-linear transformation in the entire cipher, making it a critical source of security. Each of the 16 bytes of the state is replaced with a different byte according to a fixed lookup table called the **S-box**.

The substitution is performed as follows: each byte is split into its two 4-bit nibbles. The first nibble (4 most significant bits) is used as a row index into the S-box, and the second nibble (4 least significant bits) is used as a column index. The byte at the intersection of that row and column in the S-box table becomes the new value for that position in the state.



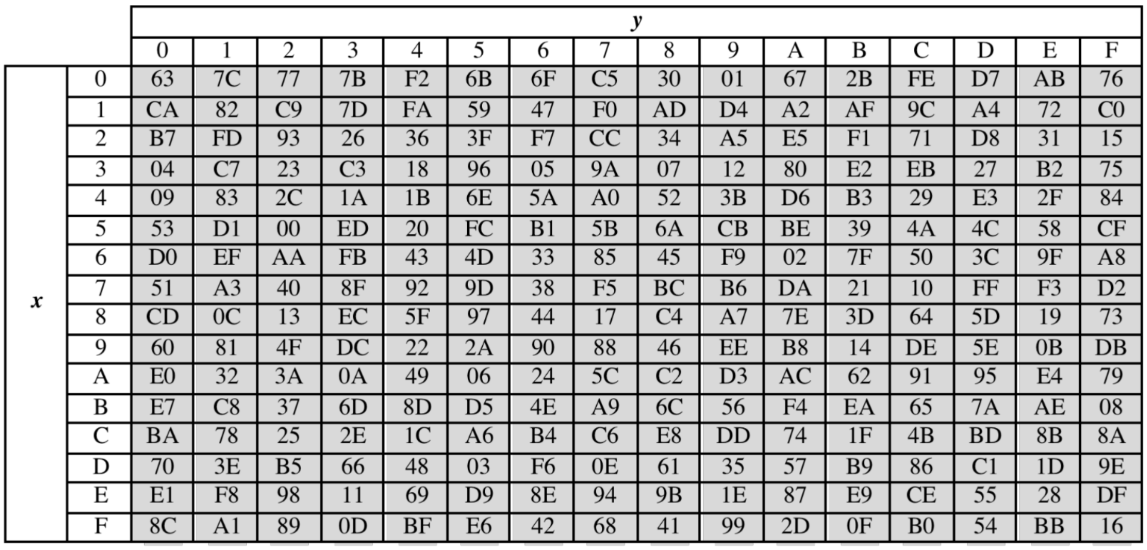
x: leftmost 4 bits of $s_{i,j}$
y: rightmost 4 bits of $s_{i,j}$
Es: $s_{i,j}$ = F2 (hexadecimal)
$x = F$
$y = 2$
y
x
S-box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 8: The SubBytes operation using the S-box. Each byte $s_{i,j}$ is replaced by $s'_{i,j}$.

The AES S-box is not a random permutation. It is mathematically constructed to be resistant to linear and differential cryptanalysis. The construction involves two steps:

1. Taking the multiplicative inverse of the byte in the finite field GF(2^8).
2. Applying a specific affine (linear + constant) transformation over GF(2).

This construction ensures that the S-box has excellent non-linearity properties. The inverse operation, **InvSubBytes**, used in decryption, uses an inverse S-box which is constructed by applying the inverse affine transformation followed by finding the multiplicative inverse.



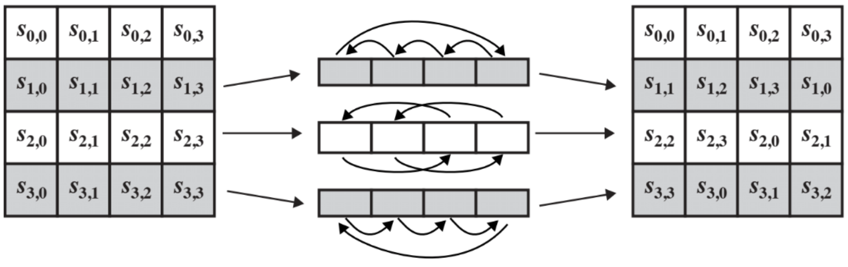
		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 9: The AES S-box table for the SubBytes transformation. (Da Slide 28)

2. ShiftRows (Permutation) This is a permutation step that provides diffusion by transposing bytes between columns. The rows of the state matrix are cyclically shifted to the left by different offsets:

- **Row 0:** is not shifted.
- **Row 1:** is shifted left by 1 byte.
- **Row 2:** is shifted left by 2 bytes.
- **Row 3:** is shifted left by 3 bytes.

The key feature of this operation is that bytes from a single column in the input state are spread across four different columns in the output state. This ensures that the `MixColumns` step in the next round will operate on a mix of bytes from different columns, propagating changes throughout the entire state.



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

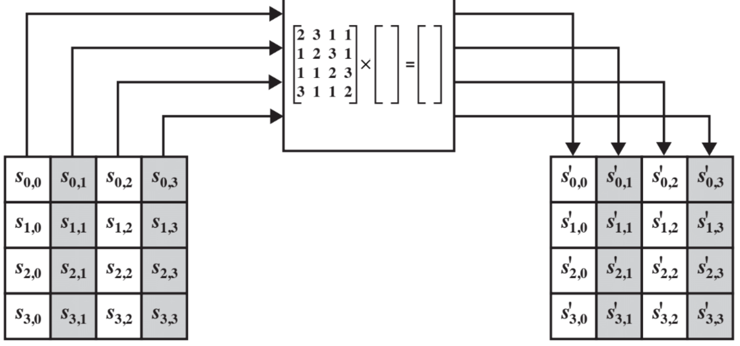
Figure 10: The ShiftRows operation.

3. MixColumns (Diffusion) This step provides strong diffusion within each of the four columns of the state. Each column is treated as a four-term polynomial over the finite field GF(2^8) and is multiplied modulo $x^4 + 1$ by a fixed polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. This can be represented as a matrix multiplication in GF(2^8):

$$\begin{pmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{pmatrix}$$

where $s'_{i,c}$ and $s_{i,c}$ are the bytes of the state in column c after and before the transformation, respectively. All arithmetic is performed in GF(2^8).

Together, `ShiftRows` and `MixColumns` ensure that after just a few rounds, a change in a single input byte will affect all 16 bytes of the state. This property is crucial for the security of the cipher. This step is omitted in the final round of encryption to ensure that the cipher remains invertible and for performance, as it is a linear operation that would be immediately undone by the final `AddRoundKey` and the inverse key schedule during decryption.



2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
×
=
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 11: The MixColumns operation, represented as a matrix multiplication on each column.

4. AddRoundKey This is the step where the secret key is introduced into the state. A 128-bit **round key**, derived from the main encryption key via the key schedule algorithm, is bitwise XORed with the 128-bit state. This operation is its own inverse (since $A \oplus B \oplus B = A$), which simplifies the decryption process. It is a very simple and fast operation, but it is critical as it makes the encryption dependent on the key.

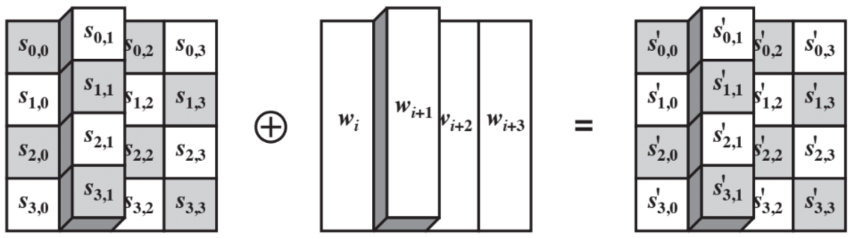

$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$\oplus$
w_i w_{i+1} w_{i+2} w_{i+3}
=
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 12: The AddRoundKey operation: a bitwise XOR of the state and the round key.

7.2.3 Decryption

The decryption process in AES involves applying the inverse of these four transformations in the reverse order. The corresponding inverse operations are **InvShiftRows**, **InvSubBytes**, **InvMixColumns**, and **AddRoundKey** (which is its own inverse). The round keys from the key schedule are also used in reverse order.

7.2.4 Security of AES

AES has withstood extensive cryptanalysis for over 20 years. It is designed to be resistant to known attacks, including **differential cryptanalysis** and **linear cryptanalysis**. However, specific implementations can be vulnerable to **side-channel attacks**, such as **timing attacks**, if not carefully implemented (e.g., using constant-time operations instead of lookup tables for the S-box).

7.3 Dedicated Stream Ciphers

While block ciphers in CTR or OFB mode can be used as stream ciphers, there are also dedicated stream cipher designs that are often faster and require less code.

A simple (but insecure) example is a **Linear Feedback Shift Register (LFSR)**. An LFSR generates a sequence of bits. While it can have a long period and good statistical properties, its linearity makes it cryptographically insecure. Knowing just $2n$ bits of the output stream is enough to solve a system of linear equations and recover the entire internal state (the "key").

Modern dedicated stream ciphers add non-linearities to LFSR-like structures to resist such attacks.

7.3.1 RC4

RC4 is a widely known stream cipher designed by Ron Rivest in 1987. It was initially a trade secret but was reverse-engineered and made public. It is very simple to implement in software. However, significant vulnerabilities have been discovered over the years:

- Biases in the initial bytes of the keystream.
- Vulnerabilities when used with a public IV, as was the case in the WEP protocol.

Due to these weaknesses, the use of RC4 has been prohibited in modern secure protocols. RFC 7465 formally forbids its use in TLS.

