Security & Cryptography Class Notes

Anna Visman

Academic Year 2024-2025

Contents

1 Lecture 1

1.1 Security Overview

A computer system is said to be secure if it satisfies the following properties:

- **Confidentiality**: Unauthorized entities cannot access the system or its data
- **Integrity**: When you receive data, it is the right one
- **Availability**: The system or data is there when you need it

Remark 1. *The mere presence of these properties does not necessarily mean that the system is fully secure in practice.*

A secure system is reliable:

- Keep your personal data confidential
- Allow only authorised access or modifications to resources
- Ensure that any produced results are correct
- Give you correct and meaningful results whenever you want them

Terminology:

- **Assets**: Things we want to protect (hardware, software, data)
- **Vulnerabilities**: Weaknesses in a system that may be exploited in order to cause loss and harm
- **Threats**: A loss or harm that might befall a system (interception, interruption, modification, fabrication)
- **Attack**: An action which exploits a vulnerability to execute a threat
- **Control/Defence**: Removing/reducing a vulnerability. You control a vulnerability to prevent an attack and defend against a threat

Methods of Defence:

- Prevent it
- Deter it: make the attack harder or more expensive
- Deflect it: make yourself less attractive to attacker
- Detect it: notice that the attack is occurring
- Recover from it: mitigate the effects of the attack

Principle of Easiest Penetration: A system is only as secure as its weakest link. An attacker will go after whatever part of the system is easiest for them, not most convenient for you. In order to build secure systems, we need to learn how to think like an attacker!

1.2 Defense Overview

Software controls:

- Passwords and other forms of access control
- Operating systems separate users' actions from each other
- Virus scanner watch for malware
- Development controls enforce quality measures on the original source code
- Personal firewalls that run on your desktop

Hardware controls:

- Not usually protection of the hardware itself, but rather using separate hardware to protect the system as a whole
- Fingerprint readers
- Smart tokens
- Firewalls
- Intrusion detection systems

Physical Systems:

- Protection of the hardwell itself, as well as physical access to the console, storage media, etc.
- Locks
- Guards
- Off-site backups

Policies and Procedures:

- Non-technical means can be used to protect against some classes of attack (e.g. VPNs for accessing interal company network)
- Rules about choosing Passwords
- Training in best security practices

1.3 Cryptography Overview

Objectives of Cryptography:

- Protecting data privacy
- Authenticaion (message, data origin, entity)
- Non-repudiation: preventing the sender from later denying that they sent the message

Definition 1. *Kerkhoff's Principle: The adversary knows all details about a crypto system except the secret key.*

Definition 2. *Cipher: A method or algorithm used to transform readable data (called plaintext) into an unreadable format (called ciphertext) to protect its confidentiality.*

Encryption is the process of converting plaintext into ciphertext. Decryption is the reverse process. Encryption uses the key k, decryption uses the key k'. If k = k', the system is symmetric. If k $\neq$ k', the system is asymmetric. Decryption(Encryption(m)) = m.

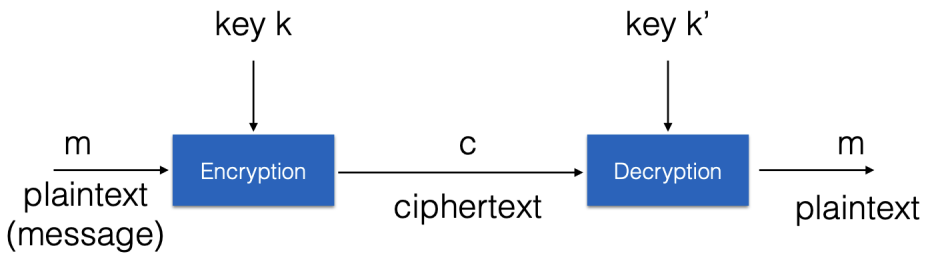


Figure 1: Encryption

Feature	Private Key Encryption	Public Key Encryption
Keys	Same key for encryption & decryption	Two keys: public and private
Speed	Faster	Slower
Key sharing	Must be kept secret	Only the private key is secret
Use cases	Encrypting large data, e.g., files	Secure key exchange, digital signatures

Table 1: Summary of Differences Between Private and Public Key Encryption

1.4 Topics Covered in Course

- Classical systems: simple ciphers, substitution, permutation, transposition, Caesar, Vigenere
- Information Theoretic Security
- Defining security: pseudorandomness, one-way functions, trapdoor functions
- Notions of security: perfect secrecy, semantic security, IND security
- Attacks on encryption schemes: objective, levels of computing power, amount of information available
- Types attacks: ciphertext-only, known plaintext, chosen plaintext, chosen ciphertext, adaptive
- Different types of adversaries: unbounded/polynomial computing power
- Security: unconditionally secure, computationally secure
- and more... see slides

2 Lecture 2

2.1 Number Theory

2.1.1 Modular Arithmetic

Definition 3. *A positive integer N is called the* modulus. *Two integers a and b are said to be congruent modulo N, written $a \equiv b \pmod{N}$, if N divides $b - a$.*

Examples:

$$18 \equiv 4 \pmod{7}, \quad -18 \equiv 3 \pmod{7}.$$

The set of integers modulo N is denoted by $\mathbb{Z}/N\mathbb{Z}$ or $\mathbb{Z}_N$:

$$\mathbb{Z}/N\mathbb{Z} = \{0, 1, \ldots, N-1\}, \quad \#(\mathbb{Z}/N\mathbb{Z}) = N.$$

Properties of Modular Arithmetic:

1. Addition is closed: $\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a + b \in \mathbb{Z}/N\mathbb{Z}$.
2. Addition is associative: $\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a + b) + c = a + (b + c)$.
3. 0 is an additive identity: $\forall a \in \mathbb{Z}/N\mathbb{Z} : a + 0 = 0 + a = a$.
4. The additive inverse always exists: $\forall a \in \mathbb{Z}/N\mathbb{Z} : a + (N - a) = (N - a) + a = 0$.
5. Addition is commutative: $\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a + b = b + a$.
6. Multiplication is closed: $\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a \cdot b \in \mathbb{Z}/N\mathbb{Z}$.
7. Multiplication is associative: $\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a \cdot b) \cdot c = a \cdot (b \cdot c)$.
8. 1 is a multiplicative identity: $\forall a \in \mathbb{Z}/N\mathbb{Z} : a \cdot 1 = 1 \cdot a = a$.
9. Multiplication and addition satisfy the distributive law: $\forall a, b, c \in \mathbb{Z}/N\mathbb{Z} : (a + b) \cdot c = a \cdot c + b \cdot c$.
10. Multiplication is commutative: $\forall a, b \in \mathbb{Z}/N\mathbb{Z} : a \cdot b = b \cdot a$.

2.1.2 Modular Exponentiation

Modular exponentiation is a technique used to efficiently compute expressions of the form $a^b \mod m$, especially for large b. The key idea is to repeatedly square the base a, reduce modulo m at each step, and combine results as needed.

Example: Compute $3^4 \mod 11$

1. Write the problem:
$$3^4 \mod 11$$
2. Break it into smaller steps using properties of modular arithmetic:
   (a) First, compute $3^2 \mod 11$:
   $$3^2 = 9 \quad \Rightarrow \quad 9 \mod 11 = 9$$
   (b) Then, square the result to get $3^4 \mod 11$:
   $$3^4 = (3^2)^2 = 9^2 = 81 \quad \Rightarrow \quad 81 \mod 11 = 4$$
3. Final result:
$$3^4 \mod 11 = 4$$

General Algorithm: Exponentiation by Squaring

1. If b is even:
$$a^b \mod m = \left(a^{b/2} \mod m\right)^2 \mod m$$

2. If b is odd:

$$a^b \mod m = \left(a \cdot a^{b-1} \mod m\right) \mod m$$

You can also simplify the problem by reducing the base modulo:

Definition 4. *For any a, b, n, if $a \equiv b \pmod{n}$, then $a^k \equiv b^k \pmod{n}$ for any positive integer k.*

See an example of this in practice session 1 exercise 1e.

2.1.3 Groups and Rings

Definition 5. *A* group *is a set with an operation that is:*

- *Closed,*
- *Has an identity element,*
- *Associative, and*
- *Each element has an inverse.*

Definition 6. *A group is* abelian *if it is also commutative.*

Examples:

- The integers under addition $(\mathbb{Z}, +)$, where the identity is 0 and the inverse of x is $-x$.
- The nonzero rationals under multiplication $(\mathbb{Q}^*, \cdot)$, where the identity is 1 and the inverse of x is $1/x$.

Group types:

- Multiplicative group: operation is multiplication.
- Additive group: operation is addition.
- Cyclic abelian group: generated by a single element.

Definition 7. *An abelian group G is called* cyclic *if there exists an element in the group, called the* generator*, from which every other element in G can be obtained either by repeated application of the group operation to the generator, or by the use of the inverse operation.*

- *If the group operation is multiplication ($(G, \cdot)$), a generator g produces all elements by repeated multiplication or division: $h = g^x$, where h is an arbitrary element in the group.*
- *In modular arithmetic, g is a generator if $g^x \mod m$ produces all nonzero elements of the group as x varies.*

Example: The group $\mathbb{Z}_7^*$ (the multiplicative group of integers modulo 7) consists of the nonzero integers modulo 7 under multiplication. The elements of the group are:

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}.$$

An element $g \in \mathbb{Z}_7^*$ is a generator if the powers $g^x \mod 7$ (for $x = 1, 2, 3, \ldots, 6$) produce **all elements** of $\mathbb{Z}_7^*$ exactly once. Let's test whether 3 is a generator:

1. Compute the powers of 3 modulo 7:

$$\begin{aligned}
3^1 \mod 7 &= 3, \\
3^2 \mod 7 = 9 \mod 7 &= 2, \\
3^3 \mod 7 = 27 \mod 7 &= 6, \\
3^4 \mod 7 = 81 \mod 7 &= 4, \\
3^5 \mod 7 = 243 \mod 7 &= 5, \\
3^6 \mod 7 = 729 \mod 7 &= 1.
\end{aligned}$$

2. The results are:

$$\{3, 2, 6, 4, 5, 1\}.$$

Since this list contains all elements of $\mathbb{Z}_7^*$, 3 is a generator of $\mathbb{Z}_7^*$. Other generators of $\mathbb{Z}_7^*$ include 5. You can verify this by computing $5^x \mod 7$ for $x = 1, 2, \ldots, 6$.

Definition 8. *A* ring *is a set with two operations $(+, \cdot)$ satisfying:*

- *The set is an abelian group under addition.*
- *Multiplication is associative and closed.*
- *Distributive laws hold.*

If multiplication is commutative, the ring is called *commutative*. Examples:

- Integers, real numbers, and complex numbers form infinite rings.
- $\mathbb{Z}/N\mathbb{Z}$ forms a finite ring.

2.1.4 Primes and Divisibility

Definition 9. *An integer a divides another integer b, denoted $a \mid b$, if $b = k \cdot a$ for some integer k.*

Definition 10. *A number p is* prime *if its only divisors are 1 and p.*

Examples of primes: $2, 3, 5, 7, 11, \ldots$.

Definition 11. *Greatest Common Divisor: $c = \gcd(a, b)$ if and only if c is the largest number that divides both a and b.*

Theorem 1. *Every positive integer can be written as a product of primes in a unique way.*

Definition 12. *Two integers a and b are coprime, relatively prime or mutually prime if the only positive integer that is a divisor of both of them is 1.*

Definition 13. *Euler's Totient Function: $\phi(p)$ is the number of integers less than p that are relatively prime to p.*

- *If N is a prime then $\phi(N) = N - 1$.*
- *If p and q are both prime and $p \neq q$, then $\phi(pq) = (p-1)(q-1)$*

$$\phi(N) = N\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\ldots\left(1 - \frac{1}{p_k}\right) = n\prod_{p|n}(1 - \frac{1}{p})$$

where $p_1, p_2, \ldots, p_k$ are the prime factors of N.

Euler's Totient function counts the number of positive up to a given integer N that are relatively prime to N.

2.1.5 Linear Congruences

Finding the solution to the linear congruence equation:

$$a \cdot x \equiv b \pmod{N}$$

We want to know how many solutions exist for x modulo N given the coefficients a, b, and the modulus N.

1. Compute the greatest common divisor (gcd) of a and N, denoted as $\gcd(a, N) = g$.
2. The following cases determine the number of solutions:
   (a) **If $g = 1$:**
   - When a and N are coprime ($\gcd(a, N) = 1$), the equation has **exactly one solution** modulo N. This is because a has a multiplicative inverse modulo N.

(b) **If $g > 1$ and $g \mid b$:**
- If $\gcd(a, N) = g > 1$ and g divides b, then there are **exactly g solutions** modulo N.
- These solutions can be determined by reducing the equation to a simpler congruence modulo N/g.

(c) **If $g > 1$ and $g \nmid b$:**
- If g does not divide b, then the equation has **no solution**. This is because b is not in the span of a modulo N.

Definition 14. *Multiplicative Inverse Modulo N: A number that, when multiplied by a given number a, gives a result of 1 modulo N. In other words, the multiplicative inverse of a modulo N is a number x such that:*

$$a \cdot x \equiv 1 \pmod{N}$$

- *The multiplicative inverse of a modulo N is denoted as a^{-1}.*
- *A multiplicative inverse of a modulo N exists only if a and N are coprime, i.e.,* $\gcd(a, N) = 1$.
- *If a and N are not coprime, it's impossible to find x such that $a \cdot x \equiv 1 \pmod{N}$.*
- *When N is a prime p, then for all non-zero values of $a \in \mathbb{Z}/p\mathbb{Z}$ we always obtain a unique solution to the equation $a \cdot x \equiv 1 \pmod{p}$.*

Inverse in this case means that the two numbers multiply to 1 modulo N. Think about regular numbers: the inverse of 2 is $\frac{1}{2}$ under multiplication, because $2 * \frac{1}{2} = 1$.

2.1.6 Fields

Definition 15. *A field is a set G with two operations $(G, \cdot, +)$. It satisfies the following properties:*

- *$(G, +)$ is an abelian group with identity element 0 (G is a commutative group under addition).*
- *$(G\backslash\{0\}, \cdot)$ is an abelian group ($G\backslash\{0\}$ is a commutative group under multiplicatio).*
- *Multiplication distributes over addition, i.e., $(G, \cdot, +)$ satisfies the distributive law.*

A field is like the "ideal playground" for numbers: You can add, subtract, multiply, and divide (except by 0). Both addition and multiplication behave nicely (associative, commutative, etc.). Examples of fields include familiar systems like real numbers and rational numbers. The key difference between rings and fields is that in a ring, division is not always possible. In a field, division (except by 0) is always possible, because every nonzero element has a multiplicative inverse.

$$\mathbb{Z}/N\mathbb{Z}$$

is a field if and only if N is prime (because then every nonzero element has a multiplicative inverse). Else, it is a ring.
Think of $\mathbb{Z}/N\mathbb{Z}$ as a "clock" with N hours. Once you pass $N - 1$, you wrap around back to 0. Arithmetic in $\mathbb{Z}/N\mathbb{Z}$ always "cycles" within the set $\{0, 1, \ldots, N - 1\}$.

$(\mathbb{Z}/N\mathbb{Z})^*$ is the set of all elements that are invertible (the set of elements that are coprime to N).

$$(\mathbb{Z}/N\mathbb{Z})^* = \{x \in \mathbb{Z}/N\mathbb{Z} : \gcd(x, N) = 1\}$$

The size of $(\mathbb{Z}/N\mathbb{Z})^*$ is given by Euler's Totient function: $\phi(N)$. If N is a prime p, then $(\mathbb{Z}/N\mathbb{Z})^* = \{1, ..., p - 1\}$.

2.1.7 Lagrange's Theorem

Lagrange's Theorem states that if $(G, \cdot)$ is a finite group with order (size) $n = \#G$, then for any element $a \in G$, the order of a (the smallest positive integer k such that $a^k = 1$) divides n. In particular, it follows that:

$$a^n = 1 \quad \text{for all } a \in G.$$

Application in Modular Arithmetic: In the context of modular arithmetic, consider the group of units $\mathbb{Z}/N\mathbb{Z}^*$ (the set of integers modulo N that are coprime to N, with multiplication as the group operation). If $x \in \mathbb{Z}/N\mathbb{Z}^*$, then the group has size $\phi(N)$, where $\phi(N)$ is Euler's totient function (the count of integers less than N that are coprime to N). Therefore:

$$x^{\phi(N)} \equiv 1 \pmod{N}.$$

2.1.8 Fermat's Little Theorem

Fermat's Little Theorem states that if p is a prime number and a is any integer, then:

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

If a is not divisible by p, then this can be rewritten as:

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

Explanation: This theorem tells us that raising a to the power of $p-1$ gives a remainder of 1 when divided by p, provided a and p are coprime. Fermat's Little Theorem is useful for simplifying modular exponentiation and serves as a foundation for more advanced results like Euler's theorem.

2.2 Basic Algorithms

2.2.1 Euclid's GCD Algorithm

The **Greatest Common Divisor** (GCD) of two integers a and b is the largest integer d such that d divides both a and b.

Key Idea: If we could factorize a and b, we could easily determine their GCD. For example, consider:

$$a = 2^4 \cdot 157 \cdot 4513^3, \quad b = 2^2 \cdot 157 \cdot 2269^3 \cdot 4513.$$

Here, the GCD is given by:

$$\gcd(a, b) = 2^2 \cdot 157 \cdot 4513 = 2{,}834{,}164.$$

However, computing prime factorizations is often impractical for large numbers. Instead, we use Euclid's Algorithm.

The Euclidean Algorithm is based on the principle:

$$\gcd(a, b) = \gcd(a \mod b, b).$$

The algorithm starts with two numbers a, b, where $a > b$. The remainder $r = a \mod b$ is computed. Then, a is repeatedly replaced with b (the smaller number), and b with r (the remainder). This process continues until the remainder is 0. The last non-zero remainder (b) is the GCD of a and b.

Steps:

1. Let $r_0 = a$ and $r_1 = b$.
2. Compute remainders $r_2, r_3, \ldots$ using:

$$r_{i+2} = r_i \mod r_{i+1}, \quad \text{where } r_{i+2} < r_{i+1}.$$

3. Stop when $r_{m+1} = 0$. The GCD is r_m.

Example: Compute $\gcd(21, 12)$:

$$\begin{aligned} \gcd(21, 12) &= \gcd(21 \mod 12, 12) = \gcd(9, 12), \\ \gcd(9, 12) &= \gcd(12 \mod 9, 9) = \gcd(3, 9), \\ \gcd(3, 9) &= \gcd(9 \mod 3, 3) = \gcd(0, 3). \end{aligned}$$

Thus, $\gcd(21, 12) = 3$.

2.2.2 The Extended Euclidean Algorithm

In addition to computing the GCD, the Extended Euclidean Algorithm finds integers x and y such that:

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

This is useful in many applications, such as finding modular inverses. For $\gcd(a, b) = d$ where $d = 1$, we can compute $ax + yN = 1$. Here **x** is the multiplicative inverse of a in modulo N. So, if $\gcd(a, N) = 1$, then $a^{-1} \mod N = x \mod N$.

Algorithm:

1. Start with $r_0 = a$, $r_1 = b$, $s_0 = 1$, $s_1 = 0$, $t_0 = 0$, $t_1 = 1$.
2. For each step, compute:
$$q_i = \left\lfloor \frac{r_{i-1}}{r_i} \right\rfloor, \quad r_{i+1} = r_{i-1} - q_i r_i,$$
$$s_{i+1} = s_{i-1} - q_i s_i, \quad t_{i+1} = t_{i-1} - q_i t_i.$$
3. Stop when $r_{i+1} = 0$. Then, $\gcd(a, b) = r_i$, and $x = s_i$, $y = t_i$.

Example: Compute $\gcd(36, 24)$ and coefficients x, y:

$$\text{Step 1: } q = \left\lfloor \frac{36}{24} \right\rfloor = 1, \quad r = 36 - 1 \cdot 24 = 12,$$
$$\text{Update: } x = 0 - 1 \cdot 1 = -1, \quad y = 1 - 1 \cdot 0 = 1,$$
$$\text{Step 2: } q = \left\lfloor \frac{24}{12} \right\rfloor = 2, \quad r = 24 - 2 \cdot 12 = 0,$$
$$\text{Update: } x = 1 - 2 \cdot (-1) = 3, \quad y = 0 - 2 \cdot 1 = -2.$$

Thus, $\gcd(36, 24) = 12$, with $x = -1$, $y = 1$.