7.3.2 Salsa20 and ChaCha20

Salsa20 is a modern and secure stream cipher designed by Daniel J. Bernstein in 2005. It generates a keystream by applying a pseudorandom function to a combination of a key, an IV (or nonce), and a counter. It is built on a simple set of operations known as **ARX** (Add-Rotate-XOR) on 32-bit words, which are very fast on modern CPUs.

ChaCha20 is a variant of Salsa20, also designed by Bernstein. It modifies the core round function to increase diffusion and improve performance on some architectures. Key features of ChaCha20:

- Very fast in software implementations (often faster than AES).

- No known cryptanalytic weaknesses.
- Not susceptible to timing attacks as its operations take constant time.

ChaCha20, often combined with the Poly1305 authenticator, has been adopted as a modern, high-security cipher suite in many protocols, including TLS 1.3 and various VPNs.

8 Message Authentication Codes (MACs)

8.1 Beyond Confidentiality: Authentication and Integrity

Private-key encryption schemes, as discussed so far, provide **confidentiality**—they protect the content of a message from being read by unauthorized parties. However, encryption alone *does not* guarantee two other critical security properties:

- **Source Authentication:** Proof that a message originates from the claimed sender.
- **Message Integrity:** Assurance that a message has not been altered or tampered with during transmission.

A common misconception is that if only the sender and receiver know the secret key, a correctly decrypted message must be authentic. This is false. Many encryption schemes are **malleable**, meaning that an attacker can systematically modify a ciphertext and produce a new, valid ciphertext that decrypts to a predictably altered plaintext, without ever knowing the underlying message.

Examples of Malleability:

- **Stream Ciphers (and CTR/OFB modes):** In a stream cipher where $c = m \oplus k_s$ (where k_s is the keystream), an attacker can flip a bit in the plaintext by flipping the corresponding bit in the ciphertext. If an attacker computes $c' = c \oplus e$ (where e is an error vector, e.g., a single '1' bit), the decryption will be:

$$D(c') = c' \oplus k_s = (c \oplus e) \oplus k_s = (m \oplus k_s \oplus e) \oplus k_s = m \oplus e$$

  The plaintext is predictably modified, but the decryption is still "successful" in that the algorithm produces an output.

- **AES-CBC Mode:** Modifying a ciphertext block c_i will completely corrupt the corresponding plaintext block m_i but will produce a predictable, targeted change in the next plaintext block, m_{i+1}, since $m_{i+1} = D_k(c_{i+1}) \oplus c_i$. Modifying the IV has a similar predictable effect on the first plaintext block.

Since an attacker can manipulate messages, we need a separate mechanism to ensure integrity and authenticity. This mechanism is the **Message Authentication Code (MAC)**.

8.2 The MAC Model

A Message Authentication Code is a cryptographic primitive that provides integrity and authenticity. It is a symmetric-key algorithm where two parties, sharing a secret key, can verify that messages sent between them have not been altered.

A MAC scheme consists of three algorithms:

Key Generation Algorithm (Gen): Generates a secret key k (e.g., a uniformly random n-bit string).

MAC Algorithm (Tagging): Takes the secret key k and a message m as input and produces a short, fixed-length string called a **tag** (or MAC).

$$t = \text{MAC}(k, m)$$

Verification Algorithm (V): Takes the key k, a message m, and a tag t as input, and outputs a bit: 1 if the tag is valid for the message, and 0 otherwise.

$$\text{V}(k, m, t) \in \{0, 1\}$$

For the scheme to be correct, it must hold that for any key k and message m:

$$\text{V}(k, m, \text{MAC}(k, m)) = 1$$

In most practical implementations, the verification algorithm is deterministic and simply re-computes the tag for the received message m using the shared key k and checks if the computed tag matches the received tag t.

8.3 Security Definition for MACs

The security goal for a MAC is to be **unforgeable**. An attacker should not be able to produce a valid tag for any message for which they have not already seen a tag generated by the legitimate sender.

The strongest security definition is **Existential Unforgeability under an Adaptive Chosen-Message Attack (EUF-CMA)**. This is defined via the following game:

1. A random key k is generated.
2. The adversary has access to a **MAC oracle**, $\text{MAC}(k, \cdot)$. They can query this oracle with any messages of their choice $(m_1, m_2, \dots, m_q)$ and receive the corresponding tags $(t_1, t_2, \dots, t_q)$.
3. The adversary's goal is to output a new message/tag pair (m, t).
4. The adversary wins if the pair is valid (i.e., $\text{V}(k, m, t) = 1$) **and** the message m was not one of the messages they previously queried to the oracle (i.e., $m \notin \{m_1, \dots, m_q\}$).

A MAC scheme is EUF-CMA secure if for every probabilistic, polynomial-time adversary, the probability of winning this game is negligible.

$$\Pr(\text{adversary wins}) \leq \text{negl}(n)$$

This strong definition means the adversary cannot forge a tag for *any* new message, even if the message is nonsensical (*existential* forgery).

8.4 Achieving CCA Security

The ultimate goal for a symmetric encryption scheme is to be secure against **Chosen-Ciphertext Attacks (CCA)**. A CCA-secure scheme protects confidentiality even if the adversary has access to a **decryption oracle**, which allows them to decrypt any ciphertext of their choice (except for the challenge ciphertext itself).

CPA-secure schemes like CBC or CTR are not CCA-secure because they are malleable. An adversary can use the decryption oracle in a CCA game by submitting a slightly modified version of the challenge ciphertext and use the result to win the game.

8.4.1 The Padding Oracle Attack

A famous real-world example of a partial decryption oracle is the **padding oracle attack**. When using block ciphers, messages often need to be padded to a multiple of the block length. A common standard is PKCS#5, where a message is padded by adding b bytes, each with the value b. If a server, upon decrypting a message, returns an error message that distinguishes between "bad MAC" and "invalid padding," this "invalid padding" message acts as a partial decryption oracle. It leaks one bit of information: whether the padding of the decrypted plaintext is valid or not. Serge Vaudenay showed that this single bit of information is enough to decrypt messages encrypted in CBC mode. An attacker can craft ciphertexts by modifying the last byte of the preceding ciphertext block and send them to the oracle. By observing which modifications result in a valid padding, the attacker can iteratively recover the bytes of the plaintext one by one.

8.5 Authenticated Encryption (AE)

To achieve both confidentiality and integrity, and to protect against malleability and CCAs, we combine a CPA-secure encryption scheme with a secure MAC. This combined construction is known as **Authenticated Encryption (AE)**.

There are three canonical ways to combine encryption and authentication:

1. **Encrypt-and-Authenticate (E&A):** The plaintext is encrypted and MAC'd independently. The ciphertext and tag are sent together.

   $$c = E(k_E, m), \quad t = \text{MAC}(k_M, m)$$

   Flaw: This is generally insecure. If the MAC is deterministic, it can leak information about the plaintext, violating CPA security.

2. **Authenticate-then-Encrypt (AtE):** A MAC is computed on the plaintext, and then the plaintext and the tag are encrypted together.

   $$t = \text{MAC}(k_M, m), \quad c = E(k_E, m||t)$$

   Flaw: This approach can be vulnerable. For example, the decryption endpoint must first decrypt, then check the tag. If the decryption of a malformed ciphertext leads to a padding error that is distinguishable from a tag verification error, it could re-enable padding oracle attacks.

3. **Encrypt-then-Authenticate (EtA):** The plaintext is encrypted first, and then a MAC is computed over the resulting ciphertext.

   $$c = E(k_E, m), \quad t = \text{MAC}(k_M, c)$$

   This is the **most secure and recommended** approach. The receiver first verifies the integrity and authenticity of the ciphertext by checking the tag. If the tag is invalid, the ciphertext is discarded immediately without any attempt to decrypt it. This prevents any information leakage from the decryption process (like padding oracles). Only authentic ciphertexts are ever decrypted.

An **Authenticated Encryption (AE)** scheme is one that is both **CCA-secure** and **unforgeable**. The Encrypt-then-Authenticate construction, when built with a CPA-secure encryption scheme and a secure MAC using independent keys, provides this level of security.

9 Cryptographic Hash Functions

9.1 Introduction to Hash Functions

A **hash function**, denoted as H, is a mathematical function that takes a variable-length input message, m, and produces a fixed-length output, h, called a hash value, hash code, or message digest.

$$h = H(m)$$

Hash functions are a fundamental tool in cryptography and are used in a wide variety of applications, including:

- Message Authentication Codes (MACs)
- Digital Signatures
- One-way storage of passwords
- Key Derivation Functions (KDFs)
- File fingerprinting and integrity verification

While simple hash functions (like a bitwise XOR of message blocks) can provide basic integrity checks against random errors, they are not secure against a malicious adversary. A **cryptographic hash function** must satisfy several strong security properties.

9.2 Security Properties of Cryptographic Hash Functions

A hash function is considered cryptographically secure if it is computationally infeasible to violate the following three properties:

9.2.1 Preimage Resistance (One-Way Property)

This property ensures that the function is "one-way". Given a hash value h, it should be computationally infeasible to find any input message m such that $H(m) = h$.

- **Formal Definition:** A function H is preimage resistant if, given a randomly chosen h, it is computationally infeasible to find an m such that $H(m) = h$.

This property is essential for applications like password storage.

9.2.2 Second Preimage Resistance (Weak Collision Resistance)

Given an input message m_1, it should be computationally infeasible to find a *different* input message m_2 that hashes to the same value.