2.2.3 The Chinese Remainder Theorem

Let $m_1, ..., m_r$ be pairwise relatively prime (i.e., $\gcd(m_i, m_j) = 1$ for all $i \neq j$). Let $x = a_i \mod m_i$ for all i. The CRT guarantees a unique solution given by:

$$x = \sum_{i=1}^{r} a_i M_i y_i \mod M$$

where

$$M_i = M / m_i$$

and

$$y_i = M_i^{-1} \mod m_i$$

y_i is the modular inverse of M_i modulo m_i (this can be computed using the Extended Euclidean Algorithm). The theorem is a way to solve a system of simultaneous modular congruences, finding a unique solution for a number that satisfies multiple modular equations, provided that the moduli are coprime/relatively prime.

$$x \equiv a_1 \mod m_1$$
$$x \equiv a_2 \mod m_2$$
$$...$$
$$x \equiv a_k \mod m_k$$

In that case, there exists a **unique solution** x modulo M, where M is the product of all the moduli:

$$M = m_1 \cdot m_2 \cdot \ldots \cdot m_k.$$

Example:

$$x \equiv 5 \mod 7$$
$$x \equiv 3 \mod 11$$
$$x \equiv 10 \mod 13$$

Then $M = 7 \cdot 11 \cdot 13 = 1001$. $M_1 = 1001/7 = 143$, $M_2 = 1001/11 = 91$, $M_3 = 1001/13 = 77$. $y_1 = 5, y_2 = 4, y_3 = 12$.

$$x = \sum_{i=1}^{r} a_i M_i y_i \mod M = 5 \cdot 143 \cdot 5 + 3 \cdot 91 \cdot 4 + 10 \cdot 77 \cdot 12 \mod 1001$$
$$= 3575 + 1092 + 9240 \mod 1001 = 13907 \mod 1001 = 894$$

add here why we need the CRT

2.2.4 Computing Legendre and Jacobi Symbols

Definition 16. *Let n be a positive integer. An integer a is called a* ***quadratic residue modulo*** *n if there exists an integer x such that:*

$$x^2 \equiv a \pmod{n}.$$

In other words, a is a quadratic residue modulo n if a is congruent to the square of some integer x modulo n. If no such x exists, then a is called a ***quadratic non-residue modulo*** *n.*

Example: For $n = 7$, the integers modulo 7 are $\{0, 1, 2, 3, 4, 5, 6\}$. Computing the squares of each integer modulo 7:

$$\begin{aligned}
0^2 &\equiv 0 \pmod{7},\\
1^2 &\equiv 1 \pmod{7},\\
2^2 &\equiv 4 \pmod{7},\\
3^2 &\equiv 2 \pmod{7},\\
4^2 &\equiv 2 \pmod{7},\\
5^2 &\equiv 4 \pmod{7},\\
6^2 &\equiv 1 \pmod{7}.
\end{aligned}$$

The quadratic residues modulo 7 are:

$$\{0, 1, 2, 4\}.$$

The quadratic non-residues modulo 7 are:

$$\{3, 5, 6\}.$$

Symmetry of squares: squaring numbers modulo n often produces repeated results due to symmetry in the group of residues. This means that different numbers can have the same square when considered modulo n. For each x, its symmetric counterpart $n - x$ produces the same square modulo n. The total number of unique quadratic residues is approximately half of n (or $\lfloor n/2 \rfloor + 1$ if 0 is included).

Definition 17. *Legendre Symbol: Let p be a prime number, and let a be an integer. The* ***Legendre symbol*** $\left(\frac{a}{p}\right)$ *is defined as follows:*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \pmod{p},\\ -1 & \text{if } a \text{ is a quadratic non-residue modulo } p,\\ 0 & \text{if } p \mid a \text{ (i.e., if } a \equiv 0 \pmod{p}). \end{cases}$$

- $\left(\frac{a}{p}\right) = 1$ means there exists an integer x such that $x^2 \equiv a \pmod{p}$ (i.e., a is a quadratic residue modulo p).
- $\left(\frac{a}{p}\right) = -1$ means that no such integer x exists (i.e., a is a quadratic non-residue modulo p).
- $\left(\frac{a}{p}\right) = 0$ means that a is divisible by p (i.e., $a \equiv 0 \pmod{p}$).

In simpler terms, the Legendre symbol answers the question: **"Can a be written as the square of some number, when working modulo p?"**
To detect squares modulo a prime p, we define:

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}. \tag{1}$$

Additional formulae:

$$\left(\frac{a}{p}\right) = \left(\frac{a \pmod{p}}{p}\right), \tag{2}$$

$$\left(\frac{a \cdot b}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right), \tag{3}$$

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}, \tag{4}$$

$$\left(\frac{a}{p}\right) = \begin{cases} -\left(\frac{p}{a}\right) & \text{if } p \equiv 3 \pmod 4, \\ \left(\frac{p}{a}\right) & \text{otherwise.} \end{cases} \tag{5}$$

Example: Compute the Legendre symbol $\left(\frac{15}{17}\right)$ to check if 15 is a quadratic residue modulo 17.

$$\begin{aligned} \left(\frac{15}{17}\right) &= \left(\frac{3}{17}\right) \cdot \left(\frac{5}{17}\right) && \text{by equation (3)} \\ &= \left(\frac{17}{3}\right) \cdot \left(\frac{17}{5}\right) && \text{by equation (5)} \\ &= \left(\frac{2}{3}\right) \cdot \left(\frac{2}{5}\right) && \text{by equation (2)} \\ &= (-1) \cdot (-1)^3 && \text{by equation (4)} \\ &= 1. \end{aligned}$$

So $\left(\frac{15}{17}\right) = 1$, and thus 15 is a quadratic residue modulo 17.

Instead of manually testing all possible values of x to see if $x^2 \equiv a \pmod p$, the Legendre symbol gives a quick, direct answer. This is especially helpful when working with large prime numbers.

- **Public-key cryptography** (like RSA) often relies on modular arithmetic and quadratic residues. The Legendre symbol helps in many cryptographic algorithms, such as those involving **Elliptic Curve Cryptography (ECC)**, **zero-knowledge proofs**, and **primality testing**.
- For example, in **the Diffie-Hellman key exchange**, one might need to check if certain numbers are quadratic residues in a modular group.

The Legendre symbol above is only defined when its denominator is a prime, but there is a generalization to composite denominators called the Jacobi symbol.

Definition 18. *For any integer a and odd integer n, the **Jacobi symbol** is defined as the product of the Legendre symbols corresponding to the* prime factors *of $n > 2$:*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k},$$

where

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$$

is the prime factorization of n.

It is defined as follows:

$$\left(\frac{a}{n}\right) = \begin{cases} 0 & \text{if } n \mid a, \\ 1 & \text{if } a \text{ is a quadratic residue modulo } n \text{ and } a \not\equiv 0 \pmod n, \\ -1 & \text{if } a \text{ is a quadratic non-residue modulo } n. \end{cases}$$

Remark 2. *If a is square, then the Jacobi symbol will be 1. However, if the Jacobi symbol is 1, a* might *not be a square.*

2.3 Primality Tests

Prime numbers are needed almost always in every public key algorithm. How can you find prime numbers?

Theorem 2. ***The Prime Number Theorem:*** *The number of primes less than X can be given estimated with:*

$$\pi(X) \approx \frac{X}{\log(X)}$$

There are many prime numbers! The probability of a random value to be a prime is $\frac{1}{\log(p)}$. If we need a prime number with 100% certainty, we need a proof of primality.

2.3.1 Fermat's Primality Test

Recall that

$$a^{\phi(N)} \equiv 1 \mod N$$

If N is a prime, this equality holds. However, if this equality holds, N is *not necessarily* prime. Probably prime: N is composite with a probability of $\frac{1}{2^k}$. k refers to the number of independent tests or iterations performed to check the primality of a number N. Each test involves choosing a random integer a and checking whether $a^{N-1} \equiv 1 \pmod N$.

Remark 3. *Carmichael numbers are composite numbers that pass the Fermat primality test for all possible values of a. They are rare but can be problematic in cryptographic applications. They always return* probably prime.

Algorithm 2.1: Fermat's test for primality

```
for i = 0 to k − 1 do
    Pick a ∈ [2, ..., n − 1].
    b ← a^{n−1} mod n.
    if b ≠ 1 then return (Composite, a).
return "Probably Prime".
```

Figure 2: Algorithm for Fermat Primality Test

2.3.2 Miller-Rabin Primality Test

The Miller-Rabin test is an improvement over the Fermat test. Unlike deterministic primality tests (which can definitively prove whether a number is prime), the Miller-Rabin test provides a result with high probability. If the test declares a number to be composite, then it is definitely not prime. However, if the test declares the number to be prime, there is still a small chance that it is actually composite (this is the "probabilistic" part). The Miller-Rabin test checks whether a number n passes certain conditions that hold for all prime numbers. It does this by examining the modular arithmetic properties of numbers related to n. If n passes these tests, it is likely prime. If it fails, n is definitely composite.

Algorithm 2.2: Miller–Rabin algorithm

```
Write n − 1 = 2^s · m, with m odd.
for j = 0 to k − 1 do
    Pick a ∈ [2, ..., n − 2].
    b ← a^m mod n.
    if b ≠ 1 and b ≠ (n − 1) then
        i ← 1.
        while i < s and b ≠ (n − 1) do
            b ← b^2 mod n.
            if b = 1 then return (Composite, a).
            i ← i + 1.
        if b ≠ (n − 1) then return (Composite, a).
return "Probable Prime".
```

Figure 3: Algorithm for Miller-Rabin Test

2.4 Elliptic Curves

Definition 19. *An elliptic curve is an equation of the form* $F : y^2 = x^3 + ax + b \mod p$, *with constants a, b.*

- $p > 3$, *otherwise* $x^3 = x$
- *If P is on F, then also* $P + P, P + P + P$, ... *are on F.*

Not all equations make good elliptic curves. They must satisfy a condition that ensures there are no sharp points or self-intersections. The curve must be *smooth* and *non-singular*. This means that the *discriminant must be nonzero*.

$$4a^3 + 27b^2 \neq 0$$

Point Addition:

- Addition of two points P and Q on the curve gives you another point R, which is also on the curve. To add $P = (x_1, y_1)$ and $Q = (x_2, y_2)$:

  1. Draw a straight line through P and Q. This line will generally intersect the curve at exactly one more point, say R'.
  2. Reflect R' across the x-axis to get $R = (x_3, y_3)$, the result of $P + Q$.

  The formulas to compute $R = (x_3, y_3)$ are:

  $$x_3 = \lambda^2 - x_1 - x_2$$

  $$y_3 = \lambda(x_1 - x_3) - y_1$$

  where λ (the slope of the line) is:

  $$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\delta y}{\delta x}$$

- If you're adding P to itself (doubling), the line you draw is the tangent to the curve at P. The formulas for $R = 2P = (x_3, y_3)$ are:

  $$x_3 = \lambda^2 - 2x_1$$

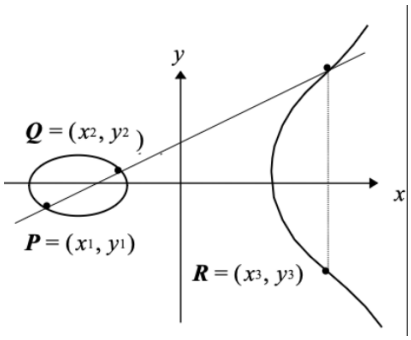
  $$y_3 = \lambda(x_1 - x_3) - y_1$$

  Here, λ (the slope of the tangent) is:

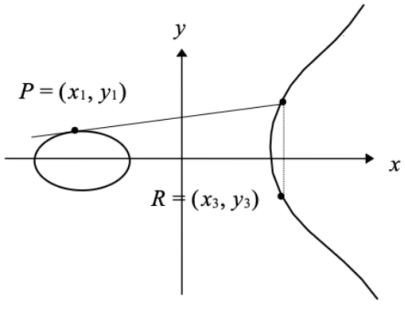
  $$\lambda = \frac{3x_1^2 + a}{2y_1}$$

- If P and Q are vertical opposites (e.g., $P = (x, y)$ and $Q = (x, -y)$), the line through them is vertical, and their sum is the **point at infinity**, $\mathcal{O}$. Think of $\mathcal{O}$ as the "zero" for point addition.

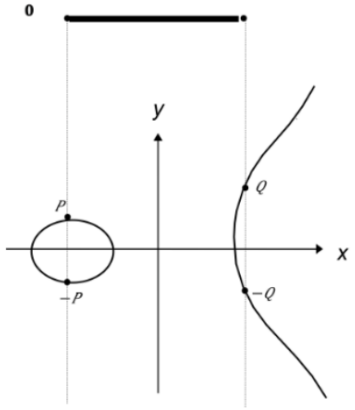
- Special properties:
  - **Commutative:** $P + Q = Q + P$
  - **Associative:** $(P + Q) + R = P + (Q + R)$
  - **Identity Element:** Adding the point at infinity $\mathcal{O}$ to any point P gives P (like adding zero).



(a) Point Addition



(b) Point Doubling



(c) Zero Point

Using elliptic curves lets us create very secure systems with shorter keys (really large prime numbers are not required), which means faster and more lightweight encryption.

Definition 20. ***Elliptic Curve Discrete Log Problem****: For a given integer m and a point P, it is easy to compute $Q = mP$. However, given P and Q, it is hard to compute m.*

Imagine a simple operation: start at one point on the curve, "add" it to itself repeatedly, and you get another point. If I tell you the starting point and the number of additions, it's easy to figure out the end point. But if I give you the end point and ask you to figure out the number of additions, that's really hard. This is what makes elliptic curve cryptography (ECC) secure. For cryptographic validity, the curve must:

- Have a sufficiently large number of points (called the order) to ensure security.
- Avoid known vulnerabilities (e.g., weak curves susceptible to specific attacks like small subgroup attacks).

3 Lecture 3

3.1 The Syntax of Encryption

Some terms:

- M: message space
- C: ciphertext space
- K: key space
- Three algorithms
  - Key Generation: **Gen** is a *probabilistic* algorithm that outputs a key $k \in K$ chosen according to some distribution.
  - Encryption: **Enc** takes as input a key $k \in K$ and a message $m \in M$ and outputs a ciphertext $c \in C$.
  - Decryption: **Dec** takes as input a key $k \in K$ and a ciphertext $c \in C$ and outputs a message $m \in M$.

3.2 Classical Ciphers

3.2.1 Caesar Cipher

The Caesar cipher is one of the simplest and oldest encryption techniques, named after Julius Caesar, who used it to secure his communications. It works by shifting the letters of the alphabet by a fixed number of positions. For example, with a shift of 3, the letter "A" becomes "D", "B" becomes "E", and so on. To decrypt a message, the recipient simply shifts the letters back by the same number. Although it is easy to understand and implement, the Caesar cipher is considered insecure by modern standards, as there are only a limited number of possible shifts (25), making it vulnerable to brute force attacks.

3.2.2 Shift Cipher

A keyed variant of Caesar's cipher: the secret key can take value between 0 and 26. Each letter in the plaintext is shifted by a certain number of positions in the alphabet. The key difference is that the shift can vary, meaning the amount of movement of each letter can differ across the message. In a typical shift cipher, a number (the key) determines how far to shift each letter. Decryption requires shifting in the opposite direction by the same key. Like the Caesar cipher, the shift cipher is vulnerable to brute force attacks due to the limited number of possible shifts.

$$c = m + k \mod 26$$

Cryptanalysis of the shift cipher can be done using letter and bigram frequencies. This cipher is based on language, and language has structure. Thus it can be decrypted. This approach leverages the fact that in natural languages, certain letters or combinations of letters appear more frequently than others. Example approach:

- **Analyze the ciphertext**: Count the frequency of each letter in the encrypted message.
- **Compare with language statistics**: Match the observed frequencies with the known frequency distribution of letters in the target language (e.g., "E" is often the most frequent in English).
- **Make educated guesses**: Substitute frequently occurring ciphertext letters with likely plaintext letters.
- **Refine the substitution**: Use context, common words, and patterns to adjust and decode the message.

3.2.3 Statistical Distance

Used to evaluate the security of cryptographic systems by measuring how close an observed probability distribution (e.g., the distribution of ciphertexts) is to a desired distribution (usually uniform). Attacks can exploit patterns or non-uniformity in data. The goal is that a secure encryption scheme produces ciphertext that is statistically indistinguishable from random noise.

$$\delta[X, Y] = \frac{1}{2} \sum_{u \in V} |\Pr_{X \leftarrow D_1}[X = u] - \Pr_{Y \leftarrow D_2}[Y = u]|$$

where X, Y are random variables with distributions $D_1, D2$ respectively.

The statistical distance between the ciphertext distribution (produced by the encryption scheme) and the uniform distribution is measured. A smaller statistical distance implies stronger security, as it becomes harder for an attacker to distinguish between encrypted data and random noise.

3.2.4 Substitution Cipher

Each letter (or symbol) in the plaintext is replaced with another letter or symbol according to a specific rule or mapping. The mapping defines the "key" for the cipher, and the process is reversible if the recipient knows the key. The key space consists of all bijections, or permutations ($26! \approx 2^{88}$ for English alphabet). But using letter frequencies, the cipher can be easily solved.

Example:

- Plaintext: `HELLO`
- Cipher Key:
$$\text{A} \to \text{Q},\ \text{B} \to \text{W},\ \text{C} \to \text{E},\ \ldots,\ \text{Z} \to \text{P}$$
- Key mapping:
  - Plain alphabet: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
  - Cipher alphabet: `QWERTYUIOPASDFGHJKLZXCVBNM`
- Ciphertext: `XUBBE`

Types of Substitution Ciphers:

- *Mono-alphabetic*: Uses a single mapping for the entire message (e.g., Caesar cipher).
- *Poly-alphabetic*: Uses multiple mappings that change throughout the message for added complexity.

3.2.5 Vigenère Cipher

Also called a poly-alphabetic substitution cipher: encrypt each letter with a dfferent alphabet. Use a keyword to shift letters in the plaintext. Unlike the Caesar cipher, where each letter is shifted by a constant number, the Vigenère cipher uses a series of shifts based on the letters of a keyword. This makes it more secure than simple substitution ciphers, as the pattern of shifts changes across the message. Still, cryptanalysis is easy.

Definition 21. ***Kasiski Test:*** *Find the occurence of the repeated sequences and use* gcd *to determine the key length.*

The plaintext is encrypted by shifting each letter by the number of positions specified by the corresponding letter in the keyword. If the keyword is shorter than the plaintext, it is repeated until it matches the length of the plaintext.

Example:

- Plaintext: `HELLO`
- Keyword: `KEY`
- Repeat the keyword to match the length of the plaintext: `KEYKE`
- Shift each letter in the plaintext by the corresponding letter in the keyword:
  - `H` (shift by `K`, which is 10) $\to$ `R`
  - `E` (shift by `E`, which is 4) $\to$ `I`
  - `L` (shift by `Y`, which is 24) $\to$ `J`
  - `L` (shift by `K`, which is 10) $\to$ `V`
  - `O` (shift by `E`, which is 4) $\to$ `S`
- Ciphertext: `RIJVS`

3.2.6 Permutation Cipher

The *positions* of the characters in the plaintext are rearranged according to a specific permutation (a predetermined key) rather than *replacing* characters with other symbols. The basic idea is that the order of the characters is shuffled, but no new symbols are introduced.

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 3 & 5 \end{pmatrix} = (1243) \in S_5.$$

Once upon a time there was a little girl called Snow White.

onceu ponat imeth erewa salit tlegi rlcal ledsn owwhi teahb.

coenu npaot eitmh eewra lsiat etgli crall dlsdn wohwi atheb.

Figure 5: Permutation Cipher

The key is a sequence of numbers that indicates the new positions of each character in the plaintext. Here, $(1243) \in S_5$, meaning; position 1 goes to position 2, position 2 goes to position 4, and so on. The key space is $|S_n| = n!$.

Key take aways: all historical, classical systems are broken. They rely on substituion and permutation. Designing secure ciphers is hard.

4 Lecture 4

4.1 Information Theory

Information theory is the mathematical study of the quantification, storage, and communication of information. By Claude Shannon. A key measure in information theory is *entropy*. Entropy quantifies the amount of uncertainty involved in the value of a random variable or the outcome of a random process. For example, identifying the outcome of a fair coin flip (which has two equally likely outcomes) provides less information (lower entropy, less uncertainty) than identifying the outcome from a roll of a die (which has six equally likely outcomes).

4.2 Security Definitions

Definition 22. *Computationally Secure: it takes N operations using the* ***best known algorithm*** *to break a cryptographic system and N is too large to be feasible.*

Definition 23. *Provably Secure: breaking the system is reduced to solving some well-studied hard problem.*

Definition 24. *Unconditional Secure/Perfectly Secure: the system is secure against an adversary with unlimited computational power.*

Key size is important. Advances in computer hardware and algorithms are important. In the future, it will be broken due to hardware or better algorithms.

4.3 Probability and Ciphers

Definition 25. *Let P denote the set of plaintexts, K the set of keys, and C denote the set of cipher texts. $p(P = m)$ is the probability that the plaintext is m. Then,*

$$p(C = c) = \sum_{k:c\in\mathbb{C}(k)} p(K = k) \cdot p(P = d_k(c))$$

This formula calculates the probability of a specific cipher text $C = c$ occuring. It sums over all keys k that could result in the ciphertext c , combining the probabilities of the key and the corresponding plaintext. It is fundamental in several cryptographic analyses:

- Ciphertext distribution: compute the distribution of the ciphertexts given the plaintext and key distributions. By ensuring $p(C = c)$ is uniform, cryptographers achieve perfect secrecy in schemes like the One-Time Pad.
- Evaluating security metrics: the uncertainty of C is maximized for a secure cipher, ensuring that an attacker cannot infer patterns.

4.4 Perfect Secrecy

Previously, the ciphertext revelas a lot of information about the plaintext. We want a system in which ciphertext does not reveal anything about the plaintext.

Definition 26. *Perfect secrecy: a cryptosystem has perfect secrecy if*

$$p(P = m|C = c) = p(P = m)$$

for all plain texts m and ciphertexts c.

Lemma 1. *Assume the cryptosystem is perfectly secure, then*

$$\#K \geq \#C \geq \#P$$

where # denotes the number of items in the corresponding set.

4.4.1 One-Time Pad

Theorem 3. *Shannon's Theorem: Let $(P, C, K, e_k(), d_k())$ denote a cryptosystem with $\#K = \#C = \#P$. Then the cryptosystem provides* perfect secrecy *if and only if:*

- *Every key is used with equal probability $1/\#K$*
- *For each $m \in P$ and $c \in C$, there is a unique key k such that $c = e_k(m)$*

Definition 27. *One-Time Pad a cryptographic system that provides perfect secrecy when implemented correctly. It is a symmetric encryption technique (shift cipher) where the sender and receiver use a shared secret key, known as the pad, that is as long as the plaintext message.*

$$\#K = \#P = \#C = 26^n$$

and $p(K = k) = 1/26^n$. Also known as the Vernam cipher. It is perfectly secure because:

- *The key is truly random*
- *The key is as along as the plaintext*
- *The ciphertext reveals no information about the plaintext. The ciphertext C is independent of the plaintext P without the key K, and observing the ciphertext does not reduce the uncertainty about the plaintext.*
- *The key is used only once (hence the name)*

Example: Suppose the plaintext P is "A", and the key K is a random letter. After encryption, the ciphertext C could be any letter, with equal probability. If an attacker intercepts $C = Z$, they cannot determine whether the plaintext P was "A", "B", or any other letter without the key. Every possible plaintext is equally likely.

4.5 Entropy

Due to the key distribution problem (the key must be as long as the message), perfect secrecy is not practical. Instead, we need a cryptosystem in which **one key can be used many times**, and **a small key can encrypt a long message**. Such a system is not perfectly secure, but it should be computationally secure. We need to measure the amount of information first: Shannon's entropy.

Definition 28. *Shannon's Entropy: Let X be a random variable which takes a finite set of values x_i, with $1 \leq 1 \leq n$, and has a probability distribution $p(x)$. We use the convention that if $p_i = 0$ then $p_i \log_2(p_i) = 0$. The entropy of X is defined as:*

$$H(X) = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$$

Properties:

- $H(X) \geq 0$
- $H(X) = 0$ *if* $p_i = 1$ *and* $p_j = 0$ *for* $i \neq j$
- *if* $p_i = 1/n$ *for all* i*, then* $H(X) = \log_2(n)$

Example: For a specific question X: "Will you go out with me?", the answer is Yes or No. If you always say No, the amount of information, $H(X) = 0$. If you always say Yes, $H(X) = 0$. You know the result. If you say Yes and No with equal probability, $H(X) = 1$. When you get the answer, no matter what it is, you learn a lot. Here, $H()$ is the entropy, and is independent of the length of X.

4.6 Joint Entropy, Conditional Entropy and Mutual Information

Used to measure the uncertainty and interdependence between random variables.

Definition 29. *The **joint entropy** of two random variables X and Y measures the total uncertainty in the pair (X, Y), considering them together. The total amount of information contained in* one *observation of both X and Y.*

$$H(X,Y) = -\sum_{x \in X} \sum_{y \in Y} P(x,y) \log_2 P(x,y)$$

where $P(x,y)$ is the joint probability *of $X = x$ and $Y = y$.*

$$H(X,Y) \leq H(X) + (HY)$$

Definition 30. *The **conditional entropy** of X given Y, $H(X|Y)$, measures uncertainty in X after knowing Y. It quantifies how much information about X is still unknown once Y is known.*
The entropy of X given an observation of $Y = y$:

$$H(X \mid Y = y) = -\sum_{x} p(X = x \mid Y = y) \cdot \log_2 p(X = x \mid Y = y).$$

Then,

$$H(M \mid C) = \sum_{c} p(C = c) \cdot H(M \mid C = c) = -\sum_{m} \sum_{c} p(C = c) \cdot p(M = m \mid C = c) \cdot \log_2 p(M = m \mid C = c).$$

Definition 31. *Conditional and joint entropy are connected as follows:*

$$H(X,Y) = H(Y) + H(X|Y)$$

$$H(X|Y) \leq H(X)$$

Definition 32. ***Mutual information** quantifies the amount of shared information between X and Y. It measures how much knowing X reduces the uncertainty about Y (or vice versa). The expected amount of information that Y gives about X (or X about Y).*

$$I(X;Y) = H(X) + H(Y) - H(X,Y) = H(Y) - H(Y \mid X) = H(X) - H(X \mid Y)$$

Mutual information represents the reduction in uncertainty about one variable due to knowledge of the other. If X and Y are independent, $I(X;Y)$ because knowing one provides no information about the other.

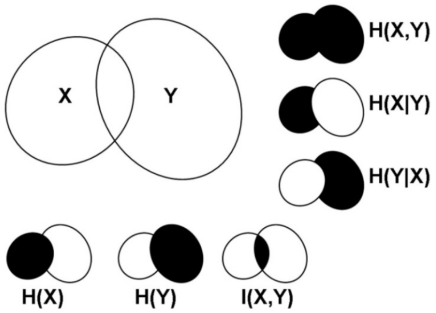
Example Suppose X and Y represent the outcomes of rolling two six-sided dice:

- $H(X)$: Entropy of die X (e.g., how random die X's outcomes are).
- $H(X,Y)$: Joint entropy, the uncertainty of the combined outcome of X and Y.
- $H(Y \mid X)$: Given a roll of X, the remaining uncertainty in Y.
- $I(X;Y)$: Shared information (likely 0 if the dice rolls are independent).

4.6.1 Application to Ciphers

- $H(P|K,C) = 0$: If you have the cipher text and the key, you can decrypt and obtain the plaintext. There is no surprise, hence 0.
- $H(C|P,K) = 0$: When you have the key and the plaintext to can encrypt and obtain the ciphertext. True for deterministic cryptosystems. Not for modern ones that use randomness (the same plaintext can be encrypted with the same key but correspond to a different ciphertext).
- $H(K,C) = H(K) + H(P)$
- Then, $H(K|C) = H(K,C) - H(C) = H(K) + H(P) - H(C)$. Given the ciphertext, can we obtain information of the key?

Graphical Representation



4.7 Spurious Keys and Unicity Distance

Definition 33. *Spurious keys: the remaining possible but incorrect keys.*

Tactic of an attacker is to reduce the number of spurious keys to zero. Then the one correct key remains.
Plaintext are not random: natural languages... Random text has an entropy of $H_L = 4.7$ bits (5 bits per letter). *continue...*

4.7.1 Entropy of a Natural Language

Definition 34. *The entropy of a natural language measures the average* amount of information *conveyed per character, word, or sentence in that language, accounting for redundancy and patterns in its structure. It is defined by*

$$H_L = \lim_{n\to\infty} \frac{H(P^n)}{n}$$

where n is the length of letters, and P^n is the n-grams.

For English (estimation): $1.0 \leq H_L \leq 1.5$ bits per character. This means you need 5 bits of data to represent only 1.5 bits of information. Natural languages are highly redundant (e.g., letter and word patterns), making them less random than completely independent symbols. The low entropy of natural languages makes cryptographic attacks easier because it introduces predictability and redundancy (useless information) in the plaintext, which attackers can exploit. To mitigate this, cryptographic systems should:

- Use strong encryption methods that destroy plaintext patterns (e.g., AES or OTP).
- Introduce randomness (e.g., padding).
- Avoid predictable plaintext in sensitive communications.

4.7.2 Redundancy

Definition 35. *Redundancy is information that is expressed more than once. The repetition or predictability of information, where certain elements (like letters, words, or phrases) appear more frequently or follow patterns. It is defined as:*

$$R_L = 1 - \frac{H_L}{\log_2 \#\mathbb{P}}$$

where $\#\mathbb{P}$ is the message space.

Example: For English,

$$R_L \approx 1 - \frac{1.25}{\log_2 26} = 0.75$$

This means 75 percent redundancy, 75 % of the information in a message is predictable or can be removed without losing meaning. You can zip your files 3 quarters.

Definition 36. *The **average number of spurious keys** is:*

$$\bar{s}_n \geq \frac{\#\mathbb{K}}{\#\mathbb{P}^{n \cdot R_L}} - 1$$

An attacker wants this number to be zero.

We can make this number zero making the number of keys equal to the number of n-grams times the redundancy. This leads to unicity distance.

4.7.3 Unicity Distance and Ciphers

Definition 37. ***Unicity distance** is the average number of cipher texts for which the expected number of spurious keys becomes 0.*

$$n_0 \approx \frac{\log_2 \#\mathbb{K}}{R_L \cdot \log_2 \#\mathbb{P}}$$

In the context of a substition cipher, the unicity distance is the length of the ciphertext required for an attacker to break the cipher with high probability (i.e., to uniquely determine the key). It depends on the number of possible keys and the redundancy of the language used. Given $\#P = 26, \#K = 26!, R_L = 0.75$,

$$n_0 \approx \frac{88.4}{0.75 \cdot 4.7} \approx 25$$

To successfully break the cipher (i.e., determine the correct key), an attacker needs to analyze at least 25 characters of ciphertext. This is because, with this many characters, the redundancy and patterns in the language (due to its predictable letter frequencies) will provide enough information for the attacker to determine the key. This highlights the cipher's vulnerability: there are 26! possible keys! The smaller the unicity distance, the easier it is to break the cipher with limited ciphertext.

For a modern stream cipher $\#P = 2$ (binary symbol 0 or 1), $\#K = 2^l, R_L = 0.75$, then

$$n_0 \approx \frac{l}{0.75} = \frac{4 \cdot l}{3}$$

Here, the unicity distance tells us how many ciphertext bits are needed to break the streamcipher. The distance is proportional to the key length l. Longer key lengths lead to larger unicity distances, making it harder to break the cipher.

With no redundancy, $R_L = 0$. Then:

$$n_0 \approx \frac{l}{0} = \infty$$

There are no predictable patterns in the plaintext. Perfect randomness would imply that every symbol in the plaintext carries the maximum amount of information, with no repeated or predictable patterns.In such a case, the language would have maximum entropy, and there would be no chance of making guesses about the plaintext from patterns in the ciphertext. In practice: no natural language has zero redundancy.

5 Lecture 5

5.0.1 What does it mean to be secure?

Modern cryptography is focused on three key aspects:

1. **Definitions:** Concrete mathematical definition of what it means for a particular cryptographic mechanism to be secure.
2. **Schemes:** Design the schemes which will (hopefully) meet the security definitions.
3. **Proofs:** Check whether the design meets the security definitions (provable/reductionist security).

For instance with factorization, given a large number, it is very difficult to factorize. An algorithm to do this efficiently in polynomial time would break all schemes that rely on factorization.

5.1 Security Games

Security games are mathematical frameworks used in cryptography to formally evaluate the security of a computational problem or cryptographic system. They involve an interaction between two entities:

- **The challenger**, who provides an input (e.g., a computational problem).
- **The adversary** (A), who attempts to solve the problem or break the cryptographic system under specific rules and constraints.

The goal is to quantify the probability of the adversary's success within a defined resource limit (like time or computational power).

5.1.1 The FACTOR Problem

In this example, the security game focuses on the **Factorization Problem**, where the adversary tries to factorize a large number N, generated as the product of two large primes p and q.

Steps in the Security Game:

1. Setup by the Challenger:
   - Two large primes p and q are randomly chosen, each with a bit size of $v/2$.
   - The product $N = p \cdot q$ is computed and given to the adversary A.
2. Goal of the Adversary:
   - The adversary A "wins" the game if they can find two integers p' and q' such that:
   $$p' \cdot q' = N,$$
   $$p', q' \neq p, q.$$
3. Time Constraint:
   - The adversary has a limited amount of computational time t to solve the problem.

5.1.2 Measuring the Adversary's Advantage

The advantage of the adversary A in this game is defined as:

$$\mathrm{Adv}_v^X(A, t) = \Pr\left[A \text{ wins the game } X \text{ for } v = \log_2 N \text{ in time less than } t\right].$$

Here:

- v is the bit-length of N, representing the problem's complexity.
- t is the allowed time for A.

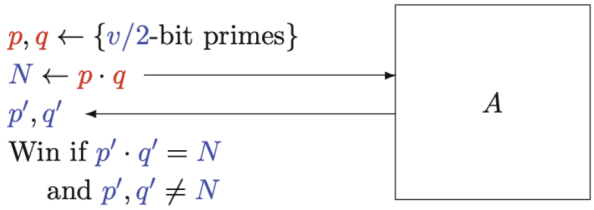


Figure 6: Security game to define the FACTOR problem

- Pr is the probability that A successfully factors N under these conditions.

$$\mathrm{Adv}_v^X(A) = 2 \cdot \left| \Pr\left[A \text{ wins } X \text{ for } v = \log_2 N\right] - \frac{1}{2} \right|$$

A always wins: 1. A always guesses: 0.

5.2 Pseudo-Random Functions

Definition 38. *A pseudorandom function $F_k(x)$ is a deterministic function that takes an input x and produces an output that is computationally indistinguishable from a truly random function, given only oracle access to the function. No efficient adversary can distinguish between $F_k(x)$ and a truly random function, without knowing the secret key k.*

Example in Security Game: The game below does not follow Kerkhoff's principle. The adversary always wins, because everything is known. There is no key/security parameter. The adversary can easily check if $F(x) = y$.

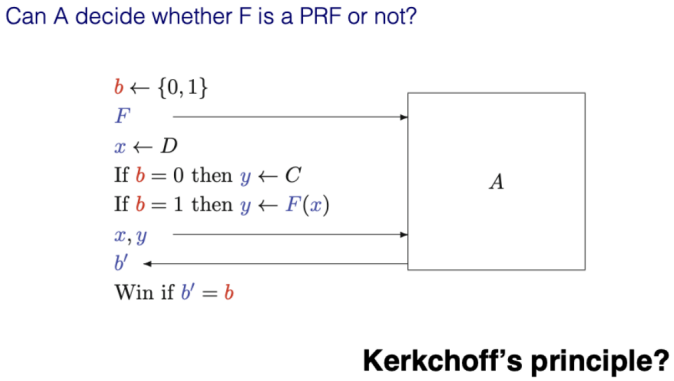


Figure 7: Security game to define the PRF problem. Here, the adversary always wins!

Changing the SG to include a key parameter k: we are not having one function F, but several functions chosen based on a secret key: $F_k(x)$. The adversary does not know the key. The adversary now has access to an oracle:

Definition 39. *An **oracle** is an abstract, theoretical concept representing a "black box" that provides answers to specific queries according to a defined rule or function. The term is often used to model adversaries' access to resources or information in a controlled manner during security analyses.*

- *The oracle operates according to predefined rules or a specific function (e.g., encryption, decryption, signature generation, or random sampling).*
- *The adversary can query the oracle to obtain information or perform operations, but cannot directly access the oracle's internal workings.*
- *For example, an encryption oracle takes a plaintext as input and outputs the corresponding ciphertext, while a decryption oracle does the reverse.*

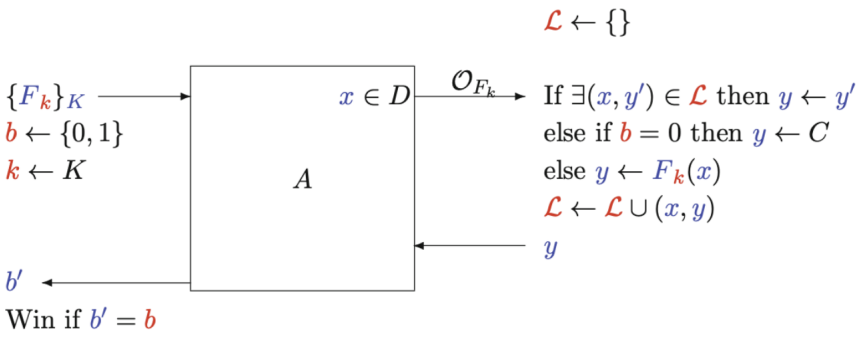


Figure 8: The final security game for a PRF

Example: Oracle in a Security Game For a PRF function security game, the oracle $\mathcal{O}_k(x)$ behaves as follows:

- With secret key k, the oracle implements $\mathcal{O}_{F_k}$, where F_k is a pseudorandom function.
- Alternatively, the oracle can implement $\mathcal{O}_R$, where R is a truly random function.
- The oracle should not give two different answers for the same input x. Hence, it has a memory $\mathscr{L}$.

The adversary queries the oracle with different x values and observes the outputs. The adversary's goal is to distinguish whether the oracle is implementing a PRF or truly random function, without knowing the secret key k. In figure 8, this is indicated by the bit b. The adversary wins if they can correctly guess b.

5.2.1 Adversary's Advantage

The adversary's *advantage* measures how well it can distinguish the oracle's behavior from random guessing. This is formally defined as:

$$\mathrm{Adv}^{\mathrm{PRF}}_{\{F_k\}_K}(A) = 2 \cdot \left| \Pr\left[A^{O_{F_k}} \text{ wins}\right] - \frac{1}{2} \right|$$

Key Components:

- $\Pr\left[A^{O_{F_k}} \text{ wins}\right]$: The probability that A correctly guesses the bit b.
- $\frac{1}{2}$: The probability of guessing b purely at random.

Interpretation:

- If A cannot distinguish $F_k(x)$ from a random function, its success probability is $\frac{1}{2}$, meaning $\mathrm{Adv}^{\mathrm{PRF}}_{\{F_k\}_K}(A) = 0$.
- If A has some advantage in distinguishing $F_k(x)$ from random, the value Adv increases, indicating the weakness of the PRF.

5.2.2 Why is this important?

This security game is critical for evaluating whether a function $F_k(x)$ behaves indistinguishably from a truly random function. In cryptographic systems:

- Strong PRFs ensure adversaries can't predict outputs or distinguish the function, which is vital for the security of encryption, key derivation, and other cryptographic primitives.
- A PRF that is distinguishable from random may leak information or fail to provide the desired security guarantees.

5.3 Trapdoor Functions

Definition 40. *A **one-way function** is a function that is easy to compute in one direction but* hard to invert*.*

- ***Easy to compute:*** *Given an input x, it is computationally efficient to calculate $f(x)$.*
- ***Hard to invert:*** *Given $f(x)$, it is computationally infeasible to find x (or any x' such that $f(x') = f(x)$) without additional information.*

Definition 41. *A **trapdoor** function is a one-way function that is easy to compute in one direction but hard to invert unless a secret "trapdoor" is known. An extra piece of information helps to invert the function, e.g. RSA.*

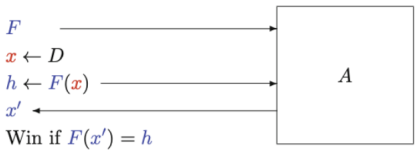


Figure 9: Security game for a one-way function

5.4 Public Key Cryptography

- **Key distribution problem** in symmetric (private key) systems.
- A **key pair** is needed:
  - **Private key** for decryption (only the owner can decrypt).
  - **Public key** for encryption (everyone can encrypt).
- Public and private keys are linked in a mathematical way:
  - Obtaining the private key from the public key is **NOT easy**.
  - Obtaining the public key from the private key is **easy**.

The security of an encryption scheme (both symmetric and asymmetric) is dependent on:

- The goal of the adversary
- The types of attacks allowed
- The computational model

5.5 Basic Notions of Security

Notation and valid encryption:

$$\forall k \in \mathbb{K}, \forall m \in \mathbb{P}, d_k(e_k(m)) = m$$

What does it mean for a symmetric encryption scheme to be secure? Adversary should not learn the underlying (plaintext) message.

5.5.1 OW-PASS attack

One-wayness under passive attack.

- Adversary's Capability: The adversary only observes the encryption of a plaintext message but cannot interact with the encryption mechanism in any way.
- Goal: The adversary is given the ciphertext $c = \text{Enc}(m)$ and attempts to recover the plaintext m.
- Assumptions:
  - The adversary has no control over the plaintexts being encrypted.
  - Security is evaluated against passive observers who can only eavesdrop.
- Use Case: Suitable for scenarios where the attacker does not have active access to the encryption process.

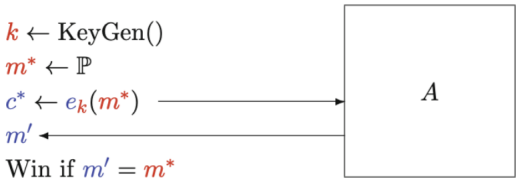


Figure 10: Security game for symmetric key OW-PASS

5.5.2 OW-CPA attack

OW-PASS is very limiting for the adversary. She should have an encryption oracle. One-Wayness under Chosen Plaintext Attack:

- Adversary's Capability: The adversary can interact with the encryption mechanism by choosing plaintexts and obtaining their corresponding ciphertexts.
- Goal: The adversary, after observing ciphertexts for chosen plaintexts, attempts to recover the plaintext of a given ciphertext.
- Assumptions:
  - The adversary can actively query the encryption oracle with plaintexts of their choice.
  - This security definition is stronger than OW-PA because it assumes a more powerful adversary.
- Use Case: Applies to scenarios where the attacker has partial or controlled access to the encryption process, such as in chosen plaintext attacks on public-key cryptography.