- **Formal Definition:** A function H is second preimage resistant if, given an input m_1, it is computationally infeasible to find an $m_2 \neq m_1$ such that $H(m_2) = H(m_1)$.

This property prevents an attacker from substituting a different message for a given one without changing its hash value, which is crucial for digital signatures.

9.2.3 Collision Resistance (Strong Collision Resistance)

It should be computationally infeasible to find *any* two distinct inputs, m_1 and m_2, that hash to the same value.

- **Formal Definition:** A function H is collision resistant if it is computationally infeasible to find any pair (m_1, m_2) such that $m_1 \neq m_2$ and $H(m_1) = H(m_2)$.

This is the strongest of the three properties. If a hash function is collision resistant, it is also guaranteed to be second preimage resistant. Preimage resistance is also implied with high probability.

9.3 Attacks on Hash Functions

9.3.1 Brute-Force Attacks and the Birthday Paradox

An adversary can always attempt to break these properties by brute force (i.e., hashing random messages until a desired outcome is found). The security of a hash function against brute-force attacks depends on the length of its output, n.

- **Preimage and Second Preimage Attacks:** To find a (second) preimage for an n-bit hash, an attacker must, on average, compute 2^n hashes.
- **Collision Attacks:** To find a collision, an attacker needs significantly fewer operations. Due to a statistical phenomenon known as the **Birthday Paradox**, a collision can be found with high probability after computing only about $2^{n/2}$ hashes.

The Birthday Paradox states that in a group of just 23 people, there is a greater than 50% chance that at least two of them share the same birthday. This counter-intuitive result shows that collisions in a set of N possible outcomes occur much faster than one might expect. For a hash function with an n-bit output ($N = 2^n$ possible outcomes), a collision is likely to be found after about $\sqrt{N} = 2^{n/2}$ trials.

This vulnerability leads to the **Birthday Attack**, where an attacker generates many variations of a legitimate message and a fraudulent message. They then hash these variations until they find a pair (one legitimate, one fraudulent) that produces the same hash value. They can then have the legitimate message signed by an authority, and later attach that signature to the fraudulent message, which will be accepted as valid.

Because of the birthday attack, a hash function must have an output length large enough to make a $2^{n/2}$ attack infeasible. For this reason, 128-bit hashes like MD5 are considered completely insecure, and 160-bit hashes like SHA-1 are deprecated.

9.4 Constructing Hash Functions: The Merkle-Damgård Transform

Most traditional hash functions, including the SHA-1 and SHA-2 families, are built using the **Merkle-Damgård transform**. This is a general method for extending a fixed-length **compression function**, f, which is collision-resistant, into a hash function that accepts variable-length inputs.

The process is as follows:

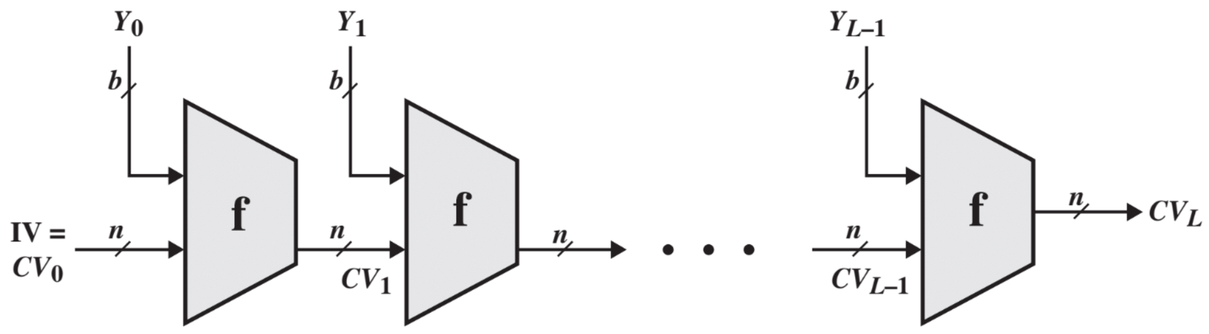
1. **Padding:** The input message m is padded so its length is a multiple of the block size b. Crucially, the length of the original message is also appended to the padded message. This step, known as MD-strengthening, is vital for the security of the construction.
2. **Iteration:** The padded message is divided into b-bit blocks $Y_0, Y_1, \dots, Y_{L-1}$. The hash is computed iteratively. A chaining variable CV, of size n (the hash output size), is maintained. It is initialized with a fixed public value, the IV (Initialization Vector).

$$\begin{aligned} CV_0 &= \text{IV} \\ CV_i &= f(CV_{i-1}, Y_{i-1}) \quad \text{for } i = 1, \dots, L \\ H(m) &= CV_L \end{aligned}$$

It can be proven that if the compression function f is collision-resistant, then the resulting hash function H constructed using the Merkle-Damgård transform is also collision-resistant.

9.5 The Secure Hash Algorithm (SHA) Family

SHA-1: An early standard producing a 160-bit hash. It is now considered broken and deprecated due to the discovery of practical collision attacks (complexity far below the theoretical 2^{80}).



IV	=	Initial value	L	=	number of input blocks
CV_i	=	chaining variable	n	=	length of hash code
Y_i	=	ith input block	b	=	length of input block
f	=	compression algorithm			

Figure 13: Merkle-Damgard transform.

SHA-2: A family of hash functions (SHA-224, SHA-256, SHA-384, SHA-512) that produce longer hash digests. They are based on the same Merkle-Damgård structure as SHA-1 but use a more complex and robust compression function. SHA-256 and SHA-512 are widely used today.

The compression function of SHA-512, for example, is a block cipher-like structure that runs for 80 rounds, using complex non-linear functions and additions modulo 2^{64} to provide strong confusion and diffusion.

9.6 SHA-3 and the Sponge Construction

Due to concerns that a common vulnerability might be found in the Merkle-Damgård design used by both SHA-1 and SHA-2, NIST held a competition for a new hash standard, SHA-3. The winning design, announced in 2012, is an algorithm named **Keccak**.

Keccak is based on a novel approach called the **Sponge Construction**. This is a versatile structure that can produce outputs of any desired length. It operates in two phases:

1. **Absorbing Phase:** The state (a large internal block of bits) is initialized to zero. The padded input message is divided into blocks. In each step, a message block is XORed into a portion of the state, and then a fixed, invertible permutation f is applied to the entire state.
2. **Squeezing Phase:** After all input blocks have been absorbed, the output hash is generated by extracting blocks from the same portion of the state, applying the permutation f between each extraction, until the desired number of output bits is produced.

The state is divided into two parts:

- **Bitrate (r):** The size of the message blocks that are XORed into the state. This part is also where output blocks are extracted.
- **Capacity (c):** The inner part of the state that is not directly affected by message blocks or exposed in the output. The security of the sponge construction is determined by the size of the capacity. For n bits of security against collisions, the capacity must be at least $2n$.

SHA-3 is an instance of Keccak. It uses a 1600-bit state and different values for bitrate and capacity to define the different versions (SHA3-224, SHA3-256, etc.). Because its internal structure is completely different from SHA-2, it provides a secure alternative in case any future weaknesses are found in the Merkle-Damgård design.

10 MAC Algorithms

This section explores practical constructions of Message Authentication Codes (MACs) using cryptographic primitives like block ciphers and hash functions. We will also cover modern Authenticated Encryption with Associated Data (AEAD) schemes that integrate confidentiality and integrity into a single algorithm.

10.1 MACs from Block Ciphers: CBC-MAC

A natural way to build a MAC is to use a block cipher, which is assumed to be a pseudorandom permutation (PRP). For a message m that fits into a single block, a secure MAC can be generated simply by encrypting the message with a secret key k:

$$t = E_k(m)$$

Since E_k is a PRP, its output is computationally indistinguishable from random, making the tag unpredictable for an adversary.

To extend this to variable-length messages, the **Cipher Block Chaining (CBC-MAC)** construction is used. It is similar to CBC mode encryption, but with two key differences:

1. The initialization vector (IV) is always a block of zeros.

2. The output tag is **only** the final block of the chain. The intermediate blocks are not part of the output.

$$t = \text{CBC-MAC}(k, m_1||m_2||\dots||m_N) = c_N$$

where $c_i = E_k(m_i \oplus c_{i-1})$ and $c_0 = \vec{0}$.

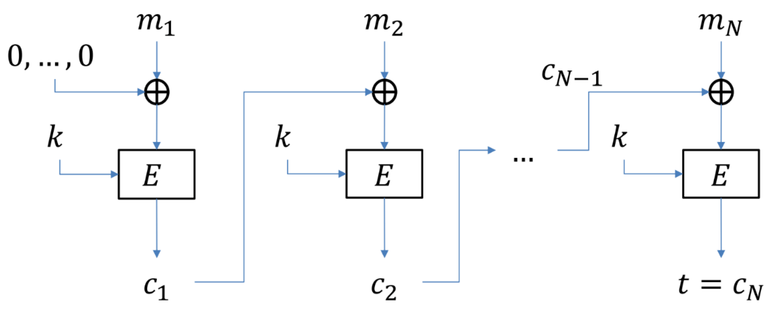


Figure 14: The CBC-MAC construction.

10.1.1 Limitations and Insecurity of Basic CBC-MAC

The basic CBC-MAC construction is only secure for messages of a **fixed, pre-determined length**. If an attacker can request tags for messages of different lengths, the scheme becomes insecure.