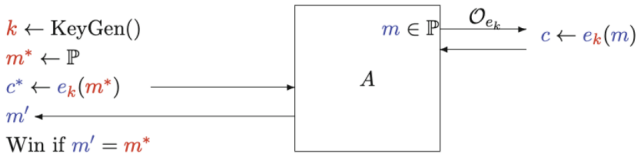


Figure 11: Security game for symmetric key OW-CPA

5.5.3 OW-CCA attack

The adversary should be able to decrypt as too (limited number). Thus, One-wayness under Chosen Ciphertext Attack:

- Adversary's Capability: The adversary is allowed to interact with both the encryption and decryption oracles. Specifically, they can:
  - Query the decryption oracle for the decryption of any ciphertext c, except for the target ciphertext c^*.
  - Query the encryption oracle to obtain the ciphertext for any plaintext m of their choice.
- Goal: The adversary attempts to recover the plaintext m^* from a given target ciphertext $c^* = \text{Enc}(m^*)$.
- Assumptions:
  - The adversary has powerful capabilities to both query encryption for chosen plaintexts and query decryption for chosen ciphertexts.
  - The decryption oracle cannot be used directly on the target ciphertext c^* to prevent trivial success.
- Use Case:

- OW-CCA security is crucial for applications requiring strong guarantees against active attackers who can manipulate ciphertexts.
- It is relevant in scenarios such as securing communications where adversaries can intercept, modify, and replay ciphertexts.

- Key Features:
  - Adversary's Advantage: OW-CCA tests the robustness of a cryptographic scheme against the most powerful adversaries who can both observe and manipulate encrypted communications.
  - Stronger Security: OW-CCA provides stronger guarantees compared to OW-PA and OW-CPA as it accounts for adversaries who have decryption capabilities.

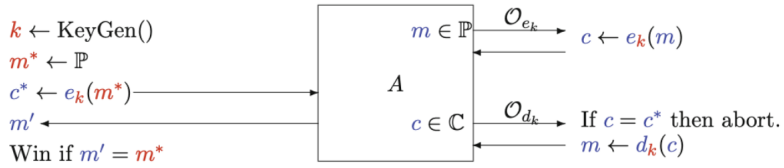


Figure 12: Security game for symmetric key OW-CCA

5.6 Modern Notions of Security

Before, the attacks assumed that you obtain the whole key. In real life, the adversary can also partially break the system (one bit of the key). The adversary should not be allowed to obtain **any** information about the plaintext!

Perfect secrecy is not practical, because every plaintext needs to have the same probability. To do so, the key needs to be as long as the message.

Definition 42. *Semantic security: like perfect security but the adversary has polynomially bounded computing power. The adversary cannot extract partial knowledge.*

$$g : M \to \{0,1\}$$

$$Pr[g(m) = 1] = Pr[g(m) = 0] = \frac{1}{2}$$

$$Adv_{\Pi}^{SEM}(S) = 2 \cdot \left| Pr[S(c) = g(d_k(c))] - \frac{1}{2} \right|$$

Semantic security ensures that an adversary cannot learn any meaningful information about a plaintext from its ciphertext, even if they have some prior knowledge about the plaintext. Formally, a cryptosystem is semantically secure if, given a ciphertext, the adversary cannot distinguish between the encryptions of two chosen plaintexts.

Semantic security is *difficult to show*! Polynomial security (IND) is easier to show. If a system is IND secure, it is also semantically secure.

Definition 43. *IND Security: an encryption scheme is* IND-secure *if the ciphertexts of two plaintexts are indistinguishable to an adversary, ensuring that the adversary gains no advantage from observing ciphertexts, even with access to oracles (depending on the attack model).*

IND Security ensures that an adversary cannot distinguish between the ciphertexts of two chosen plaintexts, even if they select the plaintexts themselves. This is a fundamental property of secure encryption. IND Security requires the use of *probabilistic* encryption for public-key encryption schemes to ensure that ciphertexts for the same plaintext look different every time they are encrypted. This randomness prevents adversaries from exploiting patterns in ciphertexts to distinguish between plaintexts.

5.6.1 IND Security Games

The IND framework is defined under different adversarial capabilities:

IND-CPA (Indistinguishability under Chosen Plaintext Attack):

- The adversary can choose two plaintexts m_0 and m_1.
- A random bit b is chosen, and the ciphertext of m_b is given to the adversary.
- The adversary must guess b.
- Success probability $> \frac{1}{2}$ implies a weakness in the scheme.

IND-CCA Indistinguishability under Chosen Ciphertext Attack):

- Similar to IND-CPA, but the adversary is also allowed to query a decryption oracle.
- The adversary cannot query the decryption oracle for the target ciphertext.
- This provides a stronger security guarantee than IND-CPA.

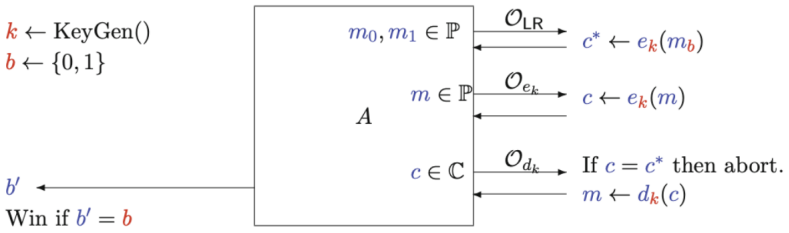


Figure 13: Security game for symmetric key IND-CCA

If the cryptosystem is deterministic, you can fool the oracles. In figure 13, $\mathcal{O}_{LR}$ will only encrypt one of m_1, m_2. You use the other message in the second step $\mathcal{O}_{e_k}$, and check if you receive the same ciphertext as in step 1. The decryption oracle $\mathcal{O}_{d_k}$ can be fooled if there is a relationship between the plain text and the cypertext. This is possible in homomorphic encryption. For secure communication, homomorphism is not appreciated. But we have it because in public cryptosystems we rely on mathematically difficult problems.

Definition 44. *An encryption algorithm is secure if it is semantically secure against a CPA attack.*

Definition 45. *An encryption algorithm is secure if it is IND-CCA secure.*

Theorem 4. *A system which is IND-PASS secure must be semanticallt secure against passive adversaries.*

$$\prod \textit{ is IND-CCA} \Rightarrow \prod \textit{ is IND-CPA} \Rightarrow \prod \textit{ is IND-PASS}$$
$$\prod \textit{ is IND-XXX} \Rightarrow \prod \textit{ is OW-XXX}$$

5.7 Other Notions of Security

- Many Time Security: How many times can we use the LR oracle?
- Real-or-Random: Oracle encrypts either the real message or a random message of the same size.
- Lunchtime Attacks: (CCA1) Adversary has decryption oracle during the find stage for some time.
- Nonce-Based Encryption: Deterministic algorithms are not IND-CPA secure. Therefore, nonce (number used once) should be used.
- Data Encapsulation Mechanism: Symmetric system but key is used once.
- Non-malleability: Given a ciphertext of an unknown plaintext, the adversary can compute a new ciphertext for a *related* plaintext.
- Plaintext-aware: One needs to know the plaintext to encrypt.

5.7.1 Malleability

Malleability in cryptography refers to a property of some encryption schemes where an adversary, given a ciphertext C, can modify it to produce another ciphertext C' such that C' decrypts to a related plaintext M', where M' is a function of the original plaintext M.
In a *malleable encryption scheme*, the attacker does not need to know M to create C'. This is considered a vulnerability because the adversary can manipulate encrypted data without breaking the encryption itself. Why is malleability a problem?

1. **Breaking data integrity:** Encryption schemes are often expected to protect both confidentiality and integrity of the plaintext. Malleability breaks this assumption, as it allows attackers to modify data without detection.
2. **Practical attacks:** Malleability can be exploited in protocols that rely on encrypted data. For instance, in electronic payments, malleable encryption could allow an attacker to modify the transaction amount in a secure message.

Example: Suppose we are using a simple additive encryption scheme:

$$C = M + k$$

where k is the encryption key, M is the plaintext, and C is the ciphertext.
An attacker, without knowing k, can modify C by adding some value d:

$$C' = C + d$$

When C' is decrypted:

$$M' = C' - k = (M + k + d) - k = M + d$$

Thus, the attacker has changed the plaintext M to $M' = M + d$ without needing to know the key k.

Mitigating Malleability:

1. **Authenticated encryption:** Combine encryption with mechanisms to verify data integrity, such as using a MAC (Message Authentication Code) or digital signatures.
2. **Non-malleable encryption schemes:** Use encryption schemes that inherently prevent modification of ciphertexts to produce related plaintexts. For example, many modern schemes like AES-GCM or RSA-OAEP are designed to be non-malleable.

5.8 Random Oracle Model

Finish later

6 Lecture 6

Randomness is extremely important in cryptography, and the same random values should never be used twice. It is hard to achieve, often we use pseudo-random number generators (PRNGs) to generate randomness.

Example of Importance of Randomness: In a perfect secrecy scheme, the key is as long as the message. If the same key is reused even once, the messages can be decrypted. XORing their ciphertexts effectively removes the key, leaving a direct relationship between the two plaintexts.

For a message m and key k, the ciphertext is $c = m \oplus k$, where $\oplus$ is the XOR-operation. For two messages encrypted with the same key:

$$c_1 = m_1 \oplus k$$
$$c_2 = m_2 \oplus k$$

Now, if we XOR the ciphertexts:

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

Using the property of XOR that $a \oplus a = 0$ and $a \oplus b \oplus a = b$, we can deduce:

$$c_1 \oplus c_2 = m_1 \oplus m_2 \oplus k \oplus k$$

$$c_1 \oplus c_2 = m_1 \oplus m_2$$

Thus, the XOR of the two ciphertexts directly reveals the XOR of the two plaintexts. The "distance" between two messages m_1 and m_2 represented by their XOR operation $m_1 \oplus m_2$, reveals the bitwise differences between them. This is often inpreted as the Hamming distance (the number of positions at which two strings of the same length differ) between the two messages. The XOR distance not only represents differences but can be exploited to reconstruct messages in insecure systems where the same key is reused (for instance, when the adversary knows one of the plaintexts already and can reconstruct the other).

6.1 Stream Ciphers

Definition 46. *Stream ciphers are based on zeroes and ones. They encrypt bits (bit by bit) instead of blocks of plaintext. Due to bit operations, they are very fast. Easy to implement on hardware and software.*

Consider the following stream cipher:

$$c = m \oplus F_k(0)$$

Where $F_k(0)$ is a function that generates a keystream based on the key k. The keystream is XORed with the plaintext to produce the ciphertext. The keystream is generated by a pseudo-random number generator (PRNG) that takes the key as input.

If the key was the same length of the message it would be perfect, but creating such a long key would be very inefficient (only done for government to government communication). However, we can use a shorter key to mimic this behaviour. The short key is used to choose the output of a PRF (the key is the seed of the PRNG). It will mimic pseudo-randomness long enough (until it repeats itself). The cipher will be passive secure if the PRF is random enough.

6.2 Linear Feedback Shift Registers (LFSRs)

We can use LFSRs for generating a binary stream.

Definition 47. *A **Linear Feedback Shift Register** (LFSR) is a sequential shift register that generates a sequence of bits based on a linear feedback function. It consists of:*

- *A sequence of single-bit registers (0 or 1)*
- *Feedback function: a linear function (usually XOR) that determines how bits are fed back into the register.*

- *Tap positions in the register used in the feedback function.*

How it works:

1. Initialization: The register is initialized with a seed value (a non-zero starting state).
2. Bit Shifting: The register shifts all bits to the right (or left), discarding the last bit.
3. Feedback Calculation: A new bit is computed based on the XOR of the bits in the "tapped" positions.
4. Repeat: The new bit is inserted into the first position of the register, and the process repeats to generate the next bit in the sequence.

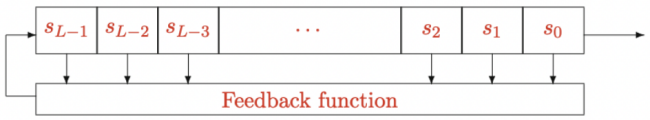


Figure 14: Feedback shift register

The initial state is very important, as it determines the entire sequence (whole sequence of zeroes always results in zeroes). The feedback function and length of the registers are also important.

6.2.1 Properties of LFSRs

- Periodicity: LFSRs are periodic; they generate a sequence that eventually repeats. The maximum period is $2^n - 1$, where n is the register size, achieved when the feedback taps correspond to a primitive polynomial.
- Deterministic: If the initial state (seed) and feedback function are known, the sequence is fully predictable.
- Efficiency: LFSRs are computationally efficient, using only shift and XOR operations.

6.2.2 Feedback Functions

Feedback functions in shift registers are used to compute the new bit that is fed back into the register. The key difference between linear and nonlinear feedback functions lies in how they combine the bits from the register.

Feature	**Linear Feedback**	**Nonlinear Feedback**
Operations	XOR and other linear operations	Nonlinear operations (AND, OR, S-boxes, etc.)
Predictability	Easier to predict with known state	Harder to predict, more secure
Efficiency	Computationally efficient	More computationally intensive
Periodicity	Periodic, with a maximum of $2^n - 1$	Can be non-periodic or have a very long period
Applications	Pseudo-random generators, CRC	Secure encryption, cryptographic systems

Table 2: Comparison of Linear and Nonlinear Feedback Functions

We would like to have a non-linear feedback function, but these are difficult to design. Difficult to estimate whether it is a good non-linear function. When you hit 0 the whole value chain becomes 0, and the cycle is broken.

6.2.3 Zero State in Feedback Functions:

If an LFSR (or any shift register using feedback functions) reaches the value **zero**, the system will typically "lock" into this state and remain there indefinitely. This is because:

1. Feedback Function Output: In a linear feedback system, the XOR of all zeros is zero, meaning that once all bits are zero, the feedback function will continue producing zeros.
2. State Transition: The state of the LFSR will not change, and the register will stay stuck at zero, generating an output sequence that consists entirely of zeros.

Implications:

- **In LFSRs:**
  - The sequence degenerates and loses all pseudo-randomness, becoming a constant stream of zeros.
  - This reduces the period of the LFSR to just 1, which is undesirable, especially in cryptographic or pseudo-random number generation applications.
- **In Cryptographic Systems:**
  - If the feedback function hits zero during an encryption or random number generation process, it can render the output insecure.
  - Attackers can easily exploit the fact that the system produces predictable, non-random output (all zeros).

Avoiding the Zero State:

1. Seed Initialization: The initial state (seed) of the register is carefully chosen to ensure it is non-zero.
2. Primitive Polynomials: For LFSRs, the taps (feedback positions) are chosen based on primitive polynomials. This ensures the register cycles through all possible non-zero states ($2^n - 1$) before repeating.
3. Nonlinear Feedback Functions: Nonlinear systems often include mechanisms to avoid degenerative states like zero, introducing additional complexity to prevent such behavior.
4. External Checks: Some systems include checks to ensure that a zero state is never reached, restarting the LFSR with a valid non-zero state if necessary.

6.2.4 Mathematical Expression

Cell is tapped or not (content of the cell contributes to the feedback function yes or no):

$$[c_1, c_2, \ldots, c_L]$$

Initial state of the registers:

$$[s_{L-1}, \ldots, s_1, s_0]$$

Output:

$$[s_0, s_1, s_2, s_3, \ldots, s_{L-1}, s_L, s_{L+1}, \ldots]$$

Then, for $j \geq L$:

$$s_j = c_1 \cdot s_{j-1} \oplus c_2 \cdot s_{j-2} \oplus \cdots \oplus c_L \cdot s_{j-L}.$$

However, we need to see a repetition after a certain period, N, such that:

$$s_{N+i} = s_i.$$

N can be maximum $2^L - 1$.

The **connection polynomial** $C(X)$ describes how the bits stored in the LFSR are combined to produce the feedback bit, which determines the next state of the LFSR. It is defined as:

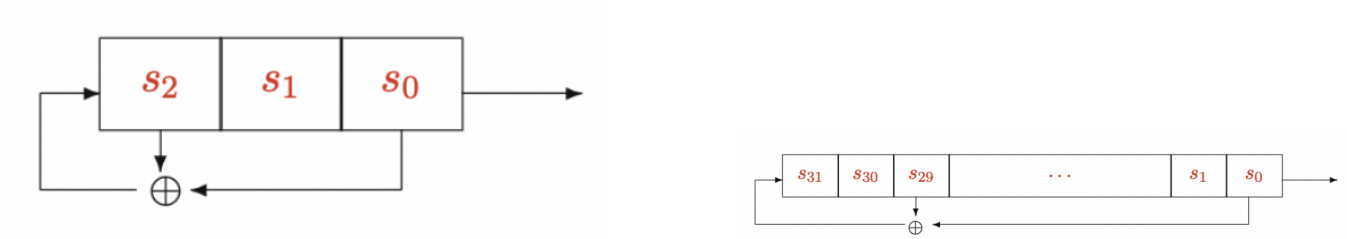
$$C(X) = 1 + c_1 \cdot X + c_2 \cdot X^2 + \cdots + c_L \cdot X^L \in \mathbb{F}_2[X]$$

where $\mathbb{F}_2[X]$ is the set of polynomials over the binary field (coefficients are either 0 or 1). Proper choice of $C(X)$ can result in a maximal period of $2^L - 1$.

The **characteristic polynomial** $G(X)$ describes the LFSR's output sequence and the polynomial that can be used to generate the entire sequence of output bits. It is a function of the feedback logic defined by the connection polynomial.

$$G(X) = X^L \cdot C(1/X)$$

Example:



(a) $X^3 + X + 1$ (b) $X^{32} + X^3 + X + 1$

Figure 15: Linear Feedback Shift Register Examples

a) This is a small LFSR with length $L = 3$, described by the connection polynomial $C(X) = X^3 + X + 1$. There are 3 registers: s_0, s_1, s_2. The feedback function involves: X^3 (feedback from the last register), X (feedback from the first register), and a constant 1. At each step, the contents of the tapped cells (s_2, s_0) are XOR-ed to produce a new bit.

b) This is a much larger LFSR with length $L = 32$, described by the connection polynomial $C(X) = X^{32} + X^3 + X + 1$. There are 32 registers. The feedback function involves the 32nd (last) register, the 3rd register, the 1st register, and a constant 1. The characteristic polynomial $G(X)$ is used to generate the entire output sequence. *I dont get the exp values for this one??*

Note: The connection polynomial determines which registers contribute to the feedback. But is not a direct description of the output stream itself. Instead, it describes the feedback mechanism of the LFSR.

How do we know that the feedback function is a good one, such that we achieve the maximum length before the stream cipher repeats itself?

6.2.5 Primitive Polynomial

Definition 48. *A* primitive polynomial *is a polynomial that generates a maximal-length sequence in a finite field, particularly in the context of binary fields $\mathbb{F}_2$ (i.e., polynomials whose coefficients are either 0 or 1). Specifically, for a polynomial to be primitive, it must satisfy the following conditions:*

- *The polynomial must be* ***irreducible*** *over $\mathbb{F}_2$ (i.e., it cannot be factored into the product of lower-degree polynomials with coefficients in $\mathbb{F}_2$).*
- *The polynomial must generate a sequence that cycles through all the nonzero elements of the finite field $\mathbb{F}_{2^L}$, where L is the degree of the polynomial.*

In simpler terms, a primitive polynomial produces a sequence of numbers that goes through all possible states except the zero state (when viewed as an LFSR) before repeating.

When used in LFSRs, primitive polynomials ensure that the generated sequence has a *maximum length* before it repeats. The length of the sequence is $2^L - 1$, where L is the degree of the polynomial.

Theorem 5. *If N is the period, then the characteristic polynomial $f(x)$ is a factor of $1 - X^N$.*

$$C(X) = 1 + c_1 \cdot X + c_2 \cdot X^2 + \cdots + c_L \cdot X^L \in \mathbb{F}_2[X]$$

Conditions for $C(X)$:

- The highest coefficient $c_L = 0$: the polynomial is singular. This means that $C(X)$ is not a full degree of L and cannot properly generate sequences of maximal length in an LFSR.
- The highest coefficient $c_L = 1$: the polynomial is non-singular. $C(X)$ is a full degree of L and there is a periodicity in the sequences generated by the polynomial. The behaviour of the sequence depends on whether $C(X)$ is irreducible or not.

$C(X)$ is irreducible if it cannot be factored into smaller polynomials over $\mathbb{F}_2[X]$ (other than itself and 1). Then, the period of the sequence generated by the LFSR is the *smallest* N such that $C(X)$ divides $1 + X^N$. N will satisfy $N|(2^L - 1)$.