A simple **extension attack** is possible: if an adversary has the tag t for a one-block message m_1, they can easily forge the tag for the two-block message $m_1||(m_1 \oplus t)$. The tag for this new message will also be t, which is a valid forgery.

To fix this, several variants have been proposed:

- **CMAC (or OMAC):** Encrypts the final tag with a second key derived from the first. This prevents the extension attack and provides a secure MAC for variable-length messages.

- **CCM (Counter Mode with CBC-MAC):** A standardized mode that combines CBC-MAC for authentication with CTR mode for encryption.

10.2 MACs from Hash Functions

It is also common to construct MACs using cryptographic hash functions. Naive constructions, however, are often insecure. Consider these simple but flawed approaches:

- $t = H(k||m)$: This construction is vulnerable to a **length-extension attack** if the hash function H is based on the Merkle-Damgård transform (like SHA-2). Given $H(k||m)$, an attacker can compute $H(k||m||\text{pad}||m')$ for an extension m' without knowing the key k.

- $t = H(m||k)$: This is generally more secure, but can be vulnerable if collisions in the underlying hash function are found.

- $t = E_k(H(m))$: This decouples hashing and encryption. However, if collisions in H can be found (i.e., $H(m_1) = H(m_2)$), then the MACs for m_1 and m_2 will be identical, allowing for forgeries.

10.2.1 HMAC: Hash-based MAC

To overcome the vulnerabilities of naive constructions, the **HMAC** (Hash-based Message Authentication Code) standard was developed. HMAC uses a nested construction that is provably secure if the underlying hash function is collision-resistant.

The HMAC construction is:

$$t = H((k \oplus \text{opad})||H((k \oplus \text{ipad})||m))$$

where:

- H is a cryptographic hash function (e.g., SHA-256).
- k is the secret key, padded to the block size of the hash function.
- *ipad* (inner pad) is a constant string of repeated '0x36' bytes.
- *opad* (outer pad) is a constant string of repeated '0x5c' bytes.

This "hash-inside-a-hash" structure, combined with the two secret derived keys ($k \oplus \text{ipad}$ and $k \oplus \text{opad}$), effectively mitigates length-extension attacks and other weaknesses of the Merkle-Damgård design. HMAC is a widely used and trusted standard (FIPS 198).

10.2.2 KMAC: Keccak-based MAC

With the advent of SHA-3 (Keccak), a new MAC construction called **KMAC** was introduced. KMAC leverages the sponge construction of Keccak, which is not vulnerable to length-extension attacks. The construction is a variant of $H(k||m)$, but made secure by the properties of the sponge. The internal capacity of the sponge keeps part of the state hidden, preventing the attack. KMAC offers similar security to HMAC but can be more efficient in certain contexts.

10.3 Authenticated Encryption with Associated Data (AEAD)

Modern applications often require both confidentiality and integrity simultaneously. **Authenticated Encryption (AE)** schemes are algorithms that provide both in a single, integrated operation.

An **AEAD** scheme is an AE scheme that can also provide integrity and authenticity for some **Associated Data (AD)**. The associated data (e.g., a packet header or metadata) is not encrypted, but it is included in the integrity check. This ensures that an attacker cannot tamper with the unencrypted parts of a message without invalidating the tag.

10.3.1 CCM: Counter Mode with CBC-MAC

Defined in NIST SP 800-38C, CCM is a widely used AEAD mode, particularly in standards like IEEE 802.11i (WPA2) and IPSec. It is an *Authenticate-then-Encrypt* scheme that combines CBC-MAC and CTR mode encryption.

- A **nonce** (a number used once) is used to initialize both CBC-MAC and CTR mode. **The nonce must never be reused with the same key**.
- First, CBC-MAC is used to generate a tag over the associated data and the plaintext.
- Then, the plaintext and the tag are encrypted using CTR mode.

Because it is an AtE scheme, it requires careful implementation to avoid potential vulnerabilities like padding oracle attacks if error messages are not handled correctly.

10.3.2 GCM: Galois/Counter Mode

Defined in NIST SP 800-38D, GCM is a high-performance AEAD mode based on the *Encrypt-then-Authenticate* paradigm. It is the most popular AEAD scheme in use today, especially in TLS and high-speed network encryption. GCM combines two components:

1. **Encryption:** CTR mode is used for encryption, making it fast and parallelizable.
2. **Authentication:** A universal hash function called **GHASH** is used to generate a tag over the associated data and the ciphertext.

GHASH operates in the finite field GF(2^{128}) and involves multiplying the data blocks by a secret key H (derived from the main encryption key). The final tag is produced by encrypting the GHASH output with a fresh keystream block from CTR mode. This final encryption step makes the tag unpredictable. GCM's design allows for high throughput and low latency, making it ideal for performance-critical applications.

10.3.3 ChaCha20-Poly1305

This is another high-performance AEAD scheme, defined in RFC 8439 and widely used in TLS 1.3. It follows the Encrypt-then-Authenticate model.

- **Encryption:** The **ChaCha20** stream cipher is used for confidentiality.
- **Authentication: Poly1305**, a fast universal hash function, is used to compute a tag over the associated data and the ciphertext.

Poly1305 works by interpreting the message as coefficients of a polynomial and evaluating it at a secret key point modulo a large prime number ($2^{130} - 5$). The result is then added to a one-time secret key. It is designed to be extremely fast in software, making it a strong competitor to AES-GCM, especially on platforms without dedicated AES hardware support.

11 Public-Key Cryptography

11.1 The Key Distribution Problem

Private-key (or symmetric) cryptography is built on the assumption that the sender and receiver have already shared a secret key in a secure manner. This raises a critical question: how is this secret key distributed in the first place? This is known as the **key distribution problem**.

In some scenarios, a pre-existing secure channel might be available (e.g., diplomatic couriers, dedicated secure communication lines), but these solutions have severe limitations:

- **Scalability:** In a network of N users, a total of $N(N-1)/2$ unique keys are needed for every pair of users to communicate privately. This number grows quadratically, making key management infeasible for large networks like the internet.
- **Open Systems:** In open systems like the internet, it is impractical to establish a priori secure channels with every potential communication partner.
- **Key Distribution Centers (KDCs):** A trusted third party (KDC) can help by sharing keys only with individual users. However, the KDC itself becomes a single point of failure and a high-value target for attackers.

11.2 The Public-Key Revolution

In 1976, Whitfield Diffie and Martin Hellman published their groundbreaking paper, "New Directions in Cryptography," which introduced a radically new approach: **public-key cryptography**, also known as **asymmetric cryptography**.

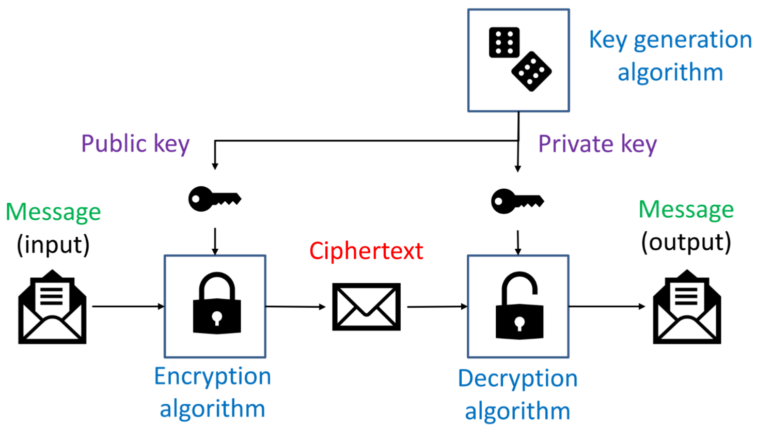


Figure 15: Public-key encryption framework.

The core idea is to use a pair of mathematically related keys for each user:

Public Key (pk): This key is made publicly available to everyone. Anyone can use it to encrypt messages intended for the key's owner.

Private Key (sk): This key is kept secret by the owner and is used to decrypt messages that have been encrypted with the corresponding public key.

11.2.1 The Public-Key Encryption Model

A public-key encryption scheme consists of three algorithms:

Key Generation Algorithm (Gen): A probabilistic algorithm that generates a pair of keys (pk, sk).

Encryption Algorithm (E): Takes the recipient's public key pk and a message m to produce a ciphertext $c = E(pk, m)$.

Decryption Algorithm (D): Takes the recipient's private key sk and a ciphertext c to recover the original message $m = D(sk, c)$.

The correctness property requires that for any key pair (pk, sk) generated by Gen, the following holds:

$$D(sk, E(pk, m)) = m$$

11.2.2 Core Requirements

For a public-key system to be secure and practical, it must satisfy several requirements:

1. All three algorithms (Gen, E, D) must be computationally easy (i.e., polynomial-time) to execute.
2. It must be computationally **infeasible** for an adversary to derive the private key sk from the public key pk.
3. It must be computationally **infeasible** for an adversary to recover the plaintext m from the ciphertext c and the public key pk.

11.2.3 Security and Probabilistic Encryption

Public-key encryption must be **probabilistic**. If it were deterministic, an adversary could simply encrypt candidate messages with the public key and compare the results to the target ciphertext.

Because the public key is known to everyone, the security game for public-key encryption is inherently a **Chosen-Plaintext Attack (CPA)**. Any adversary can use the public key to encrypt any message of their choice. Therefore, a secure public-key encryption scheme must have indistinguishable encryptions under chosen-plaintext attack (IND-CPA).