Example: 3 registers, maximum length is 7:

$$m = 3, p = 2^3 - 1 = 7$$

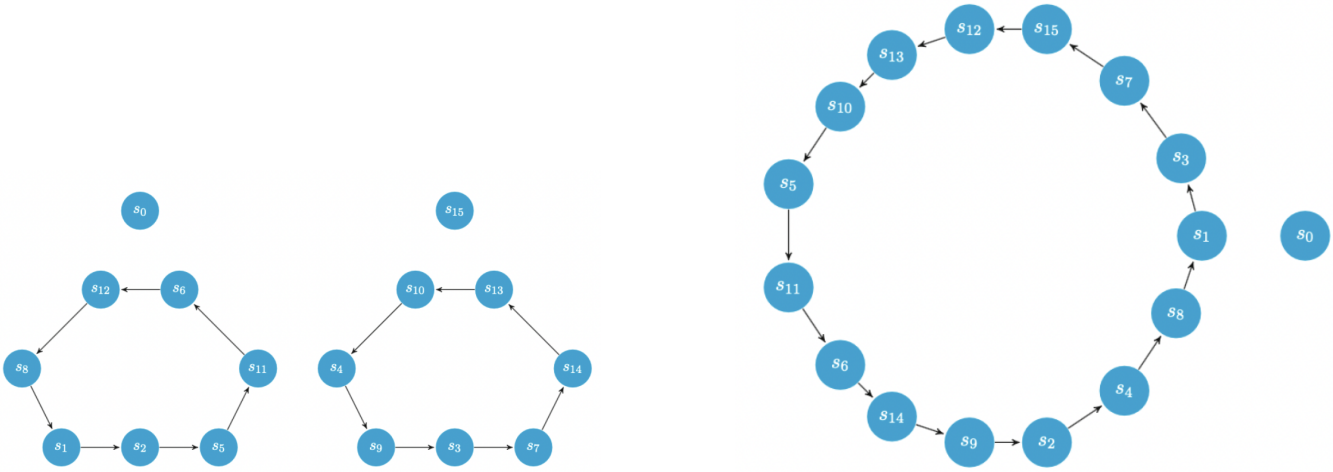
So we need to find the factors of:

$$1 - x^y = (1 - x)(1 + x + x^3)(1 + x^2 + x^3)$$

The divisors of this polynomial (irreducible polynomials) will have a period of 7.

6.3 Period

The presence of disjoint cycles in an LFSR, such as in figure 16, has implications for randomness:



(a) $X^4 + X^3 + X^2 + 1$ (b) $X^4 + X + 1$

Figure 16: State transitions of 4-bit LFSRs with connection polynomials

- The output sequence splits into two disjoint cycles of length 7 instead of a single cycle of length $2^L - 1$ (15 for $L = 4$).
- Reduced Period: The sequence repeats earlier, reducing the overall period and uniformity.
- Loss of Randomness: Missing states lead to gaps, making the sequence predictable and non-random.
- Cause: The connection polynomial is *not primitive*, preventing the LFSR from achieving a maximum-length sequence.

The right polynomial results in a big cycle with all states included, except for the zero state, as expected.

6.4 Security of LFSRs

With L registers and $2L$ known output bits, the connection polynomial can be revealed. It is a linear system. If you have sufficient equations, you can solve the linear system.

- First L bits reveal the s values
- We need to learn L unknowns: c values

$$s_j = \sum_{i=1}^{L} c_i \cdot s_{j-i} \pmod 2$$

Therefore, LFSRs are not secure!
Linear complexity notes here

6.5 Combining LFSRs

LFSR-based stream ciphers are very fast ciphers, suitable for implementation in hardware, to encrypt real-time data such as voice or video. But they need to be augmented with a method to produce a form of non-linear output. Create fast and secure non-linear behaviour by combining multiple LFSRs into a non-linear combination function:

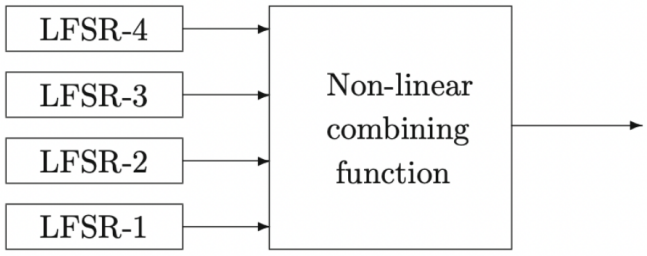


Figure 17: Combining LFSRs

6.5.1 Non-linear Combiners

Examples covered in lecture are:

- Alternating-step generator
- Shrinking generator
- A5/1
- Trivium
- RC4

How much do we need to know about these?

7 Lecture 7

Two families: symmetric cryptosystems use the same key for encryption and decryption. They rely on the principles from classical systems: substitution, permutation, and transposition. The reversing operation is dependent on the key. Asymmetric cryptosystems, on the other hand, use a pair of mathematically related keys—one for encryption (public key) and another for decryption (private key)—relying on complex mathematical problems for security.

7.1 Block Ciphers

Definition 49. *A **block cipher** is a deterministic cryptographic algorithm that operates on fixed-size blocks of data, typically n bits, using a symmetric key to perform encryption or decryption. It transforms a plaintext block of size n bits into a ciphertext block of the same size through a series of reversible, structured operations, and vice versa.*

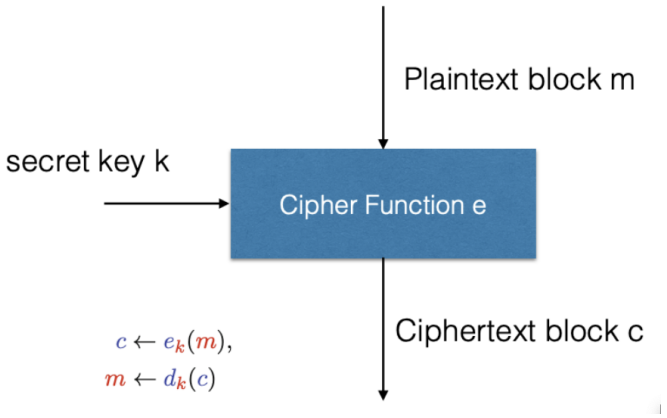


Figure 18: Block Cipher

where

- $m \in \{0,1\}^n$ is the plaintext block,
- $k \in K$ is the secret key, chosen from key space K,
- e is the encryption function,
- d is the decryption function,
- $c \in \{0,1\}^b$ is the ciphertext block.

NOTE: Block ciphers are not proper encryption schemes. You need to combine them with modes of operation! Otherwise your system will not be secure.

7.1.1 Properties of Block Ciphers

1. Input data block: Plaintext is divided into equal-sized blocks.
2. Encryption/Decryption: Each block undergoes multiple transformations (substitution, permutation, and mixing) to produce ciphertext.
3. Key: A symmetric key is applied in multiple *rounds* to ensure security.
4. Padding: If the last block of plaintext is smaller than the block size, padding is added.
5. Modes of operation: Block ciphers work with data larger than one block by using modes like:
   - ECB (Electronic Codebook)
   - CBC (Cipher Block Chaining)
   - GCM (Galois/Counter Mode)

Block ciphers are typically *64 bits* (in DES), *128 bits* (in AES) or more (in modern ciphers). They should act like a *pseudorandom permutation* (PRP).

Definition 50. *A **pseudorandom permutation** (PRP) is a function that defines a bijective (one-to-one and onto) mapping between input and output spaces, such that it is computationally indistinguishable from a truly random permutation when the key is unknown.*

In order to limit the advantage of the adversary, the key space is kept very large (e.g. *Adv* $1/|K|$). A block cipher is a *building block* for designing a cipher (a PRP). A block cipher with a *Mode of Operation* is a cipher. The goal is to design an IND-CCA secure cipher.

7.1.2 Design

DES and AES are iterated block ciphers. They *repeat a simple round function.* The round r can be fixed or variable. The more rounds, the higher the security of the cipher.

- In each round, a round key, derived from the key k, is used (by key scheduling algorithm) to process a block
- The round function should be *invertible*; for decryption the round keys are used in reverse order
- In DES: the round is invertible but not the round function
- In AES: both the round and the round function are invertible

Definition 51. *The **confusion-diffusion** paradigm is a fundamental design principle for secure cryptographic systems, particularly block ciphers.*

- ***Confusion:** Confusion ensures that the relationship between the key and the ciphertext is highly complex, making it difficult for an attacker to infer the key, even if they have access to multiple plaintext-ciphertext pairs. Split the block into smaller blocks and apply a substitution on each block.*
- ***Diffusion:** Diffusion ensures that the influence of a single bit of plaintext (or key) spreads widely over the ciphertext, so that changes in input affect many output bits. Mix permutations so that local change can effect the whole block.*

Definition 52. *A **substitution-permutation network** (SPN) is a design model for block ciphers. It consists of a series of linked operations, including substitution, permutation, and key mixing. It is a direct implementation of defintion 7.1.2.*

See figure 19 for a diagram of an SPN. *watch video on this for extra notes*

7.1.3 The Avalanche Effect

A small change in the input must affect every bit of the output. This is called the *avalanche effect.* It is a desirable property of block ciphers.

1. The S-boxes (substitution box) are designed such that 1 bit effects at least 2 bits in the output of the boxes
2. The mixing permutations are designed

In principle, you need at least 7 rounds for a good diffusion. This is mathematically shown (outside of context for this course). Too few rounds (e.g. 1 or 2) will not provide security. The rounds can easily be reverted.

7.2 Feistel Ciphers

Definition 53. *A **Feistel cipher** is a symmetric structure used to construct block ciphers, including DES. It splits the plaintext block into two halves and performs a series of rounds of processing on the halves. Each round applies a function F to one half and then combines it with the other half using a reversible operation, typically XOR.*

In a Feistel cipher, the round function is *invertible.* It is different from an SPN, as that one is based on a non-invertible component. We can choose any function F and still the round is invertible. The same hardware/software can be used for encryption and decryption (just use the keys in reverse order). The security of the cipher depends on: the round keys, the number of rounds, and the function F.

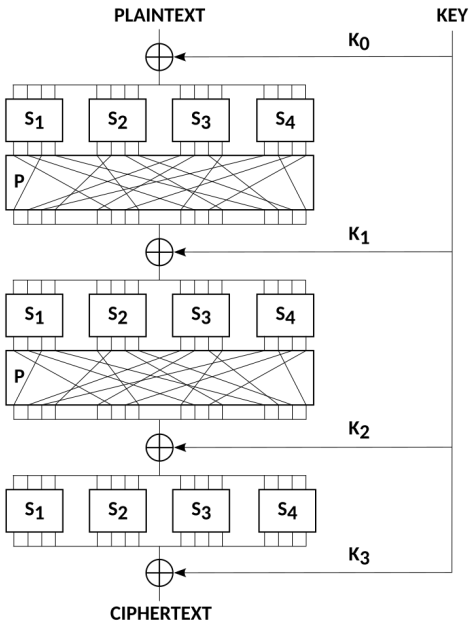


Figure 19: A sketch of a substitution–permutation network with 3 rounds, encrypting a plaintext block of 16 bits into a ciphertext block of 16 bits. The S-boxes are the Si, the P-boxes are the same P, and the round keys are the K_i.

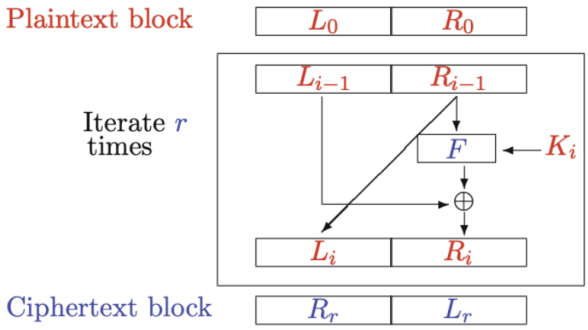


Figure 20: Basic operation of a Feistel cipher

7.2.1 Encryption

Given a plaintext block P, a Feistel cipher processes it as follows:

1. Initial split: Divide P into two equal-sized halves:

   $$P = (L_0, R_0)$$

   where L_0 and R_0 are the left and right halves, respectively.

2. Rounds of encryption: For each round i (1 to n), compute:

   $$L_i = R_{i-1}$$

   $$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

   where:

   - F is the round function (a non-linear function).
   - K_i is the round key for round i, derived from the main key.

3. Final Swap (Optional): After n rounds, the ciphertext C is obtained as:

   $$C = (R_n, L_n)$$

   (Note: The final swap is optional and does not affect decryption.)

7.2.2 Decryption

The Feistel structure makes decryption straightforward because it is inherently reversible. Using the ciphertext $C = (R_n, L_n)$:

1. Initialize R_n and L_n as inputs.

2. For each round i (in reverse order, from n to 1):

$$R_{i-1} = L_i$$

$$L_{i-1} = R_i \oplus F(L_i, K_i)$$

3. Combine L_0 and R_0 to recover the original plaintext.

The same function F and subkeys K_i are used in both encryption and decryption, but the subkeys are applied in reverse order during decryption.

7.3 Data Encryption Standard (DES)

DES is one of the earliest widely-used block ciphers. Not used anymore. But, still in ATMs (3DES). Key features are:

- Block size: 64 bits
- Key length: 56 bits (with 8 parity bits)
- Number of rounds: 16
- 16 round keys, 48 bits each
- Structure: Feistel cipher

In the Feistel structure, each round splits the input into two halves: One half is directly passed to the next round. The other half is transformed using a combination of substitution, permutation, and the round key, then XORed with the other half.

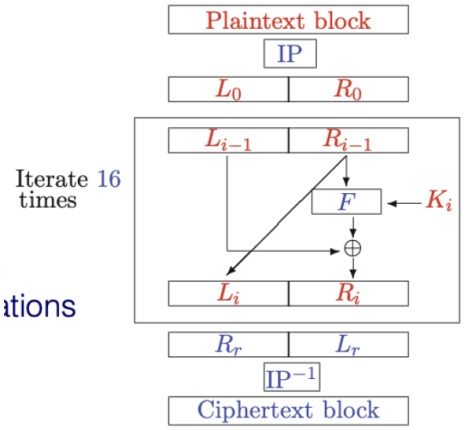


Figure 21: DES as a Feistel structure

Operations:

- 64 bits of plaintext block
- Perform an initial permutation (IP)
- Split the block into a left and right part
- Perform 16 rounds of *identical operations*
- Join the half blocks together
- Perform a final reverse permutation (IP^{-1})

Decryption is identical to encryption: rounds keys in reversed order! One algorithm is used for encryption and decryption. The same hardware can also be used for both. DES is based on a Feistel structure. All that is needed to specify DES is a round function, a key schedule, and any additional processing (initial and final permutation).

7.3.1 Function F

- Expansion permutation: the right half 32 bits is expanded to 48 bits
- Round key addition: XOR with the round key
- Splitting: 48 bits are split into 8 lots of 6-bit values
- S-boxes: non-linear part, output is 32 bits (8 lots of 4-bit values)
- P-box: Combine the previous lots to form a 32-bit value and apply permutation to form the output of F

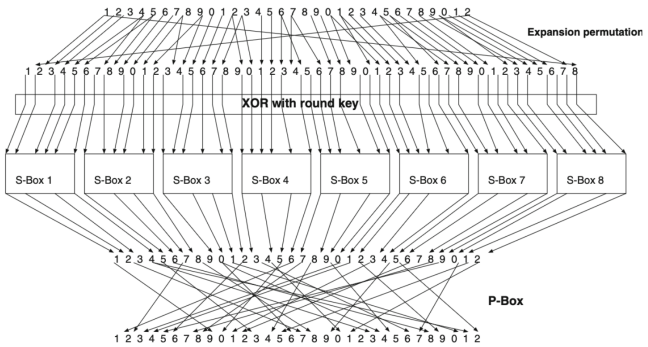


Figure 22: Structure of DES function F

7.3.2 S-boxes

In the **S-box** in the function F, the 48-bit result is divided into 8 groups of 6 bits. Each group of 6 bits is passed through a corresponding S-box, reducing it to 4 bits.

The S-box takes a 6-bit input and produces a 4-bit output. The 6-bit input is divided as follows:

- The **first and last bits** (bits 1 and 6) are used to determine the **row number**.
- The **middle 4 bits** (bits 2, 3, 4, and 5) are used to determine the **column number**.

The row and column values are used to look up the corresponding 4-bit value in the S-box table (see lecture slides). DES uses 8 S-boxes, denoted $S_1, S_2, \ldots, S_8$. Each S-box takes a 6-bit input and produces a 4-bit output. Together, the 8 S-boxes process the 48-bit input and reduce it to a 32-bit output. The S-boxes are important for the following reasons:

- Non-linearity: the S-box is the only non-linear component of the F-function, which makes DES resistant to linear and differential cryptanalysis.
- Confusion: S-boxes obscure the relationship between the plaintext, ciphertext, and the key.
- Compression: S-boxes reduce the 48-bit input to 32 bits, balancing the size for further processing.

7.3.3 Permutations and P-box

The **initial an final permutatations** are straight P-boxes that are inverses of each other. These permutations do not add cryptographic strength but are part of the standard DES algorithm.

- The IP rearranges the 64-bit plaintext input bits into a new order. It is applied only once at the very beginning of the DES encryption process
- The IP^{-1} reverses the effect of the IP. It rearranges the bits of the 64-bit block back to their original positions
- IP and IP^{-1} tables to specify how the input bits rearranged by index (see lecture slides for the tables)

The **expansion permutation** in the F-function of DES is a crucial step within each round of the encryption process. It is used to expand the 32-bit half-block input into 48 bits.

- Bit duplication
- Introduce non-linearity: duplicating and rearranging bits makes it harder for an attacker to analyze relationships between plaintext, ciphertext, and the key.
- Kex mixing: XOR with the 48-bit round key
- Table in lecture slides

The **P-box** rearranges the bits of a given input in a specified order. The purpose of this permutation is to spread the influence of each bit across the output to enhance security by increasing diffusion. The P-box is used after the S-box substitution stage within the Feistel structure of DES. The S-box reduces the input to 32 bits, and the P-box permutes these 32 bits to prepare them for the next round of processing.

7.3.4 Key Scheduling

The **Key scheduling** process in DES generates 16 **subkeys** (48 bits each) from the original **64-bit key**. These subkeys are used in the 16 Feistel rounds of DES encryption. The key scheduling process consists of the following main steps:

1. Initial 64-bit key is reduced to 56 bits by removing parity bits using the **PC-1** table.
2. The 56-bit key is split into two 28-bit halves: C_0 and D_0.
3. For each of the 16 rounds, both halves undergo **left circular shifts** (1 or 2 bits depending on the round).

$$C_i \leftarrow C_{i-1} \lll p_i$$

$$D_i \leftarrow D_{i-1} \lll p_i$$

   where p_i is the cyclic shift.
4. Combine the two halves. The **PC-2** table compresses the 56-bit key (combined C_i and D_i) into a 48-bit subkey.
5. This process is repeated 16 times to generate 16 unique subkeys.

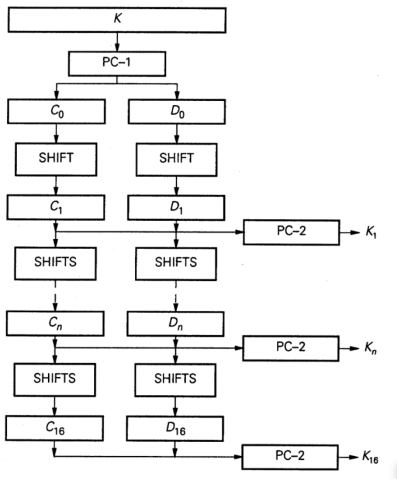


Figure 23: DES key scheduling

The key schedule ensures that even a small change in the original key produces completely different subkeys.

7.3.5 Security of DES

1, 2, and 3-round DES is not secure. DES in general is not secure! Can be cracked with exhaustive search. But: costs a lot of time or memory. Time-space tradeoff: with more memory, a few minutes is sufficient. The 56-bit key length is vulnerable to brute-force attacks (exhaustive search). Modern computational power can crack DES in hours. However, to crack DES you need mathematical, puzzle-solving skills and *luck*.

The key length is the weakest point in DES. Then can't we just encrypt twice, using two keys? The key space would become $2^{n \cdot k} = 2^{112}$ with $n = 2, k = 56$. Unfortunately, this is not true. The key space remains 2^{56}:
When trying to improve the security of a block cipher, a tempting idea is to encrypt the data several times using multiple keys. One might think this doubles or even n-tuples the security of the multiple-encryption scheme, depending on the number of times the data is encrypted, because an exhaustive search on all possible combinations of keys (simple brute force) would take $2^{n \cdot k}$ attempts if the data is encrypted with k-bit keys n times.

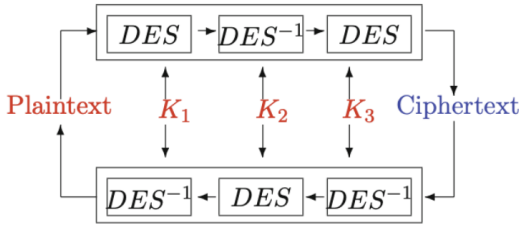


Figure 24: Triple DES

The **meet-in-the-middle attack** (MITM) is a generic attack that weakens the security benefits of using multiple encryptions by storing intermediate values from the encryptions or decryptions and using those to improve the time required to brute force the decryption keys. The attack attempts to find the keys by using both the range (ciphertext) and domain (plaintext) of the composition of several functions (or block ciphers) such that the forward mapping through the first functions is the same as the backward mapping (inverse image) through the last functions, quite literally meeting in the middle of the composed function. For example, although Double DES encrypts the data with two different 56-bit keys, Double DES can be broken with 2^{57} encryption and decryption operations. The MITM attack is the primary reason why 2DES is not used, and 3DES can be brute-forced by an attacker with 2^{56} space and 2^{112} operations.

7.3.6 Cryptanalysis of Block Ciphers

Can be done with:

- Exhaustive search
- Pre-computed intermediate values
- Divide and conquer

A block cipher should be resistant against differential and linear cryptanalysis. *explain what this is?*

7.4 Advanced Encryption Standard (AES)

AES replaced DES in the 1990s. Public design. AES is not a Feistel cipher, but a SPN. It is based on the Rijndael algorithm. Key features are:

- Each round has a key addition phase, a substitution phase (non-linear, confusion), and a permutation phase (for diffusion, avalanche effect)
- AES is built on a mathematical foundation (finite fields $\mathbb{F}_{2^8}, \mathbb{F}_2$)
- Encryption and decryption are distinct operations (do not rely *solely* on reverse ordering of the keys)
- Supports multiple key lengths: 128, 192, and 256 bits (and thus also blocks of these sizes)

7.4.1 Finite Fields and Arithmatic in AES

Elements of $\mathbb{F}_{2^8}$ are stored as bit vectors (or bytes) representing **binary polynomials**. From the hexadecimal representation of a byte, we can derive the polynomial representation. For example, the byte `0x57` is represented by the polynomial $x^6 + x^4 + x^2 + x + 1$. The polynomial representation is used for arithmetic operations in AES.

$$\texttt{0x57} = 5 \cdot 16 + 7 = 87$$

in decimal. The bit representation in binary is given by concatenating the bit representation of 5 and 7:

```
0101 0111
```

The bit pattern then corresponds to the binary polynomial, using the indexes where the bits are set to 1 as the exponents of the polynomial:

$$x^6 + x^4 + x^2 + x + 1$$

Arithmethic in the finite field $\mathbb{F}_{2^8}$ is a cornerstone of the AES algorithm. It is a Galois field with $2^8 = 256$ elements.

Field Construction

- $\mathbb{F}_{2^8}$ is constructed as a polynomial field over $\mathbb{F}_2$ (the field with two elements, 0 and 1).
- Elements of $\mathbb{F}_{2^8}$ are represented as polynomials of degree at most 7 with coefficients in $\mathbb{F}_2$. For example:
$$a(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0, \quad a_i \in \{0, 1\}.$$
- Addition and multiplication of elements are defined modulo an irreducible polynomial $m(x)$ of degree 8 over $\mathbb{F}_2$. In AES, the chosen irreducible polynomial is:
$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

Addition

- Addition in $\mathbb{F}_{2^8}$ corresponds to bitwise XOR of the coefficients of the two polynomials.
- Example:
$$(a_7x^7 + a_6x^6 + \cdots + a_0) + (b_7x^7 + b_6x^6 + \cdots + b_0) = (a_7 \oplus b_7)x^7 + \cdots + (a_0 \oplus b_0).$$

Multiplication

1. **Polynomial Multiplication**:
   - Multiply two polynomials $a(x)$ and $b(x)$ normally, treating them as polynomials over $\mathbb{F}_2$.
   - Example:
   $$a(x) = x^2 + 1, \quad b(x) = x^3 + x$$
   $$a(x) \cdot b(x) = (x^2 + 1)(x^3 + x) = x^5 + x^3 + x^3 + x = x^5 + x.$$
2. **Modulo Reduction**:
   - Reduce the result modulo the irreducible polynomial $m(x)$.
   - For example, if $a(x) \cdot b(x) = x^9 + x^5$, reduce it modulo $x^8 + x^4 + x^3 + x + 1$:
   $$x^9 = x \cdot x^8 \equiv x(x^4 + x^3 + x + 1) \mod m(x).$$
3. The final result after reduction is the product in $\mathbb{F}_{2^8}$.

Inversion

- Inversion (finding the multiplicative inverse) is critical in the AES S-box.
- The inverse of an element $a(x)$ in $\mathbb{F}_{2^8}$ is the unique element $b(x)$ such that:
$$a(x) \cdot b(x) \equiv 1 \mod m(x).$$
- This is typically computed using the Extended Euclidean Algorithm.

7.4.2 State Matrix

AES organizes the input plaintext (or ciphertext) as a 4x4 matrix of bytes called the state. Each entry in the matrix is one byte (8 bits), and the matrix evolves through each round of AES encryption or decryption. Each round key is held by a 4x4 matrix.

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}, K_i = \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix}$$

Mapping Input Plaintext to the State Matrix

- AES operates on 128-bit blocks (16 bytes) at a time.
- The 128-bit plaintext block is split into 16 bytes and arranged column-by-column into the state matrix.

For a plaintext block:

$$P = [\text{byte}_0, \text{byte}_1, \ldots, \text{byte}_{15}],$$

the state matrix is represented as:

$$S = \begin{pmatrix} \text{byte}_0 & \text{byte}_4 & \text{byte}_8 & \text{byte}_{12} \\ \text{byte}_1 & \text{byte}_5 & \text{byte}_9 & \text{byte}_{13} \\ \text{byte}_2 & \text{byte}_6 & \text{byte}_{10} & \text{byte}_{14} \\ \text{byte}_3 & \text{byte}_7 & \text{byte}_{11} & \text{byte}_{15} \end{pmatrix}.$$

7.4.3 AES Operations

The state array is manipulated using four main operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. These operations are repeated for multiple rounds, with the number of rounds depending on the key length.

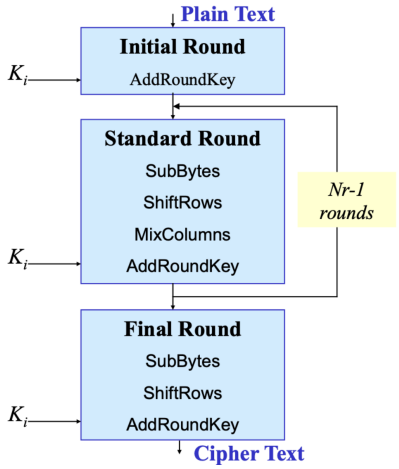


Figure 25: Overview of rounds in AES encryption

AddRoundKey This operatation takes the state matrix and XORs it, byte by byte, with the round key matrix. The inverse of this operation is clearly the same operation.

SubBytes
The SubBytes operation in the AES algorithm is a simple substitution step that replaces each byte in the state with another byte using a fixed substitution box (S-box). Criteria for S-Box:

- S-box must be resistant against linear and differential cryptanalysis
- S-box must be simple

- S-box must be invertible

ShiftRows The ShiftRows operation performs a cyclic shit on the state matrix. Each row is shifted by a different offset (row 1 with 0, row 2 with 1, etc.)

MixColumns This operation ensures that the rows in the state matrix "interact" with each other over a number of rounds; combined with the ShiftRows operation it ensures each byte of the output state depends on each byte of the input state. Each column in the state $[a_0, a_1, a_2, a_3]$ is considered one at a time to create a new column $[b_0, b_1, b_2, b_3]$. This is done by taking the polynomial

$$a(X) = a_0 + a_1 \cdot X + a_2 \cdot X^2 + a_3 \cdot X^3$$

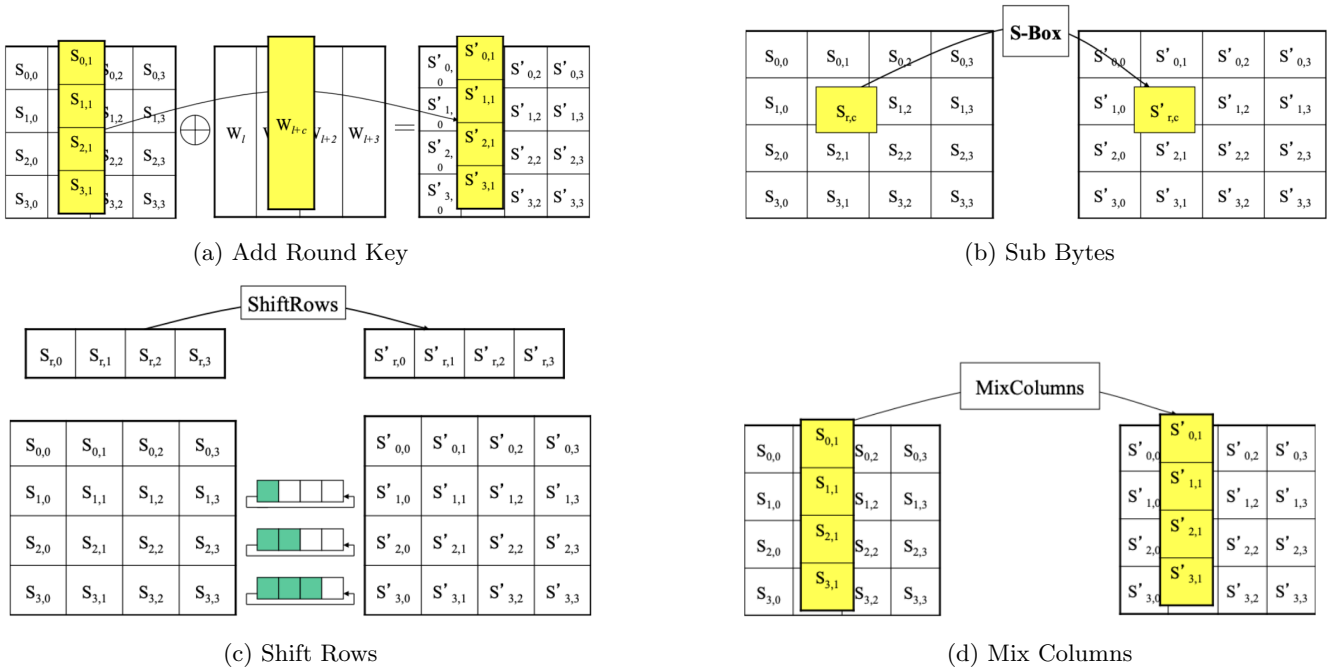
and multiplying it by the polynomial $c(X)$ that was chosen for AES to maximize diffusion.

$$c(X) = \texttt{0x02} + \texttt{0x01} \cdot X + \texttt{0x01} \cdot X^2 + \texttt{0x03} \cdot X^3$$

Then the modulo is taken with the invertible polynomial $M(X)$ to ensure invertibility of the MixColumns operation.

$$M(X) = X^4 + 1$$

This can be represented by a multiplication matrix (see book and slides) that is invertible in $\mathbb{F}_{2^8}$.



(a) Add Round Key

(b) Sub Bytes

(c) Shift Rows

(d) Mix Columns

Figure 26: AES Operations visualized

7.4.4 Key Scheduling

The AES key schedule generates a series of round keys from the initial secret key. These round keys are used in each round of AES encryption or decryption. Here, we focus on AES with 128-bit blocks.

The main key is 128 bits long, and we need to produce 11 round keys $K_0, ..., K_{11}$ all of which consist of four 32-bit words, each word corresponding to a column of the matrix K as described above. The secret key is split into 4-byte words: for AES-128, the key has 4 words W_0, W_1, W_2, W_3. AES requires a separate **round key** for each encryption round, here: 10 rounds $\rightarrow$ 11 round keys (each 16 bytes). The round keys are derived using a process called **key expansion**:

- **RotWord**:

- Rotate the bytes in a word to the left by one position.
- Example: $[b_0, b_1, b_2, b_3] \rightarrow [b_1, b_2, b_3, b_0]$.

- **SubWord**:
  - Substitute each byte of the word using the AES S-box (just like in the SubBytes step of AES).
- **RC**:
  - Add a constant value, called the **round constant**, to the first byte of the word. This ensures that each round key is different.

Finish this later, i dont really get it...

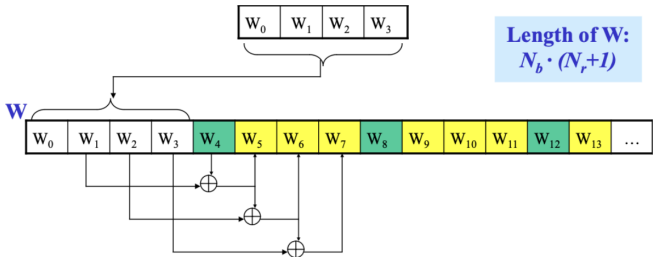


Figure 27: AES Key Scheduling

7.4.5 Encryption and Decryption

Algorithm 13.1: AES encryption outline

```
AddRoundKey(S, K_0).
for i = 1 to 9 do
    SubBytes(S).
    ShiftRows(S).
    MixColumns(S).
    AddRoundKey(S, K_i).
SubBytes(S).
ShiftRows(S).
AddRoundKey(S, K_10).
```

Algorithm 13.2: AES decryption outline

```
AddRoundKey(S, K_10).
InverseShiftRows(S).
InverseSubBytes(S).
for i = 9 downto 1 do
    AddRoundKey(S, K_i).
    InverseMixColumns(S).
    InverseShiftRows(S).
    InverseSubBytes(S).
AddRoundKey(S, K_0).
```

Figure 28: Pseudocode for AES Encryption and Decryption

7.5 DES vs. AES

Feature	DES	AES
Performance	Fast in hardware, slower in software	Fast in hardware and software
Key Length	One key length (not expandable)	Multiple key lengths (expandable in the future)
Block Length	One block length (not expandable)	Multiple block lengths
Number of Iterations	Many iterations (Feistel structure)	Relatively small amount of iterations

Table 3: Comparison of DES and AES

7.6 Modes of Operation

Block ciphers take an input of a fixed size, and output a ciphertext of this same size. Unless your message is exactly that length, we have to do something to our message to be able to use a block cipher. Originally, 4 standard modes:

- ECB (Electronic Codebook)
- CBC (Cipher Block Chaining)
- CFB (Cipher Feedback)
- OFB (Output Feedback)

Over the years, many more: CTR (Counter Mode)
Sync with slides

7.6.1 Electronic Codebook (ECB)

Naively split the message into blocks of the correct size, and encrypt each block separately. If you get to a message at the end that is too short, we pad it in some way. This is trivial to implement and it is quite fast too, especially if you have have a multi-core processor and can do this in a multi-threaded environment. However, it is not secure. If you have the same plaintext block, you will get the same ciphertext block. This is a problem, as it leaks information. The whole point of encryption is that you learn nothing about the ciphertext, and in this case you do.

- If $m_i = m_j$ then $c_i = c_j$, i.e. the same input block always generates the same output block.
- Deletion of parts of the message is possible without detection. There is no way to detect if the ciphertext has been tampered with during transmission or storage. An attacker can manipulate individual ciphertext blocks without affecting others, potentially altering the decrypted plaintext in a controlled manner.
- A **replay attack** is also possible: an attacker can capture and replay individual ciphertext blocks to manipulate the decrypted plaintext. For instance, in a scenario where ECB mode is used for encrypting financial transactions, an attacker could replay a captured ciphertext block representing a valid transaction to authorize multiple fraudulent transactions.

7.6.2 Cipher Block Chaining (CBC)

One way of countering the problems with ECB Mode is to chain the cipher, and in this way add context to each ciphertext block.

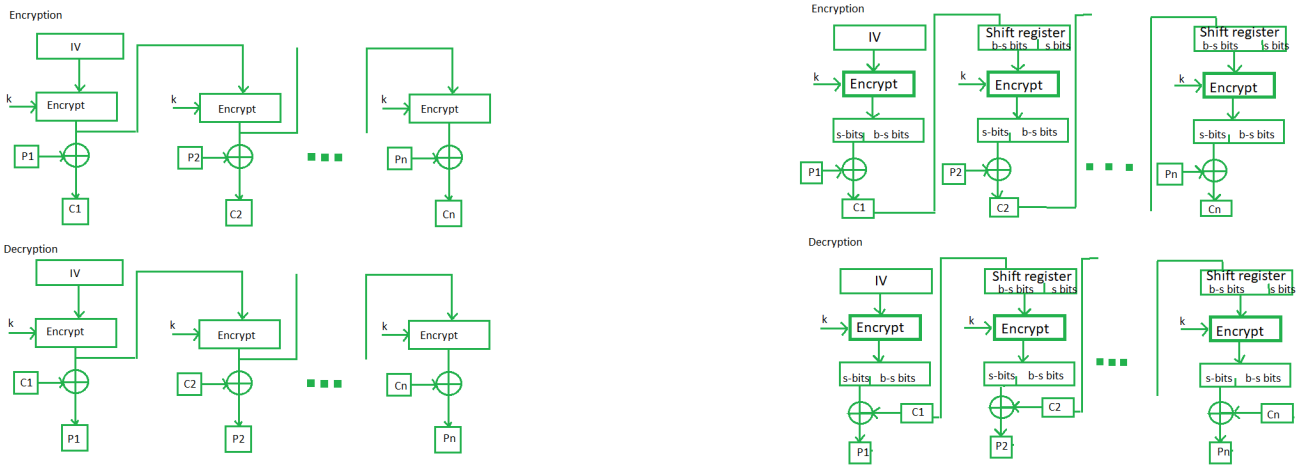
- Each block is XORed with the previous ciphertext block before encryption. This means that the same plaintext block will not generate the same ciphertext block.
- The first block is XORed with an **initialization vector** (IV) instead of a previous ciphertext block. The IV should be random and unique for each encryption operation.
- The IV must be known to the recipient to decrypt the message. It is typically sent along with the ciphertext.
- CBC mode is not parallelizable, as each block depends on the previous block. This can be a performance issue.
- If an attacker manipulates a ciphertext block during transmission, it will affect the decryption of that block and cause errors in the decryption of the subsequent plaintext block due to the chaining mechanism (bit-flipping resulting from XOR operations). This vulnerability can also lead to padding oracle attacks, where the attacker exploits information leaked during decryption about whether the padding of a plaintext is correct. By carefully modifying the ciphertext and observing the behavior of the decryption process, an attacker may infer details about the plaintext or even recover it entirely.

7.6.3 Output Feedback (OFB)

In OFB, encryption and decryption are performed by XORing the plaintext or ciphertext with a generated keystream.

1. Initialization:
   - A secret key K and an initialization vector IV are shared between the sender and receiver.
   - The IV is a random value used as the starting input to the block cipher to ensure that encryption is unique for each session.
2. Keystream Generation:
   - The IV is encrypted using the block cipher to produce the first keystream block:
   $$S_1 = E_K(IV)$$
   - Subsequent keystream blocks are generated by encrypting the previous keystream block:
   $$S_{i+1} = E_K(S_i)$$
   - This process is independent of the plaintext or ciphertext.
3. Encryption:
   - The plaintext blocks P_i are XORed with the corresponding keystream blocks S_i to produce the ciphertext blocks C_i:
   $$C_i = P_i \oplus S_i$$
4. Decryption:
   - Since XOR is symmetric, the ciphertext is XORed with the same keystream block to retrieve the plaintext:
   $$P_i = C_i \oplus S_i$$

Each block is processed independently, and there is no chaining between ciphertext blocks. Errors in a ciphertext block do not propagate to other blocks during decryption.



(a) Output Feedback Mode

(b) Cipher Feedback Mode

Figure 29: OFB and CFB Schemes

Advantages:

- Error containment: If a bit in the ciphertext is corrupted during transmission, only the corresponding bit in the plaintext is affected upon decryption. Other blocks remain unaffected.

- No padding required
- Pre-computed keystream: save computation time
- Resilience to reordering: since decryption does not depend on previous ciphertext blocks, reordering of ciphertext blocks does not affect decryption (as long as the corresponding block position is known).

Disadvantages:

- Key and IV reuse compromise security. XORing two ciphertexts reveals the XOR of the plaintexts.
- No error detection: if the ciphertext is tampered with, the error will go undetected during decryption.

7.6.4 Cipher Feedback (CFB)

Similar to OFB, but encryption and decryption rely on a feedback mechanism that incorporates ciphertext into the keystream generation.

1. Initialization:
   - A secret key K and an initialization vector IV are shared between the sender and receiver.
   - The IV serves as the starting input to the encryption process.
2. Encryption:
   - The block cipher encrypts the IV to produce the first keystream block:
   $$S_1 = E_K(IV)$$
   - The plaintext block P_1 is XORed with S_1 to produce the ciphertext:
   $$C_1 = P_1 \oplus S_1$$
   - The ciphertext C_1 is fed back into the block cipher to generate the next keystream:
   $$S_2 = E_K(C_1)$$
3. Decryption:
   - The ciphertext block C_i is XORed with the keystream block S_i to recover the plaintext:
   $$P_i = C_i \oplus S_i$$
   - The ciphertext C_i is fed back into the block cipher for subsequent keystream generation.

Key properties:

- Error propagation: A bit error in a ciphertext block affects decryption of the current plaintext block and the next block.
- No padding required: CFB works as a stream cipher and does not require padding for incomplete blocks.
- Loss of synchronization: Dropped ciphertext blocks cause decryption errors in all subsequent blocks.

7.6.5 Counter Mode (CTR)

So what we want is: for the messages to decrypt into different ciphertexts, we dont want to divulge that two messages are the same. Also, we want it to be parallelizable.

We no longer encrypt the message. We encrypt a countermessage, and then XOR this ciphertext with the message, converting the block cipher into a stream cipher. We also use a **nonce**: a random number used *only once*.
The block cipher becomes a stream cipher: only now we XOR the keystream with the plaintext in blocks.

1. Initialization:

- A secret key K and an initialization vector IV (or nonce) are shared between the sender and receiver.
- The IV is combined with a counter, which is incremented for each block.