It is also important to note that a public-key system requires an **authenticated channel** to distribute public keys. A sender must be sure that the public key they are using truly belongs to the intended recipient. Otherwise, they are vulnerable to a **Man-in-the-Middle (MITM)** attack, where an attacker substitutes their own public key for the legitimate one.

11.3 One-Way and Trapdoor Functions

Public-key cryptography is made possible by the existence of mathematical functions that are easy to compute in one direction but hard to reverse.

One-Way Function: A function f is one-way if it is easy to compute $f(x)$ for any input x, but it is computationally infeasible to find x given only the output $f(x)$.

Trapdoor Permutation: A trapdoor permutation is a special type of one-way function that becomes easy to invert if one possesses a secret piece of information, called the **trapdoor**.

In public-key encryption, the encryption function $E(pk, \cdot)$ acts like a trapdoor permutation. The public key pk defines the function, which is easy to compute but hard to invert. The private key sk is the trapdoor that makes inversion (decryption) easy.

11.4 Applications of Public-Key Cryptography

Public-key algorithms are computationally much more expensive (slower) than symmetric-key algorithms. Therefore, they are not used to encrypt large amounts of data. Instead, they are used for specific tasks where a symmetric-key alternative is not available.

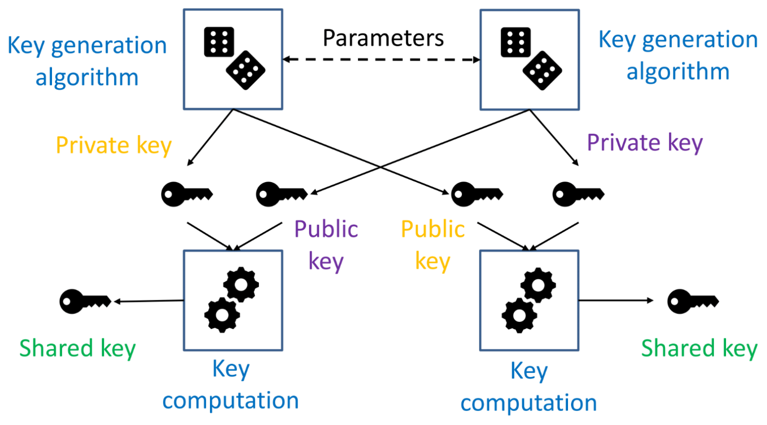


Figure 16: Key exchange framework.

11.4.1 Key Exchange

The most common use of public-key cryptography is to establish a shared secret key for symmetric encryption over an insecure channel. This is known as a **Key Exchange** protocol (e.g., Diffie-Hellman) or a **Key Encapsulation Mechanism (KEM)**. In a KEM, a sender encrypts a temporary, random symmetric key using the receiver's public key and sends it. The receiver decrypts it with their private key, and both parties can then communicate using a fast symmetric cipher.

11.4.2 Digital Signatures

Public-key cryptography also enables **digital signatures**, which provide authenticity, integrity, and non-repudiation. A digital signature scheme is the inverse of a public-key encryption scheme:

Signing Algorithm (S): The owner of the private key sk can create a signature s for a message m: $s = S(sk, m)$.

Verification Algorithm (V): Anyone with the corresponding public key pk can verify the signature on the message: $V(pk, m, s)$ outputs 1 if the signature is valid, and 0 otherwise.

Since only the owner knows the private key, only they can create a valid signature. This provides:

- **Integrity:** Any modification to the message will invalidate the signature.
- **Authenticity:** A valid signature proves the message was signed by the owner of the private key.
- **Non-repudiation:** The signer cannot later deny having signed the message.

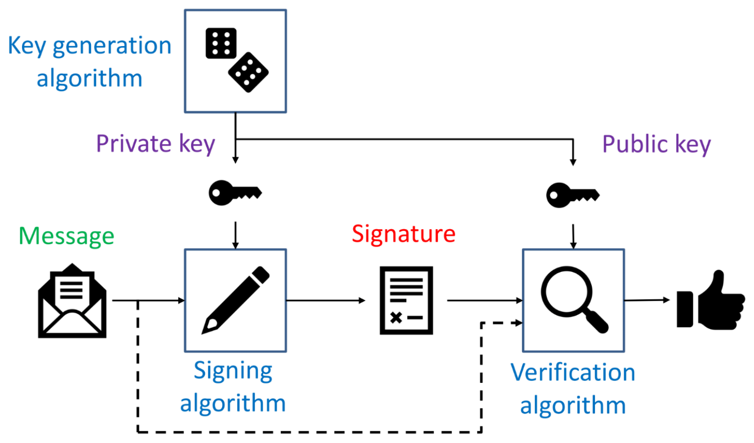


Figure 17: Digital signature framework.

It is a common myth that digital signatures are just public-key encryption with the keys reversed. While this is conceptually true for the RSA algorithm, secure digital signature schemes (like RSA-PSS or ECDSA) are specific algorithms distinct from their encryption counterparts.

12 Public-Key Algorithms: Factoring (RSA)

This section details the RSA algorithm, the first practical and widely used public-key cryptosystem. Its security is based on the presumed difficulty of the integer factorization problem.

12.1 Number Theory Preliminaries for RSA

The security and correctness of the RSA algorithm rely on several key theorems from number theory.

12.1.1 Fermat's Little Theorem

If p is a prime number, then for any integer a not divisible by p, it holds that:

$$a^{p-1} \equiv 1 \pmod{p}$$

An alternative form states that for any integer a and prime p:

$$a^p \equiv a \pmod{p}$$

12.1.2 Euler's Totient Function

Euler's totient function, denoted $\phi(n)$ (phi of n), is defined as the number of positive integers less than or equal to n that are relatively prime to n. Key properties of $\phi(n)$ include:

- For a prime number p, $\phi(p) = p - 1$.
- For two distinct prime numbers p and q, $\phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$.

The set of integers relatively prime to n forms a multiplicative group under multiplication modulo n, denoted $\mathbb{Z}_n^*$, which has order $\phi(n)$.

12.1.3 Euler's Theorem

Euler's theorem generalizes Fermat's Little Theorem. It states that for any two positive integers a and n that are relatively prime ($\gcd(a, n) = 1$):

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

This theorem is the mathematical foundation for the correctness of the RSA algorithm.

12.2 The RSA Algorithm

Developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman, RSA is a complete public-key system providing both encryption and digital signatures.

12.2.1 Key Generation

Each user generates a public/private key pair as follows:

1. **Select two large, distinct prime numbers**, p and q. For security, these must be chosen randomly and kept secret.
2. **Compute the modulus** $N = pq$. The bit length of N (e.g., 2048 bits) is the key length.
3. **Compute Euler's totient function** for N: $\phi(N) = (p-1)(q-1)$.
4. **Choose the public exponent** e, an integer such that $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$. Common choices for e are small primes like 65537 for efficiency.
5. **Compute the private exponent** d, which is the modular multiplicative inverse of e modulo $\phi(N)$.

   $$d \equiv e^{-1} \pmod{\phi(N)}$$

   This can be efficiently computed using the Extended Euclidean Algorithm.
6. **Publish the public key** $pk = (N, e)$.
7. **Keep the private key secret** $sk = (N, d)$. The primes p and q must also be kept secret.

12.2.2 Encryption and Decryption

Encryption: To encrypt a message m (represented as an integer where $0 \leq m < N$), compute the ciphertext c using the recipient's public key (N, e):

$$c = m^e \pmod N$$

Decryption: To decrypt the ciphertext c, the recipient uses their private key (N, d):

$$m = c^d \pmod N$$

Both operations are performed using an efficient modular exponentiation algorithm (e.g., square-and-multiply).

12.2.3 Correctness of RSA

The decryption process works because of Euler's theorem. Since $d \equiv e^{-1} \pmod{\phi(N)}$, we know that $ed = 1 + k\phi(N)$ for some integer k. Therefore:

$$c^d \equiv (m^e)^d \equiv m^{ed} \equiv m^{1+k\phi(N)} \equiv m \cdot (m^{\phi(N)})^k \pmod N$$

If m is relatively prime to N, by Euler's theorem, $m^{\phi(N)} \equiv 1 \pmod N$. This simplifies to:

$$m \cdot (1)^k \equiv m \pmod N$$

The proof can be extended to all $m < N$.

12.3 Security and Practical Considerations

12.3.1 RSA Assumption and Factoring

The security of RSA relies on the **RSA assumption**: given a public key (N, e) and a ciphertext $c = m^e \pmod N$, it is computationally infeasible to find m. This is also known as the RSA problem. This problem is closely related to the **integer factorization problem**. If an adversary could factor the modulus N into its prime factors p and q, they could compute $\phi(N) = (p-1)(q-1)$ and then derive the private key d from the public key e. Currently, the best-known algorithms for factoring large numbers are sub-exponential and are too slow to be practical for the key sizes used today (e.g., 2048 bits or more).

12.3.2 RSA and CPA Security: The Need for Padding

The basic (”plain”) RSA algorithm described above is **deterministic**. Encrypting the same message m always results in the same ciphertext c. As a result, plain RSA is **not CPA-secure**. To achieve CPA security, the message must be randomly padded before encryption. Standards like **Optimal Asymmetric Encryption Padding (OAEP)** are used in practice. OAEP is a randomized and reversible padding scheme that makes RSA probabilistic and resistant to chosen-plaintext attacks.