2. Keystream Generation:
   - The block cipher encrypts the IV combined with the counter to produce the keystream:

$$S_i = E_K(\text{Counter}_i).$$

3. Encryption:
   - Each plaintext block P_i is XORed with the keystream S_i to produce the ciphertext:

$$C_i = P_i \oplus S_i.$$

4. Decryption:
   - The ciphertext block C_i is XORed with the same keystream S_i to recover the plaintext:

$$P_i = C_i \oplus S_i.$$

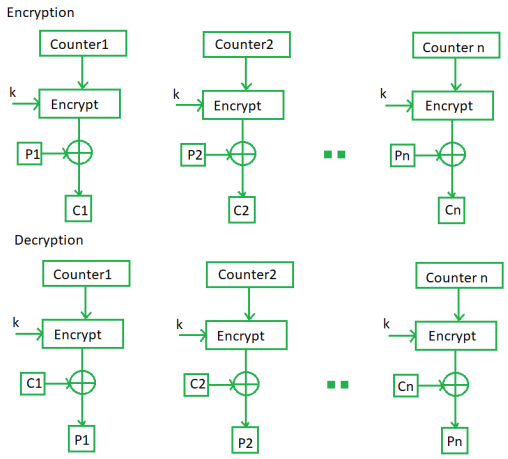


Figure 30: Counter Mode (CTR) operation

Advantages:

- High throughput due to parallelism
- No error propagation by errors in ciphertext blocks
- Random access: decryption of any block can be performed independently without processing earlier blocks

Disadvantages:

- Key and IV reuse compromise security
- CTR mode does not inherently provide integrity checking. If the ciphertext is modified, the changes will go undetected unless an additional mechanism (e.g., a Message Authentication Code, or MAC) is used.

Note: you have to be extremely careful about not reusing the nonce value. Say you have a 96-bit nonce, that puts a hard limit on the number of messages you can send before you have to change the key.

7.6.6 Security: how to achieve CCA?

Finish with slides

8 Lecture 8

8.1 Hash Functions

Hash functions are used for integrity protection.

- Blackbox function that takes an input and produces a fixed-size output
- Hash functions are deterministic, i.e., same input always produces the same output
- No matter how much data you give it, alwahys produces a fixed-size output
- Hash functions are fast to compute
- You can't reverse or predict the hash (one-way functions)
- Hash functions are collision resistant (only for SHA-256, not for MD5)

Asymmetric digital signature: sign hash of the document MAC: Message Authentication Code, combine hash of message with secret key to obtain a MAC Key generation

8.1.1 Preimage Resistance

Hash functions receive an arbitrary length bit string. They output a fixed length string called the hash value, digest, or hashcode. Cryptographic hash functions are one-way: easy to compute y given x but infeasible to find x given $y : H : \{0,1\}^* \to \{0,1\}^n$, with $H(x) = y$.

Definition 54. *A hash function H is **preimage resistant** if given y, it is computationally infeasible to find x such that $H(x) = y$. Given an output of t bits, it should take $O(2^t)$ time to find a preimage (x is preimage of y).*

Definition 55. *A hash function H is **second preimage resistant** if given m, it is computationally infeasible to find m' such that $H(m) = H(m')$.*

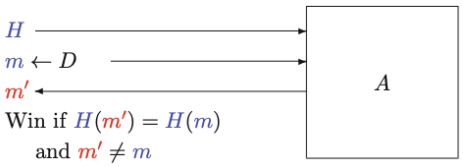


Figure 31: Security game for second preimage resistance

This security game can be won by the adversary, because the domain of the message space (arbitrary length) is huge compared to the domain of the output. The output is fixed size, let's say 128. Then the domain is "only" $2^1 28$. There will be a number of collisions.

8.1.2 Collision Resistance

Assume now that the domain is much larger than the co-domain. Given H, it should be infeasible to compute m and m' such that $H(m) = H(m')$.

Definition 56. *A hash function H is **collision resistant** if it is computationally infeasible to find m and m' such that $H(m) = H(m')$. Given an output of t bits, it should take $O(2^{t/2})$ time to find a collision.*

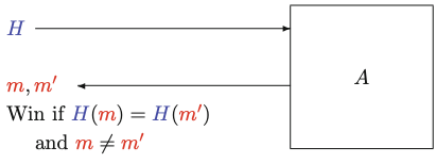


Figure 32: Security game for collision resistance of a function

This security game can also be won by the adversary! There are several collusions: pigeonhole principle. We solved the same problem in PRFs by using a keyed version of the functions. However, we need unkeyed hash functions! This is because we need to be able to verify the integrity of the data without having the key (public use, no secret). Keyed hash functions are for security and authentication.

Definition 57. *A function H is said to be collision resistant (by human ignorance) or HI-CR secure if it is believed to be infeasible to write down a collision for the function, i.e. two elements in the domain mapping to the same element in the codomain.*

The probability to have a collision is $\sqrt{2^{t+1}}$ (Birthday paradox).

In summary, a cryptographic hash function:

1. Preimage Resistant: It should be hard to find a message with a given hash value
2. Second Preimage Resistant: Given one message it should be hard to find another message with the same hash value
3. Collision Resistant: It should be hard to find two messages with the same hash value

(1) is weaker than (2) and (3), (2) is weaker than (3). Some more terminology:

- one-way = preimage + second preimage resistant
- weak collision resistant = second preimage resistant
- strong collision resistant = collision resistant
- OWHF = one-way hash function = preimage and second preimage resistant
- CRHF = collision resistant hash function = collision resistant and second preimage resistant

8.2 Padding

Even though the input size can be arbitrary, many hash algorithms process data in fixed-size blocks (e.g., 512-bit blocks for SHA-256). If the input data is not a multiple of the block size, padding is added to make the data fit perfectly into blocks. Padding also helps to include the original message length in the hash calculation, which is crucial for preventing certain types of attacks (like length-extension attacks).

When you divide a message into smaller blocks, how can you pad it? Given an l bit message m, and block size b, we want to have m of length kb:

$$m||\text{pad}_i(|m|, b)$$

In theory, padding can be applied at the beginning or the end. In practice, all algorithms pat at the end.

8.2.1 Padding Methods

We define 5 of them:

- **Method 0:** $v = b - |m| \mod b$ and padding is v zeros: $m||0^*$. In other words, the message is padded with zeros until it is a multiple of the block size
- **Method 1:** $v = b - (|m| + 1) \mod b$ and padding is: $m||10^*$.
- **Method 2:** $v = b - (|m| + 65) \mod b$ and padding is: $m||10^*||L$, where L is a 64-bit integer encoding of $|m|$
- **Method 3:** $v = b - (|m| + 64) \mod b$ and padding is: $m||0^*||L$
- **Method 4:** $v = b - (|m| + 2) \mod b$ and padding is: $m||10^*1$

Any can be used, apart from method 0, but will effect security. Method 1 is also not good.

For SHA-256: The input message is padded by appending a 1 bit, followed by enough 0 bits, so that the total length is 64 bits short of a multiple of 512. Then, the original message length (in bits) is added as a 64-bit value at the end of the padded message.

8.3 Merkle-Damgård Construction

Definition 58. *A **compression function** is a hash function that takes an input and produces a fixed-size output. It is applied iteratively to compress larger inputs into a smaller output. After all blocks are processed, the final state becomes the output of the hash function.*

The MD construction is a method for building cryptographic hash functions. The construction breaks down the process of hashing into a series of steps where the input data is processed in fixed-size blocks. The key idea is to iteratively compress these blocks using a compression function.

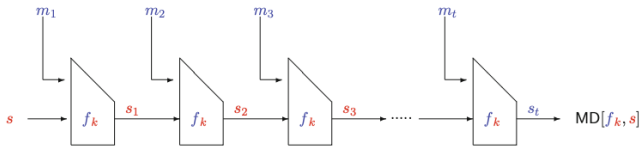


Figure 33: The Merkle-Damgård Construction MD$[f_k, s]$, where s is the internal state

If the compression function is secure, then the hash function is secure too.

8.3.1 MD Algorithm

- Divide the innput message into fixed-size blocks. Apply padding if the message length is not a multiple of the block size.
- The process begins with an initial value (IV), a predefined set of constants.
- Iteratively apply the compression function f_k to each block, taking the current state and block of data as input. The output of the compression function becomes the new state passed to the next iteration.
- After all blocks are processed, the final state becomes the final hash output.

Algorithm 14.1: Merkle–Damgård construction

Pad the input message m using a padding scheme, so that the output is a multiple of ℓ bits in length.
Divide the input m into t blocks of length ℓ bits. $m_1, \ldots, m_t$.
$s_0 \leftarrow s$.
for $i = 1$ **to** t **do**
  $s_i \leftarrow f_k(m_i \| s_{i-1})$.
return s_t.

8.3.2 Benefits of MD Construction

- Design makes streaming possible
- Hash function analysis becomes compression function analysis
- Analysis is easier because the domain of CF is finite

8.3.3 Properties of MD Construction

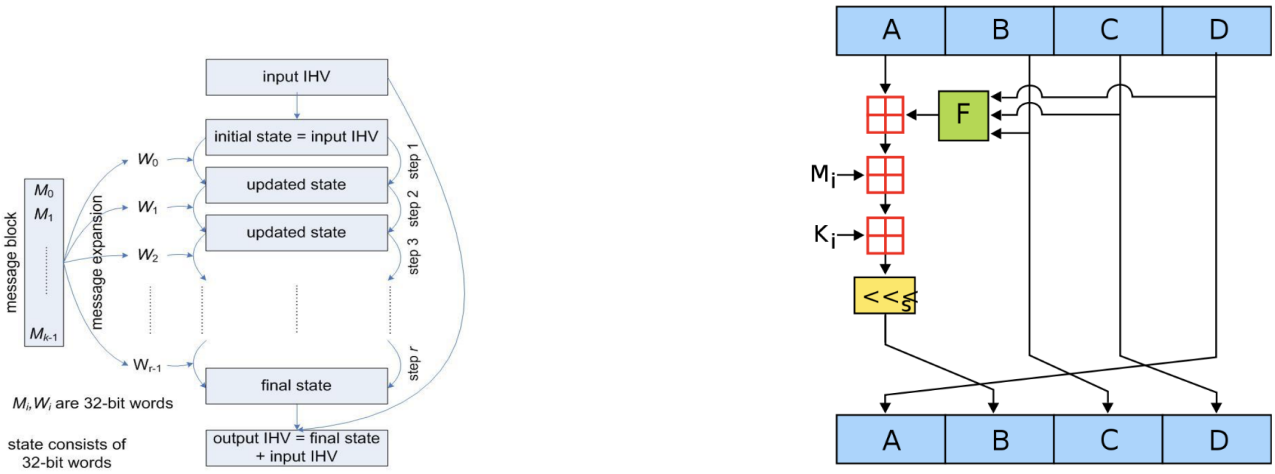
Collision resistance of the hash function depends on the padding method. Imagine there are two messages $m = 0b0$ and $m' = 0b00$. Using method 0, both of them will be mapped to $0b0000 \rightarrow$ collision. This means that the original message can be $0b0, 0b00, 0b000, 0b0000$. All of them will be mapped to the same value. Thus, MD-based hash function should *not* use padding method 0. Use method 2 instead.

8.3.4 Security properties under different scenarios

1. **s is fixed to an IV, f_k is also fixed:**
   - s is initialized to a fixed Initial Value (IV), and f_k, the compression function, is also fixed.
   - If f is **HI-CR secure** (Highly Iterated Collision-Resistant), then the hash function $H(m)$, constructed using the MD construction, is also collision-resistant.
   - **Practical Use:** This describes the standard cryptographic hash functions where the IV and compression function are predefined.
2. **s_0 is a key, f_k is a fixed function:**
   - Here, s_0 is treated as a **key** (instead of a fixed IV), while f_k, the compression function, remains fixed.
   - If f is **HI-CR secure**, the keyed hash function $H_s(m)$ (where s_0 is the key) is also collision-resistant.
   - **Practical Use:** This construction aligns with setups like HMAC, where a secret key is combined with the message for added security.
3. **f_k is from a PRF family, s is fixed:**
   - In this case, f_k comes from a **PRF family**, and s (the initial value) is fixed.
   - If f is **CR secure** (Collision-Resistant), then $H_k(m)$, constructed using the MD process, is also collision-resistant. However, this setup has limited practical implications.
   - **Practical Use:** This is more of a theoretical observation, as using a PRF family for f_k doesn't add significant value for hashing in practice.

8.4 MD-4

MD-family is a series of hash functions: MD construction with a (fixed) *unkeyed* compression function f. In MD-4 (128-bit output), the function f has 3 rounds of 16 steps each.



(a) MD-4 compression function

(b) Steps in MD-4

Figure 34: MD-4 Compression Function and Steps

MD-4 is not secure, and should not be used in cryptographic applications. Vulnerable to collision attacks, these days collisions can be found in seconds. Also weak against (second) preimage attacks.
Risks associated with MD-4:

- Digital signatures: If MD4 is used in signatures, an attacker can create fake documents with valid signatures by exploiting collisions.
- Data integrity: MD4 cannot reliably ensure data integrity, as attackers can manipulate data without detection.

MD4 has been superseded by much stronger algorithms, such as:

- MD5 (although MD5 is also now considered insecure)
- SHA-1 (also insecure, with known collision attacks since 2017)
- SHA-2 and SHA-3 (currently secure)

8.5 Birthday Paradox

The **birthday paradox** refers to the counterintuitive probability that in a group of just 23 people, there is about a 50% chance that two of them share the same birthday.

8.5.1 Why does this happen?

- Instead of calculating the chance of one specific match, we consider **any pair** of people matching, which grows quickly as the group size increases.
- The total number of pairs increases exponentially with the number of people:

$$\binom{n}{2} = \frac{n(n-1)}{2}.$$

Given a set of $t \geq 10$ elements, take a sample of size k (drawn with repetition) in order to get a probability $/geq\frac{1}{2}$ on a collision. k has to be larger than $1.2\sqrt{t}$. Consequence: if $F : A \to B$ is a random function, and $\#A >> \#B$, then one can expect a collision after $\sqrt{\#B}$ random function calls.

8.5.2 In Cryptography

- The birthday paradox is used to explain the likelihood of **collisions** in hash functions.
- For a hash function with n-bit outputs:
  - A collision is expected after approximately $\sqrt{2^n}$ hashes, i.e., $\approx 2^{n/2}$, due to the birthday paradox.
- This highlights why hash functions need **large output sizes** (e.g., 256 bits) to prevent collisions.

8.5.3 Random vs. Meaningful Birthdaying

Definition 59. ***Random Birthdaying:*** *searching for any two inputs that produce the same output in a hash function, without any specific requirements on the inputs. Do an exhaustive search on $1/2n$ bits. The messages will be random and not meaningful. Demonstrates the general weakness of a hash function if collisions can be found easily.*

Definition 60. ***Meaningful Birthdaying:*** *finding a collision where at least one of the inputs has specific, meaningful content. Start with two meaningful messages m_1, m_2 for which you want to find a collision. Identify $1/2n$ independent positions where the messages can be changed at bit-level without changing the meaning, e.g. tab—space, space—newline, etc. Do random search on those positions.*

Note: these are brute-force attacks. *add an example here?*

8.5.4 Implementing Birthdaying

- **Naïve**
  - Store $2^{n/2}$ possible messages for m_1 and $2^{n/2}$ possible messages for m_2 and check all 2 pairs.
- **Less naïve**
  - Store $2^{n/2}$ possible messages for m_1, and for each possible m_2, check whether its hash is in the list.
- **Smart:** Pollard-ρ with Floyd's cycle-finding algorithm
  - Computational complexity still $O(2^{n/2})$ – but only constant small storage required.

8.6 MAC and HMAC

HMAC (Hash-based Message Authentication Code) is a cryptographic algorithm used to ensure both the integrity and authenticity of a message. It combines a cryptographic hash function (like SHA-256) with a secret key.

8.6.1 Insecurity of Naïve Keyed Hash Construction

A naïve attempt to create a keyed hash using an *unkeyed* hash function is:

$$t = H(k||m||\text{pad}_i(|k| + |m|, b))$$

where

- H: unkeyed hash function
- k: secret key
- m: message
- pad_i: padding to ensure the input meets the block size of
- This approach involves concatenating the key k, message m and some padding before hashing.

The digest t is then computed as:

$$t = MD[f, s](k||m||\text{pad}_i(|l| + |m|, l))$$

where $MD[f, s]$ is the MD-hash function, and l is the intermediate state.

The adversary can exploit this construction. If she can query t, she can then construct valid keyed hashes by appending additional data m' to the message m. This is called a **length extension attack**. Without knowing the secret key k, the attacker can extend the hash to new messages and generate valid keyed hashes.

A valid keyed hash for the new message can be computed as:

$$m||pad_i(|l| + |m|, l)||m'$$

This construction allows an attacker to manipulate and extend the authenticated data.
remove first block (key k), append corrupted message m'??

8.6.2 Secure Keyed Hash Construction

We use nested MAC to construct a secure keyed hash function. The construction is as follows:

$$\text{NMAC}_{k_1,k_2}(m) = F_{k_1}(F_{k_2}(m))$$
$$F_{k_1} = f((x||\text{pad}_2(l + |x|, l))||k_1) = MD[f, k_1]^*(x)$$
$$G_{k_2} = MD[f, k_2]^*(m)$$

But, we want one hash function with one key, so we construct HMAC as follows:

$$\text{HMAC}_k(m) = H\left((k \oplus \text{opad})||H\left((k \oplus \text{ipad})||m\right)\right)$$
$$= MD[f, IV]((k \oplus \text{opad})MD[f, IV]((k \oplus \text{ipad}))||m)$$

or simplified:

$$\text{Inner hash} = H(\text{ipad}||M)$$
$$\text{HMAC} = H(\text{opad}||\text{Inner hash})$$

Steps in the process:

1. **Key Preparation**:
   - If the key is longer than the block size of the hash function, it is hashed first to reduce its size.
   - If the key is shorter than the block size, it is padded with zeros to match the block size.
2. **Inner Hash Calculation**:
   - XOR the key with a padding called the *inner pad* (ipad).
   - Concatenate the result with the original message.
   - Hash the resulting data using the chosen hash function.
3. **Outer Hash Calculation**:
   - XOR the key again, but this time with a different padding called the *outer pad* (opad).
   - Concatenate the result of the inner hash calculation with the outer padded key.
   - Hash the resulting data.
4. **Output**: The final hash value after the second hash computation is the HMAC, which serves as the authentication code.

8.6.3 Security of HMAC

The security of HMAC depends on the strength of the underlying hash function and the secrecy of the key. HMAC is resistant to various attacks, such as:

- Brute-force attacks.
- Collision attacks.
- Length-extension attacks (common in some hash functions).

Suppose a client wants to send a message to a server, and both share a secret key:

1. The client computes the HMAC of the message using the shared key.
2. The client sends both the message and the HMAC to the server.
3. The server computes the HMAC of the received message with the same key.
4. If the computed HMAC matches the received HMAC, the server knows the message is authentic and unchanged.

8.7 Sponge Functions

Sponge functions are a modern technique to create hash functions, MACs, key derivation functions, and more. They are based on the sponge construction, which is a generalization of the Merkle-Damgård construction.

It operates by absorbing input data and squeezing out output. The input and output can be of arbitrary length. A sponge function works on an internal state, which is divided into two parts:

- Capacity (c): The part that controls security.
- Rate (r): The part used for absorbing input and squeezing output.

Operations:

1. **Absorbing Phase**: The input is divided into blocks and absorbed into the internal state by XORing with the rate portion, followed by a transformation (permutation) of the state:

   $$S = P(S \oplus M_i)$$

   where S is the state, M_i is the input block, and P is the permutation function.

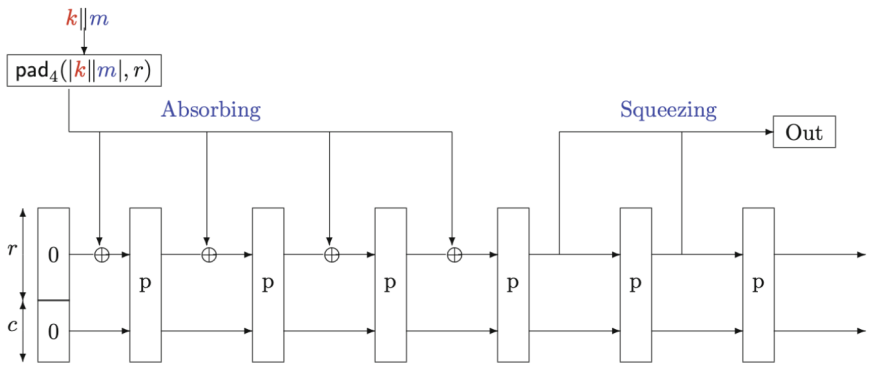


Figure 35: Sponge construction SP[p]

2. **Squeezing Phase**: After all input is absorbed, the output is extracted from the rate portion of the state. Additional squeezing rounds can be performed if more output is needed:

$$O = S_{\text{rate}} \quad \text{(extract the rate portion)}$$

The security of a sponge function is based on the size of the capacity. Larger capacities make the function more resistant to collission and preimage attacks.

Example: SHA-3 (Keccak) SHA-3, based on the Keccak sponge construction, has an internal state of 1600 bits, divided into 1024 bits for the rate and 576 bits for the capacity. The absorbing phase processes input in blocks of 1024 bits, and output is squeezed from the rate portion.