12.3.3 Attacks on RSA Implementations

In addition to factoring, RSA implementations can be vulnerable to other attacks:

- **Small Public Exponent Attacks:** If a small value of e (like $e = 3$) is used and the message m is also small, m^e might be less than N. In this case, $c = m^e$, and an attacker can recover m by simply taking the integer e-th root of c. Padding schemes prevent this.
- **Chosen-Ciphertext Attacks (CCA):** Plain RSA has a homomorphic property for multiplication ($E(m_1) \cdot E(m_2) = E(m_1 \cdot m_2)$), which can be exploited in a CCA. Even padded schemes like the older PKCS#1 v1.5 standard are vulnerable to sophisticated padding oracle attacks (e.g., Bleichenbacher's attack), which is why modern systems use OAEP or implement careful countermeasures.
- **Side-Channel Attacks:** Attacks like **timing attacks** can exploit variations in the time it takes to perform decryption to leak information about the private key d. Countermeasures include constant-time modular exponentiation and ciphertext blinding.

12.4 Practical Security and Key Lengths

The "security strength" of a cryptosystem is measured in bits. An n-bit security level means an attacker would need to perform about 2^n operations to break the system.

- For **symmetric ciphers** like AES, the security strength is equal to the key length (e.g., AES-128 provides 128-bit security).
- For **public-key systems**, the security is determined by the best-known algorithm for solving their underlying mathematical problem.

The best algorithm for factoring (and thus breaking RSA) is the General Number Field Sieve (GNFS), which has a sub-exponential complexity. Because of this, RSA requires much larger keys than symmetric ciphers to achieve the same level of security.

The following table from NIST shows comparable key sizes for different systems:

Security Strength	Symmetric (AES)	Factoring (RSA)
128 bits	128-bit key	3072-bit modulus
192 bits	192-bit key	7680-bit modulus
256 bits	256-bit key	15360-bit modulus

Table 2: NIST Recommended Key Sizes for Equivalent Security

This illustrates why public-key encryption is not a replacement for symmetric-key encryption but is used strategically for tasks like key exchange and digital signatures.

13 Digital Signatures

While public-key cryptography addresses the issues of confidentiality and key distribution, **Digital Signatures** tackle the problems of **authenticity**, **integrity**, and, crucially, **non-repudiation**. They can be viewed as the public-key equivalent of MACs (Message Authentication Codes), but with significantly stronger properties.

13.1 Limitations of MACs and the Need for Digital Signatures

We have seen that a MAC guarantees a message originates from someone possessing the shared secret key k. However, in a symmetric context, both the sender (Alice) and the receiver (Bob) know k. This creates two fundamental problems that MACs cannot solve:

1. **Receiver Forgery:** If Bob receives a message from Alice with a valid MAC, Bob himself could have generated that message and MAC, since he possesses k. Therefore, he cannot prove to a third party (e.g., a judge) that the message actually came from Alice.
2. **Sender Repudiation:** Alice could send an order ("Buy 1000 shares") and later deny having done so. Since Bob also possesses the key to generate the MAC, Alice can claim that Bob forged the order.

Digital Signatures resolve these issues by using an asymmetric key pair:

- **Private Key (sk):** Used to *sign*. Only the signer possesses it.
- **Public Key (pk):** Used to *verify*. Anyone can verify the signature.

13.2 Properties and Requirements

A secure digital signature must satisfy three main properties:

1. **Public Verifiability:** Anyone holding the signer's public key must be able to verify the validity of the signature.
2. **Transferability:** The signature can be copied and shown to third parties while maintaining its validity (essential for contracts or certificates).
3. **Non-repudiation:** The signer cannot deny having signed the message, as they are the sole possessor of the private key required to generate that specific signature.

From a computational perspective, the requirements are:

- It must be easy to generate the signature knowing sk.
- It must be easy to verify the signature knowing pk.
- It must be **computationally infeasible** to forge the signature (i.e., generate it without knowing sk) or to find a new message for which an existing signature is valid.

13.3 Security Models for Signatures

To formally define security, we consider the "signature game" between a challenger and an adversary.

13.3.1 Attack Scenarios

The adversary may have different levels of access:

- **Key-only attack:** The adversary knows only the public key pk.
- **Known-message attack:** The adversary has access to a list of valid message-signature pairs (m, s).
- **Adaptive Chosen-Message Attack (CMA):** This is the strongest scenario. The adversary can ask the oracle to sign any message of their choice, adapting their requests based on previous answers.

13.3.2 Types of Forgery

The adversary's goal is to create a forgery. The severity varies:

- **Total break:** The adversary recovers the private key sk.
- **Selective forgery:** The adversary succeeds in signing a specific message of their choice.
- **Existential forgery:** The adversary succeeds in creating a valid pair (m, s) for *at least one* message m, even if m is nonsensical or not chosen by them.

Security Definition: A digital signature scheme is considered secure (**EUF-CMA**: Existential Unforgeability under Chosen-Message Attack) if the probability that an adversary succeeds in committing even just an *existential forgery* is negligible ($negl(n)$), even after requesting signatures for many messages of their choice.

13.4 RSA-based Signature Algorithms

13.4.1 Plain RSA Signature

The basic idea is the inverse of RSA encryption.

- **Signature:** Given a message $m \in \mathbb{Z}_N$, the signature is $s = m^d \pmod N$.
- **Verification:** Calculate $m' = s^e \pmod N$ and check if $m' = m$.

Weaknesses: "Plain RSA" is insecure.

1. **Trivial Existential Forgery:** An attacker can choose a random signature s and calculate the corresponding message $m = s^e \pmod N$. The pair (m, s) is valid, even if m might be nonsense (violating the EUF-CMA definition).
2. **Multiplicative Property:** If the attacker has signatures for m_1 (s_1) and m_2 (s_2), they can calculate a valid signature for $m_3 = m_1 \cdot m_2$ simply by computing $s_3 = s_1 \cdot s_2 \pmod N$.

13.4.2 RSA-FDH (Full Domain Hash)

To make RSA secure, the raw message is never signed; instead, its hash is signed.

- **Signature:** $s = H(m)^d \pmod N$.
- **Verification:** Check if $s^e \equiv H(m) \pmod N$.

Using a cryptographic hash function H breaks the algebraic structure (preventing the multiplicative property) and, since H is preimage resistant, prevents trivial existential forgery. If H is modeled as a random oracle, RSA-FDH is provably secure.

13.4.3 RSA-PSS (Probabilistic Signature Scheme)

RSA-FDH is deterministic (same message = same signature). The current standard (FIPS 186-4) recommends RSA-PSS, which introduces probability. Before signing, the message m is processed:

1. Calculate the hash of m.
2. Concatenate a random value (**salt**).
3. Apply a masking function (MGF).

The result (Encoded Message, EM) is then signed with the RSA private key. The *salt* ensures that signing the same message twice produces different signatures, strengthening security.

13.5 Discrete Logarithm-based Schemes

Besides RSA, many signature schemes rely on the hardness of the Discrete Logarithm (DL) problem. These schemes often derive from *interactive identification protocols* transformed into digital signatures.

13.5.1 Schnorr Identification Scheme

Imagine Alice (Prover) wants to prove to Bob (Verifier) that she knows the discrete logarithm x of her public key $y = g^x$, without revealing x.

1. **Commitment:** Alice chooses a random number k and sends $I = g^k$ to Bob.
2. **Challenge:** Bob sends a random challenge r to Alice.
3. **Response:** Alice calculates $s = rx + k \pmod q$ and sends it to Bob.
4. **Verification:** Bob checks if $g^s y^{-r} = I$.

The idea is that Alice must know x to correctly answer the challenge r consistently with the value I committed at the beginning.

13.5.2 Fiat-Shamir Transformation (From Identification to Signature)

To transform the above interactive scheme into a digital signature (non-interactive), Alice cannot wait for Bob to send the challenge r. Alice uses a hash function to calculate the challenge "by herself", basing it on the message and the commitment.

- **Non-interactive Challenge:** $r = H(g^k || m)$.

The signature becomes the pair (r, s). Anyone can verify it by recomputing I from public values and checking the hash. This is the **Schnorr Signature Scheme**.

13.5.3 DSA (Digital Signature Algorithm)

DSA is a NIST standard based on a variant of the Schnorr scheme (originally designed to avoid patents).

- **Generation:** A random number k (nonce) is used for each signature.
- **Signature:**
  1. Calculate $r = (g^k \pmod p) \pmod q$.
  2. Calculate $s = k^{-1}(H(m) + xr) \pmod q$.
- **Verification:** Complex calculations to verify that r corresponds.

Importance of the nonce k: In DSA (as well as Schnorr and ECDSA), the value k must be **absolutely random, secret, and unique** for each message. If k is reused for two different messages m_1 and m_2, an attacker can solve a simple system of linear equations and **recover the private key** x. This was the famous attack vector in the Sony PlayStation 3 hack (Sony used a static k).

13.5.4 ECDSA (Elliptic Curve DSA)

ECDSA is simply the application of the DSA algorithm on a group defined by an elliptic curve instead of $\mathbb{Z}_p^*$.

- It guarantees the same security as DSA/RSA but with much shorter keys (e.g., 256-bit ECDSA $\approx$ 3072-bit RSA), making it ideal for mobile devices and smart cards.
- Here too, the management of the nonce k is critical.

14 Key Management

Cryptographic Key Management is the backbone of any secure system. It encompasses the entire lifecycle of cryptographic keys, including their generation, distribution, storage, replacement, and usage. Even the strongest algorithm (like AES or RSA) is useless if the key is generated poorly (e.g., it's predictable) or stolen during distribution.

14.1 Random Number Generation

The foundation of key generation is randomness. If an attacker can predict the random numbers used to generate a key, they can reproduce the key.

14.1.1 TRNG vs. PRNG

We distinguish between two main types of generators:

- **TRNG (True Random Number Generator):** Relies on physical, non-deterministic phenomena (entropy sources) such as thermal noise, keystroke timing, or disk activity. It produces truly unpredictable bits but is often slow. It is typically used to generate the *seed*.
- **PRNG (Pseudorandom Number Generator):** A deterministic algorithm that takes a short, random *seed* (from a TRNG) and expands it into a long stream of bits that *appears* random. It is fast and widely used for generating session keys or nonces, but its security depends entirely on the secrecy and randomness of the seed.

Real-world impact: Many IoT devices have been compromised because they generated RSA keys using poor entropy (low randomness) right after booting up, leading to predictable keys that attackers could easily factor.

14.2 Key Derivation Functions (KDF)

Often, we start with a value that has some entropy but isn't a uniform bit string (like a password or a Diffie-Hellman shared secret). A **KDF** derives a strong, cryptographically suitable key from this source material.

14.2.1 HKDF (HMAC-based KDF)

A standard and robust KDF is HKDF (RFC 5869), which operates in two steps:

1. **Extract:** Compresses the input keying material (IKM) into a fixed-length pseudorandom key (PRK) using a salt. This "concentrates" the entropy.
2. **Expand:** Generates one or more output keys from the PRK using a specific info string (to bind the key to a context) and a counter.

14.3 Key Distribution

This is the problem of how to securely deliver keys to communicating parties.

14.3.1 Symmetric Key Distribution

Sharing a secret key is challenging. Solutions include:

- **Physical delivery:** Secure but not scalable.
- **Key Distribution Center (KDC):** A trusted third party shares a *master key* with each user. When Alice wants to talk to Bob, the KDC generates a temporary *session key*, encrypts it separately for Alice (using Alice's master key) and for Bob (using Bob's master key), and distributes it. Kerberos is a classic example.

14.3.2 Asymmetric Key Distribution & KEM

Public-key cryptography simplifies distribution (no need to share a secret in advance), but we still need to establish a shared symmetric key for performance. This is often done via a **Key Encapsulation Mechanism (KEM)**:

1. Alice generates a random symmetric key k and encrypts (encapsulates) it using Bob's public key.
2. Bob decrypts (decapsulates) it using his private key.
3. Both now share k.

14.4 Public Key Infrastructure (PKI) and Certificates

The main vulnerability in public-key distribution is the Man-in-the-Middle (MitM) attack. If an attacker gives you their public key pretending it belongs to Bob, you will encrypt data for the attacker. To prevent this, we need to guarantee the **authenticity** of public keys.

14.4.1 Digital Certificates (X.509)

A digital certificate binds an identity (e.g., "www.google.com" or "Alice") to a public key. It acts like a digital passport signed by a trusted third party called a **Certification Authority (CA)**. An X.509 certificate contains:

- The subject's Public Key.
- The subject's Identity (Distinguished Name).
- Validity period (Not Before, Not After).
- **CA's Digital Signature:** This proves the certificate is authentic.

14.4.2 Trust Hierarchy and Chains

Your browser/OS trusts a small set of **Root CAs** (pre-installed).

- **Root CA:** Signs certificates for Intermediate CAs.
- **Intermediate CA:** Signs certificates for end entities (websites, users).

To verify a website's certificate, the browser follows the **chain of trust** up to a known Root CA. If the signature chain is valid and the Root is trusted, the public key is accepted.

14.4.3 Revocation

If a private key is stolen, the corresponding certificate must be revoked before it expires.

- **CRL (Certificate Revocation List):** A list of revoked serial numbers signed by the CA. It can be large and slow to update.
- **OCSP (Online Certificate Status Protocol):** Allows querying the CA in real-time to check the status of a specific certificate.

14.5 Key Protection

Finally, private keys must be stored securely. Software storage is vulnerable to malware. Secure solutions involve hardware:

- **Smart Cards / HSM (Hardware Security Module):** The key never leaves the device. Cryptographic operations are performed inside the hardware.
- **TPM (Trusted Platform Module):** A dedicated chip on the motherboard to secure keys and verify system integrity.

15 Transport Layer Security (TLS)

Transport Layer Security (TLS) is the most widely deployed security protocol in the world. It sits between the Application Layer (HTTP, SMTP, IMAP) and the Transport Layer (TCP), providing a secure channel over an insecure network. It evolved from the older SSL (Secure Sockets Layer) protocol, which is now deprecated.

The primary goal of TLS is to provide:

- **Confidentiality:** Prevents eavesdropping via symmetric encryption.
- **Integrity:** Prevents tampering via MACs or Authenticated Encryption.
- **Authentication:** Verifies the identity of the server (and optionally the client) using digital certificates.

15.1 TLS Architecture

TLS is designed as a layered protocol consisting of two main blocks:

1. **Record Protocol (Lower Layer):** It wraps application data. It handles fragmentation, compression (deprecated in 1.3), authentication (MAC), and encryption. It uses the symmetric keys established during the handshake.
2. **Handshake Protocol (Upper Layer):** It is responsible for negotiating security parameters, authenticating the server, and establishing the shared secrets.

15.1.1 Sessions vs. Connections

To improve performance, TLS distinguishes between a *session* and a *connection*:

- **Session:** An association between a client and a server created by a full handshake. It defines a set of cryptographic parameters and a **Master Secret**. Creating a session is computationally expensive (public key crypto).
- **Connection:** A transient communication channel. Multiple connections can be derived from a single session using a cheap "resumption" mechanism, avoiding the heavy cost of a full handshake.

15.2 The TLS 1.2 Handshake

The handshake is the most critical part of TLS. In version 1.2, it takes 2 round-trips (2-RTT) to establish a connection:

1. **Negotiation (Phase 1):** The Client sends a `ClientHello` with a list of supported *Cipher Suites* and a random number (R_C). The Server picks the best cipher suite and replies with `ServerHello` and its own random number (R_S).
2. **Authentication (Phase 2):** The Server sends its **Certificate** (containing its public key).
3. **Key Exchange (Phase 2 & 3):**
   - In **RSA key exchange**: The client generates a *Pre-Master Secret*, encrypts it with the server's public key, and sends it.
   - In **Diffie-Hellman (DH)**: The server sends its DH parameters signed with its private key. The client sends its DH public share.
4. **Completion (Phase 4):** Both parties derive the **Master Secret** from the Pre-Master Secret and the randoms (R_C, R_S). They switch to encryption mode and verify the handshake integrity.

15.2.1 Cipher Suites

A Cipher Suite is a string that defines the toolkit for the connection. Example: `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`.

- **ECDHE:** Key Exchange (Elliptic Curve Diffie-Hellman Ephemeral).
- **ECDSA:** Authentication (Elliptic Curve Digital Signature).
- **AES_256_GCM:** Symmetric Encryption (Authenticated Encryption).
- **SHA384:** Hashing algorithm for the PRF (Key Derivation).

15.3 TLS 1.3: A Major Overhaul

TLS 1.2 had several issues: it was slow (2-RTT), complex, and supported weak algorithms that led to attacks like POODLE, CRIME, and ROBOT. TLS 1.3 (RFC 8446) is a radical redesign aimed at security and speed.

15.3.1 Key Changes in TLS 1.3

- **Removed Weak Features:** No more RSA Key Exchange (no forward secrecy), no CBC mode (padding oracles), no SHA-1, no RC4.
- **Forward Secrecy Mandatory:** Only Ephemeral Diffie-Hellman (DHE/ECDHE) is allowed. Static RSA is gone.
- **AEAD Only:** Only Authenticated Encryption (like AES-GCM or ChaCha20-Poly1305) is supported.

15.3.2 The 1-RTT Handshake

TLS 1.3 is faster. The client "guesses" the key exchange parameters and sends its DH key share *immediately* in the `ClientHello` message.

- **Client:** Sends `ClientHello` + `KeyShare` (public key).
- **Server:** Generates its key share, computes the shared secret, encrypts its Certificate, and finishes the handshake.

This allows the client to start sending encrypted application data after just 1 round trip.

15.4 HTTPS (HTTP over TLS)

HTTPS is simply the HTTP protocol layered on top of TLS.

- **Trust:** The browser verifies that the server's certificate is signed by a trusted Root CA and matches the domain name in the URL (e.g., `www.polito.it`).
- **Encryption:** All data (URL path, headers, cookies, content) is encrypted. An eavesdropper can only see the IP address and port (and the domain name via the SNI extension in cleartext during the handshake).

16 Quantum-Safe Cryptography

The final frontier of modern cryptography is the transition to algorithms that can withstand the power of quantum computers. This field is known as **Post-Quantum Cryptography (PQC)** or Quantum-Safe Cryptography.

16.1 The Quantum Threat

Quantum computers operate on qubits, allowing them to exist in superpositions of states. While they are not "faster" at everything, they can solve specific mathematical problems exponentially faster than classical computers. This poses a catastrophic threat to the cryptographic infrastructure we have built over the last 40 years.

16.1.1 Grover's Algorithm (The Symmetric Threat)

Grover's algorithm provides a quadratic speedup for searching unsorted databases.