9 Lecture 9

Symmetric encryption algorithms are fast and efficient because they use a single shared key for both encryption and decryption. However, the **key distribution problem** arises because both parties must securely share the secret key before any communication can occur. Secure transmission of the key, no built-in key exchange mechanism in the encryption scheme, scalability in systems with multiple uses (number of unique keys grows exponentially), risk of compromise, etc.

Solution: use **public-key cryptography** (asymmetric encryption) to securely exchange keys. Public key encryption algorithms are slow, but key distribution is easier. Public key is broadcasted, private key is kept secret.

9.1 Naive RSA Algorithm

Based on the difficulty of factoring large numbers that are the product of two (large) prime numbers. Multiplying these two numbers is easy, but determining the original prime numbers from the total – or factoring – is considered infeasible due to the time it would take using even today's supercomputers.

Given a large composite number N, find d given e. It works as follows:

1. Alice chooses two large primes p and q
2. Alice computes $N = pq$
3. Alice chooses e such that $\gcd(e, (p-1)(q-1)) = 1$, i.e., e is relatively prime to $(p-1)(q-1)$, which is the totient function of N
4. Alice computes d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$, i.e., d is the modular multiplicative inverse of e modulo $\phi(N)$

The public key is (N, e) and the private key is (N, d), (p, q, d), (p, q), or (d). *why do all these work?*

9.1.1 Encryption and Decryption

To encrypt a message m (converted to a numeric value), compute:

$$C = m^e \mod N$$

where C is the ciphertext. To decrypt the ciphertext, compute:

$$m = C^d \mod N$$

The RSA algorithm works due to the following:

$$x^{(p-1)(q-1)} \equiv 1 \pmod{N}, \quad \text{for all } x \in \mathbb{Z}/N\mathbb{Z}^*,$$

where $\mathbb{Z}/N\mathbb{Z}^*$ is the set of integers coprime to N. This ensures that certain powers of x "wrap around" in modular arithmetic (Euler's theorem). The public key exponent e and the private key exponent d are chosen such that:

$$e \cdot d \equiv 1 \pmod{\phi(N)}.$$

This implies:

$$e \cdot d - s \cdot \phi(N) = 1,$$

where s is an integer. Thus, d is the modular multiplicative inverse of e modulo $\phi(N)$. Substituting $c = m^e$ in the decryption equation, we get:

$$m = (m^e)^d \pmod{N}.$$

Since $e \cdot d = 1 + s \cdot \phi(N)$, we can rewrite this as:

$$m^{ed} = m^{1+s\cdot\phi(N)} = m^1 \cdot (m^{\phi(N)})^s.$$

By Euler's theorem, $m^{\phi(N)} \equiv 1 \pmod{N}$, so:

$$m \cdot 1^s = m \pmod{N}.$$

9.1.2 Example of RSA

Let$p = 47$, $q = 59$, and $N = p \cdot q = 2773$. Then $\phi(N) = (p-1)(q-1) = 2668$. We pick $e = 17$. Now, find d such that $e \cdot d \equiv 1 \pmod{\phi(N)}$:

$$17 \cdot d \equiv 1 \pmod{2668}, \quad \text{so } d = 157.$$

Plaintext Message (M): `ITS ALL GREEK TO ME.` Convert to numeric format:

$$M = 0920\,1900\,0112\,1200\,0718\,0505\,1100\,2015\,0013\,0500.$$

Ciphertext (C) is calculated as:

$$C = M^e \pmod{N}.$$

Result:

$$C = 0948\,2342\,1084\,1444\,2663\,2390\,0778\,0774\,0219\,1655.$$

Encrypt a portion:

$$920^{17} \pmod{2773}.$$

Decrypt a portion:

$$948^{157} \pmod{2773}.$$

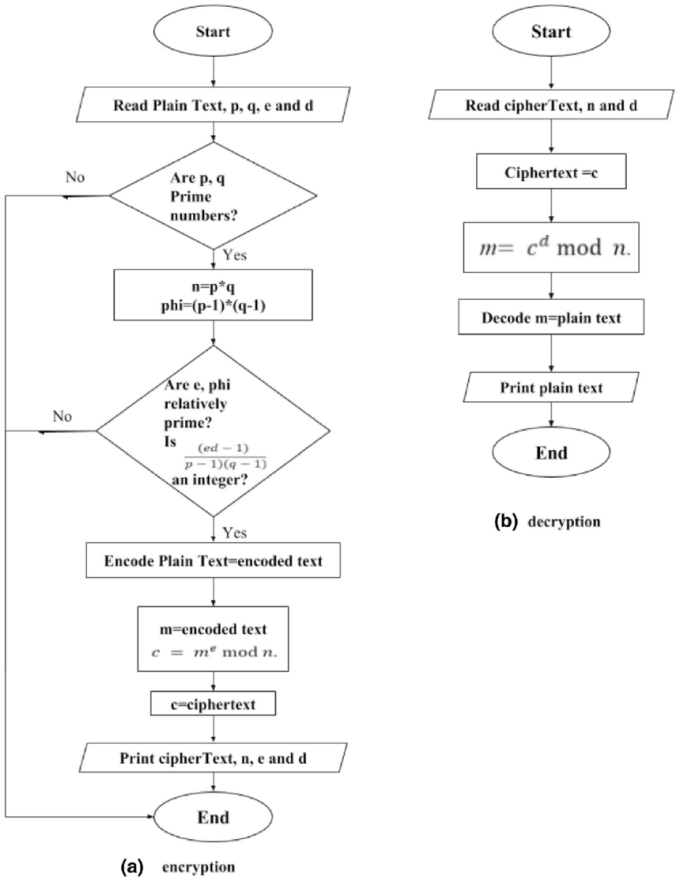


Figure 36: RSA Algorithm

9.2 Security of RSA

Computing d given e and N is equivalent to factoring N into its prime factors p and q. Thus it is no harder than factoring N. Current suggestion: modulo need to be 2048 bits long. RSA is OW-CPA, but not IND-CPA secure, because:

- Deterministic encryption: same plaintext always encrypts to the same ciphertext

- No randomness
- Mathematical structure leaks information:

$$(M_1 \cdot M_2)^e = M_1^e \cdot M_2^e$$

  Atttacker can gain information about plaintexts from ciphertext relationships
- Chosen plaintext attacks: attacker can guess relationshup between plain- and cipher text due to deterministic nature and lack of padding

But, deterministic encryption is not the only problem: RSA is **malleable** due to homomorphism.

Definition 61. ***Homomorphic Property:*** *an encryption scheme has the (multiplicate) homomorphic property if given the encryptions of m_1 and m_2 we can determine the encryption of $m_1 \cdot m_2$ without knowing m_1 or m_2.*

RSA is multiplicatively homomorphic:

$$(m_1 \cdot m_2)^e = ((m_1^e \pmod N) \cdot (m_2^e \pmod N)) \pmod N$$

The naive RSA is *not* OW-CCA secure. Remember that we hae a decryption oracle:

$$c^* = (m^*)^e \pmod N$$
$$c = 2^e \cdot c^*$$
$$\frac{m}{2} = \frac{c^d}{2} = \frac{(2^e \cdot c^*)^d}{2}$$
$$= \frac{2^{ed} \cdot (c^*)^d}{2} = \frac{2 \cdot m^*}{2} = m^*$$

9.2.1 How to make RSA IND-CPA secure?

To achieve **IND-CPA** security, RSA requires modifications, such as the use of **randomized padding schemes**. Below are two common approaches:

1. RSA-OAEP (Optimal Asymmetric Encryption Padding)
   - OAEP introduces **randomness** to the plaintext before applying the RSA encryption formula.
   - This randomness ensures that even if the same plaintext is encrypted multiple times, the resulting ciphertexts will differ.
   - RSA with OAEP is considered **IND-CPA secure in practice**.
2. Hybrid Encryption
   - RSA is often combined with **symmetric encryption** in real-world protocols.
   - RSA is used to encrypt a randomly generated **symmetric key**, ensuring IND-CPA security for the key exchange.
   - The symmetric key is then used to encrypt the actual message, leveraging the efficiency of symmetric encryption.

9.3 Rabin Encryption

Public key encryption scheme based on integer factoriziation. It is similar to RSA, but the encryption and decryption functions are different. Rabin encryption is based on a **trapdoor function**, having the advantage that inverting it is as hard as factoring intergers. RSA however, lacks this equivalence.
OW-CPA secure based in factoring problem. Mapping is not injective. *what does that mean?*

9.3.1 Key Generation

Choose two large prime numbers p and q, such that $p \equiv 3 \pmod 4$ and $q \equiv 3 \pmod 4$. Compute $N = p \cdot q$, where N is the public modulus. The **public key** is N, and the **private key** is (p, q). So, everyone can encrypt a message using N, but only the owner of p and q can decrypt it. There is no need of N at the receiver side.

9.3.2 Encryption and Decryption

To encrypt a message M (converted to a numeric value M such that $0 \leq M < N$):

$$C = M^2 \pmod{N}.$$

C is the ciphertext.

Given the ciphertext C, the decryption process involves finding the **square roots** of C modulo N. Using the Chinese Remainder Theorem (CRT), the decryption yields **four possible solutions**:

$$m_p = \sqrt{c} \pmod{p} = c^{(p+1)/4} \pmod{p}$$

$$m_q = \sqrt{c} \pmod{1} = c^{(1+1)/4} \pmod{1}$$

$$m_1, m_2, m_3, m_4.$$

The correct plaintext m out of the 4 possible ones must be determined using additional information or context.

9.3.3 Trapdoor

The trapdoor here is the ability to efficiently compute square roots $N = p \cdot q$ when you know the two prime factors. You can use the CRT to compute the sqaure roots efficiently.

To encrypt, the sender computes the ciphertext C as:

$$C = M^2 \pmod{N}.$$

Given only N, recovering M from C requires computing the square root of $C \pmod{N}$, which is difficult unless the factors p and q are known.
When the receiver knows the private key (the factors p and q):

- The receiver solves two modular equations:

  $$M^2 \equiv C \pmod{p}, \quad M^2 \equiv C \pmod{q}.$$

  Since p and q are primes, these equations can be solved efficiently using modular arithmetic techniques (such as modular square root algorithms).

- Using the CRT, the receiver combines the solutions modulo p and q to compute four possible square roots modulo N.

The decryption produces **four possible square roots** because:

- For a given p, there are two solutions: $M \pmod{p}$ and $-M \pmod{p}$.
- Similarly, for q, there are two solutions: $M \pmod{q}$ and $-M \pmod{q}$.

Using the CRT, these combine to produce four distinct solutions modulo N. The receiver must use additional context (e.g., padding) to identify the correct plaintext.

- Without knowing p and q, the decryption problem is as hard as factoring N. Factoring large composite numbers is computationally infeasible with current algorithms for sufficiently large N, making this a secure one-way function.
- Knowing p and q provides a "backdoor" (the trapdoor) to efficiently decrypt the ciphertext by finding square roots modulo N.

Analogy with RSA

- In RSA, the trapdoor is the knowledge of $\phi(N) = (p-1)(q-1)$ (or equivalently p and q), which allows the computation of the private key d, the modular inverse of the public key e modulo $\phi(N)$.
- In Rabin, the trapdoor is simply the knowledge of p and q, which enables efficient decryption by computing square roots modulo N.

9.3.4 Example

Let $p = 127$ and $q = 131$, so $N = p \cdot q = 16637$. The public key is N, and the private key is (p, q).

Let $m = 4410$ (numnerical value). Encryption gives:

$$c = m^2 \pmod{N} = 16084$$

Decryption gives:

$$m_p = \sqrt{c} \pmod{p} = \pm 35,$$

$$m_q = \sqrt{c} \pmod{1} = \pm 44$$

$$s = \sqrt{c} \pmod{N} = \pm 4410, \text{and} \pm 1616$$

So the message can be 1616, 4410, 1227, 15021.

9.3.5 Security of Rabin

- OW-CPA secure
- Not OW-CCA secure (malleable)
- Not IND-CPA secure (deterministic)

Explain a bit more why for each one
add a comparison table of RSA and Rabin?

9.4 Naive RSA Signature and Hashing

The Naive RSA Signature scheme is a basic implementation of digital signatures using the RSA cryptosystem.

Definition 62. *A **digital signature** is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature on a message gives a recipient confidence that the message came from a sender known to the recipient. A digital signature ensures authenticity, integrity, and non-repudiation of the message. The RSA signature scheme achieves this using the private and public keys of the RSA cryptosystem.*

The Naive RSA signature is called "naive" because it lacks any padding or hashing. This makes it:

- Inefficient for large messages: Large plaintexts M directly require modular exponentiation.
- Insecure against certain attacks: If M is small, the signature can leak information about the private key d. Direct signatures on raw messages without hashing can lead to forgery attacks (e.g., the existential forgery attack).

Sender "signs" a message by decrypting it with their private key:

$$s \leftarrow m^d \pmod{N}$$

The receiver "verifies" the signature by encrypting it with the sender's public key:

$$m \leftarrow s^e \pmod{N}$$

We need to check the validity of the signature. Padding is required, because:

- Without proper padding, certain attacks (such as **existential forgery**) can exploit weaknesses in the signature scheme to create forged signatures.
- Padding ensures the message m is properly formatted, making it more secure and resistant to attacks.
- The message m is padded to match the required input size for the cryptographic signature algorithm.

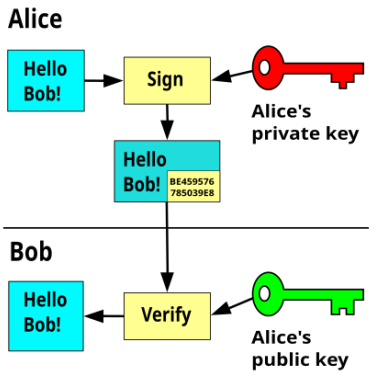


Figure 37: Digital Signature

9.4.1 Steps for Padding

Steps for padding: m: The message, represented in t bits. N: The modulus, represented in k bits, where $t < k - 32$.

1. Pad m with **zeros** on the left to make it a multiple of 8 bits.
2. Add $\frac{k-t}{8}$ additional bytes on the left:
   - Begin with '00': A leading zero.
   - Add '01': A start delimiter.
   - Insert a sequence of 'FF' bytes.
   - Include '00': A delimiter before the actual message.
3. The final padded message m is:

$$m \leftarrow 00 \parallel 01 \parallel FF \parallel FF \ldots \parallel FF \parallel 00 \parallel m.$$

9.4.2 Forgery Attacks

Prevent Existential Forgery:

- Existential forgery occurs when an attacker can create a valid-looking signature without the private key.
- Proper padding prevents attackers from trivially guessing valid signatures.

Padding also prevents againts **selective forgery** attacks, where an adversary aims to forge a signature for a specific message m of their choice. Suppose we have a signing oracle, which can compute signatures for any message, using the private key. The adversary wants to obtain a valid signature s of target message m. She can generate a random message $m_1 \in Z/NZ*$. The attacker calculates:

$$m_2 = \frac{m}{m_1} \pmod{N}$$

This ensures that $m_1 \cdot m_2 = m \pmod{N}$. She queries the signing oracle with m_1, m_2 and obtains the signatures s_1, s_2:

$$s_1 = m_1^d \pmod{N}$$
$$s_2 = m_2^d \pmod{N}$$

Using the property of modular arithmetic,, the signatures can be combined:

$$s = s_1 \cdot s_2 \pmod{N}$$
$$s = (m_1^d \cdot m_2^d) \pmod{N}$$
$$= (m_1 \cdot m_2)^d \pmod{N}$$

$$= m^d \pmod{N}$$

This gives the valid signature s for the target message m. The attack relies on the multiplicative structure of RSA. By introducing proper padding, the relation

$$m = m_1 \cdot m_2 \mod N$$

is broken, making it impossible to combine m_1 and m_2 into m whi

9.4.3 Signing Documents

Challenges of signing large documents include the need to divide the message m into smaller blocks if it is too large, adding serial numbers and redundant information to ensure integrity and uniqueness for each block, and the fact that signing and verifying each block individually can be computationally expensive.
Solution: use a hash function. Instead of signing the entire message m, the process is simplified by:

- Computing the hash of the message $h(m)$, a fixed-size digest.
- Signing the hash $h(m)$ instead of the full message.
- Separate message recovery and Verification

Sender's Side (A):

1. **Message Preparation:** The sender A computes the hash of the message: $h(M)$.
2. **Generate the Signature:** The sender uses their private key D_{SA} to sign the hash $h(M)$, creating the digital signature $S_{SA}(M)$.
3. **Transmit the Signed Message:** The sender sends M (the original message) and $S_{SA}(M)$ (the signature) to the receiver.

Receiver's Side (B):

1. **Hash Verification:** The receiver computes the hash of the received message M': $h(M')$.
2. **Verify the Signature:**
   - The receiver uses the sender's public key P_A to decrypt the signature $S_{SA}(M)$, obtaining $h(M)$.
   - Compare $h(M')$ (newly computed) with $h(M)$ (extracted from the signature).
   - If they match, the message M' is verified as authentic.
3. **Result:**
   - If $h(M') = h(M)$, verification succeeds.
   - Otherwise, verification fails.

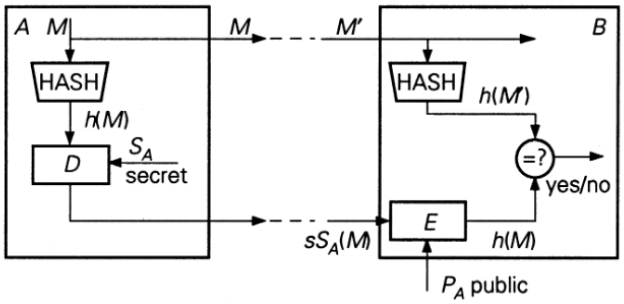


Figure 38: **Sender Side:** M is hashed ($h(M)$), and the private key generates the signature $S_{SA}(M)$. Both M and $S_{SA}(M)$ are sent to the receiver. **Receiver Side:** The receiver computes $h(M')$ and verifies the signature using P_A. If $h(M') = h(M)$, the message M' is validated.

9.4.4 Requirements for the Hash Function

- **Preimage resistance:** Adversary should not be able to cook her own signature for a message of her choice
- **Second preimage resistance:** Attacker can find another message for a valid signature
- **Collision resistance:** The signer generates two messages with $H(m) = H(m')$ and releases (m, s). Later, she claims (m', s) (repudiation)

more explanation here

9.5 More on the security of RSA

- Knowledge of d and factoring: if you know d, you can factor N using the Las Vegas algorithm
- Knowledge of $\phi(N)$ and factoring: if you know $\phi(N)$, you can factor N
- Shared modulus N is *not* a good idea!
- Use a small public exponent e: using CRT, RSA can be broken. Thus padding is important
- More attacks exist (see slides)

9.5.1 Knowledge of $\phi(N)$ and factoring

$$\phi(N) = (p-1)(q-1) = N - (p+1) + 1$$
$$S = N + 1 - \phi$$
$$S = p + q$$

$$f(X) = (X-p)\cdot(X-q) = X^2 - S\cdot X + N$$

Then:

$$p = \frac{S + \sqrt{S^2 - 4\cdot N}}{2}$$
$$q = \frac{S - \sqrt{S^2 - 4\cdot N}}{2}$$

9.5.2 Use of a Shared Modulus

There are two people sharing the same modulus. Two cases:

1. Attacker shares the modulus with another person. From d_1, the attacker can compute p and q. Then the attacker computes d_2 from e_2 (public key), p and q.
2. Attacker is not one of the two people

$$c_1 = m^{e_1} \pmod N, \quad t_1 \leftarrow e_1^{-1} \pmod{e_2}$$
$$c_2 = m^{e_2} \pmod N, \quad t_2 \leftarrow (t_1 \cdot e_1^{-1} - 1)/e_2$$

Then:

$$c_1^{t_1} \cdot c_2^{t_2} = m^{e_1\cdot t_1} m m^{-e_2\cdot t_2}$$
$$= m^{1+e_2\cdot t_2} m m^{-e_2\cdot t_2}$$
$$= m^{e_1\cdot t_1 - e_2\cdot t_2}$$
$$= m^1 = m$$

9.5.3 Use of a Small Public Exponent

For fast computation, choosing a small public exponent is common. However, this can lead to attack, particularly when the same message m is encrypted under different public keys N_1, N_2, N_3. This is because the ciphertexts (c_1, c_2, c_3) do not involve any randomness in the encryption process (as is the case in textbook RSA).

For example, if $e = 3$, then:

$$c_1 = m^3 \pmod{N_1}$$
$$c_2 = m^3 \pmod{N_2}$$
$$c_3 = m^3 \pmod{N_3}$$

Since the moduli are distinct (and therefore coprime), the CRT can be used to comhine the ciphertexts and reconstruct the original message m^3 modulo $N_1 \cdot N_2 \cdot N_3$. Because $m^3 < N_1 \cdot N_2 \cdot N_3$, the modular reductiion has no effect (no modulus wrapping). So then, $X = m^3$ and $X^{1/3} = m$. Thus, taking the cube root of this value yields the original message m.

$$X = c_i \pmod{N_i} \text{ for } i = 1, 2, 3$$
$$X = m^3 \pmod{N_1 \cdot N_2 \cdot N_3}$$

An attacker can now decrypt the message m without knowing the private keys.