- **Impact:** It reduces the effective security of symmetric keys and hash functions by half. To find a n-bit key, it takes $O(2^{n/2})$ operations instead of $O(2^n)$.
- **Mitigation:** This is manageable. We simply need to **double the key lengths**. AES-128 is broken, but AES-256 remains secure. Similarly, SHA-256 provides only 128 bits of quantum security, so we move to SHA-384 or SHA-512.

16.1.2 Shor's Algorithm (The Asymmetric Threat)

Shor's algorithm is much more dangerous. It can find the prime factors of an integer N and solve the Discrete Logarithm Problem in polynomial time.

- **Impact:** It completely breaks **RSA**, **Diffie-Hellman**, and **Elliptic Curve Cryptography (ECC)**. If a sufficiently powerful quantum computer is built, all current public-key standards (TLS certificates, digital signatures, VPN handshakes) will become insecure immediately.
- **Mitigation:** We cannot just increase key sizes (keys would become impossibly large). We must replace the algorithms entirely with new mathematical problems that are hard even for quantum computers.

16.2 Why Migrate Now? "Harvest Now, Decrypt Later"

Even though powerful quantum computers do not exist yet, the migration is urgent due to the "Harvest Now, Decrypt Later" attack strategy. Adversaries can record encrypted traffic today (e.g., VPN sessions, government communications) and store it. Once a quantum computer becomes available (potentially in 10-15 years), they can retroactively decrypt all the stored data. For long-term secrets, the time to migrate is yesterday.

16.3 Quantum-Safe Families

The cryptographic community, led by NIST, has identified five main families of math problems believed to be quantum-resistant:

1. **Lattice-based:** Relies on geometric problems in high-dimensional grids (lattices). It is the most promising and efficient family (e.g., Kyber, Dilithium).
2. **Code-based:** Relies on the hardness of decoding random linear codes with errors (e.g., McEliece). Very mature but has large public keys.
3. **Multivariate:** Relies on solving systems of multivariate quadratic equations. Good for signatures but not encryption.
4. **Hash-based:** Relies only on the security of hash functions (e.g., SPHINCS+). Very conservative security guarantees but slower and larger signatures.
5. **Isogeny-based:** Relies on walking between elliptic curves via isogenies. It offered very small keys, but a major algorithm (SIKE) was recently completely broken by a classical attack, casting doubt on this family.

16.4 Lattice-Based Cryptography

The NIST standards are dominated by lattice-based schemes.

16.4.1 Lattices and Hard Problems

A lattice is a grid of points in n-dimensional space. The security relies on problems like:

- **Shortest Vector Problem (SVP):** Given a basis for a lattice, find the shortest non-zero vector in the grid. In high dimensions ($n > 500$), this is incredibly hard.
- **Closest Vector Problem (CVP):** Given a target point in space (not on the lattice), find the lattice point closest to it.

16.4.2 Learning With Errors (LWE)

Most modern schemes use a variant called Learning With Errors. The problem is to solve a linear system $A^T s \approx t$ where we introduce small random errors ("noise").

- Without the error, solving for s is trivial (Gaussian elimination).
- With the error, it becomes computationally hard, equivalent to solving hard lattice problems.

16.5 NIST Standardization

After a multi-year competition, NIST selected the first PQC standards in 2022 (finalized in 2024).

16.5.1 Selected Algorithms

- **ML-KEM (formerly CRYSTALS-Kyber):** The standard for **Key Encapsulation**. It is lattice-based. It is fast and has small keys. It will replace ECDH in TLS 1.3.
- **ML-DSA (formerly CRYSTALS-Dilithium):** The primary standard for **Digital Signatures**. It is lattice-based. It will replace RSA/ECDSA in certificates.
- **SLH-DSA (formerly SPHINCS+):** A stateless hash-based signature scheme. It is slower but serves as a conservative backup in case lattice problems are broken.

16.5.2 Migration Strategy

The migration will likely follow a **Hybrid Approach**: combining a classical algorithm (like ECDH) with a post-quantum one (like Kyber). This ensures that communication remains secure against classical attacks even if the new quantum-safe algorithm turns out to have a flaw.

17 User Authentication

While encryption secures data in transit and at rest, **User Authentication** is the process of verifying the identity of a human (or entity) interacting with a system. It answers the question: "Is this user really who they claim to be?"

Authentication is distinct from Identification:

- **Identification:** The user claims an identity (e.g., typing a username). This is a 1-to-N search problem.
- **Verification (Authentication):** The user proves that identity (e.g., typing a password). This is a 1-to-1 comparison problem against stored credentials.

17.1 Authentication Factors

NIST classifies authentication methods into three fundamental factors (the "Something you..." paradigm):

1. **Knowledge Factor ("Something you know"):** Information the user remembers, such as a Password, PIN, or the answer to a security question.
   - *Pros:* Easy to implement, portable.
   - *Cons:* Can be guessed, forgotten, shared, or stolen via phishing.
2. **Possession Factor ("Something you have"):** A physical item the user possesses, such as a Smart Card, a Hardware Token (e.g., RSA SecurID), or a Smartphone.
   - *Pros:* Harder to duplicate remotely.
   - *Cons:* Can be lost, stolen, or broken.
3. **Inherence Factor ("Something you are"):** Biometric traits inherent to the user, such as Fingerprint, Face recognition, or Iris scan.
   - *Pros:* Cannot be lost or forgotten, very hard to share.
   - *Cons:* Probabilistic (false positives/negatives), privacy concerns, cannot be "reset" if compromised.

Multi-Factor Authentication (MFA) combines two or more different factors (e.g., Password + Token) to significantly increase security. If one factor is compromised, the attacker still cannot access the account.

17.2 Password-Based Authentication

Passwords are the most common form of authentication, but they are plagued by security issues: users choose weak passwords, reuse them across sites, and fall for phishing attacks.

17.2.1 Secure Password Storage

A server must never store passwords in plaintext. If the database is breached, all accounts are compromised immediately.

1. **Hashing:** The server stores $H(password)$. When a user logs in, the server hashes the input and compares it to the stored hash. Because cryptographic hash functions are one-way (preimage resistant), an attacker cannot easily reverse the hash to find the password.
2. **Salting:** Storing only simple hashes is vulnerable to *Dictionary Attacks* (pre-computing hashes for common passwords). To fix this, we generate a random string called a **Salt** for each user. We store $(ID, Salt, H(password||Salt))$.
    - The salt ensures that two users with the same password have different stored hashes.
    - It prevents the use of pre-computed Rainbow Tables, forcing the attacker to brute-force each account individually.

17.3 One-Time Passwords (OTP)

To mitigate the risk of password theft and replay attacks, systems use dynamic passwords that are valid for only one session.

17.3.1 HOTP (HMAC-based OTP)

Defined in RFC 4226. It relies on a shared counter C and a secret key K.

- The token and the server increment the counter synchronously.
- $OTP = HMAC(K, C)$.
- This is often used in hardware tokens where the user presses a button to generate the next code.

17.3.2 TOTP (Time-based OTP)

Defined in RFC 6238. It is a variant of HOTP where the "counter" is the current time.

- $C = \lfloor (CurrentTime - T_0)/30 \rfloor$.
- The code changes every 30 seconds.
- This is the standard used by apps like Google Authenticator or Microsoft Authenticator. It requires the device and server clocks to be roughly synchronized.

17.4 Challenge/Response Protocols

Sending a password (even encrypted) over the network is risky. Challenge/Response protocols prove identity without revealing the secret.

1. **Challenge:** The Verifier sends a unique, random value (nonce) to the User.
2. **Response:** The User performs a cryptographic operation on the nonce using their secret and sends the result back.
3. **Verification:** The Verifier checks the result.

17.4.1 Symmetric vs. Asymmetric C/R

- **Symmetric:** The user and server share a secret key K. The user computes $E(nonce, K)$. *Risk:* The server is a "honey pot" of secret keys. If the server is hacked, all user keys are stolen.
- **Asymmetric (Public Key):** The user has a key pair (sk, pk). The server registers only the public key pk. The user signs the challenge with sk. *Benefit:* The server stores no secrets. If the server DB is leaked, the attacker gets only public keys, which cannot be used to impersonate users.

17.5 FIDO and WebAuthn (Passwordless Future)

FIDO (Fast IDentity Online) is a set of open standards (FIDO2, UAF, U2F) designed to eliminate passwords and solve the phishing problem using public-key cryptography.

17.5.1 How FIDO Works

Instead of a password, the user has an **Authenticator**. This can be a roaming hardware key (like a YubiKey) or a platform authenticator (TouchID/FaceID on a phone/laptop).

1. **Registration:** The device generates a new public/private key pair for the specific website. The private key is stored securely in the device's hardware (TPM/Secure Enclave) and never leaves it. The public key is sent to the server.
2. **Authentication:**
    - The server sends a random challenge.
    - The browser/OS verifies the domain name (preventing phishing).
    - The user unlocks the private key locally (e.g., via fingerprint or PIN).
    - The device signs the challenge and sends the signature to the server.

17.5.2 WebAuthn

WebAuthn is the standard browser API that allows web applications to use FIDO authenticators. It enables standard websites to request strong, hardware-backed public-key authentication.

17.5.3 Passkeys

Standard FIDO credentials are bound to a specific hardware device. If you lose the device, you lose access. **Passkeys** solve this by allowing FIDO credentials (private keys) to be synced securely across a user's devices via the cloud (e.g., iCloud Keychain, Google Password Manager). This makes passwordless authentication usable for the general public, offering high security (phishing resistance) with high usability (biometric login across all devices).