Cryptography and Architectures for Computer Security

Niccoló Didoni

July 6, 2023

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Security models

Before introducing the different security models we have to state an important principle that has to be always respected when building a cryptosystem.

Principle 1.1 (Kerckhoff)**.** *A cryptosystem should be secure even if everything about the system is publicly known, except a single parameter, i.e., the cryptographic key.*

This means that the cypher can be studied and tested by anyone thus, lowering the risk of theoretical hidden weaknesses and/or practical design pitfalls. Moreover, the cypher primitives must be efficiently implemented in a wide range of hardware and software systems so to minimise user-induced errors and negligence of the operators.

When evaluating the security of a cryptosystem we can use the security margin, which is defined as follows:

Definition 1.1 (Security margin)**.** *The security margin of a cryptoscheme is the computational effort required to obtain a plaintext or a key.*

The security margin takes into consideration the mathematical strength and the information leakage due to implementation details. Depending on the security margin provided by a cryptosystem we can recognise four different security models:

- **Unconditional security** or perfect secrecy. An attacker with infinite computational power and resources can infer no information about the key or the original message, given an encrypted message.
- **Computational security**. An attacker is computationally limited (which is a more realistic assumption), hence he or she can infer no information assuming a lower bound to the complexity of the best methods available to break the cypher.
- **Provable security**. Proves that breaking a cryptosystem is as hard as solving a known computationally hard problem.

- **Applied security**. Quantifies the security margin of the cryptosystem implementation against active and passive hardware and software analysis. In this context:
  - An active attacker actively changes the hardware frequency or the voltage to induce an error in the hardware and obtain the secret key or the plaintext by comparing the error with the right behaviour.
  - A passive attacker measures the execution time, the power consumption and the electromagnetic emissions of the hardware to infer some information about the system. A passive attacker can only observe the system but can't actively interact with it.

1.2 Security services

A cryptography system should always offer four important properties:

- **Confidentiality**.
- **Authentication**.
- **Integrity**.
- **Non-repudiation**.

1.2.1 Confidentiality

Confidentiality ensures that a piece of information can be read only by those entitled to do so. This means that some data can be read by no one, unless by those who are entitled to by the data owner. Confidentiality is the building block on which the other properties are based, however, it's important to underline that confidentiality is different from privacy. Privacy is in fact the right to decide when, how, if, and to who any information related to some data should be disclosed. Privacy is therefore more generic since it allows us to decide when, how and where confidentiality is employed or needed.

1.2.2 Authentication

Authentication allows us to ensure and verify that a user is who he or she claims to be. This means that authentication allows to answer:

- Who the user is.
- If the user is really who he or she claims to be.

An example of authentication is logging in to a computer in fact a person is requested to specify the user name (used to identify the user) and a password (used to verify if the user is really who he or she claims). Authentication can be further divided into:

- **Entity authentication**. Entity authentication ensures that a person is who he or she claims to be (e.g., the username and password mechanism).
- **Data origin authentication**. Data origin authentication ensures that some data is exchanged between the correct endpoints and that no third party in the middle is interpreting the role of the sender or of the receiver. In other words, data origin authentication ensures that the sender and the receiver of some data are the legitimate ones.

Authorisation

Authorisation is different from authentication in fact the latter can be ensured only after having enforced the first. More precisely, authorisation ensures that an authorised user can only access the data he or she is entitled to. An example is file access in a computer, in fact after having provided username and password (i.e., after authentication) a user might be able to open and modify only some files. Authorisation can be achieved using three different models:

- **Discriminative access control models**. In discriminative access control models, the owner of a resource defines which agent can read or write the resource. This is the way authentication works in general-purpose operating systems.
- **Mandatory access control models**. In mandatory access control models, the system decides which agent can read or write each resource. More precisely, the system defines strictly ordered security levels (e.g., top-secret, secret, private, public) and assigns them to each resource (in this case they are called sensitivity) and agent (in which case are called clearance). The system then defines rules that specify what resources an agent can access, depending on the clearance and the sensitivity.
- **Role-based access control models**. In role-based access control models, users can be assigned to different roles and then each role is assigned clearance to access some resource.

1.2.3 Integrity

Integrity ensures that data exchanged in a communication is not manipulated or tampered with by a third party. When talking about manipulation we intend:

- **Insertion**.
- **Deletion**.
- **Replacement**.

Data authentication

Data integrity alone isn't very powerful since we also have to consider the source that generated the data. Data authentication takes this into consideration in fact it merges

- **data integrity** by verifying that the data hasn't been tampered with and
- **data origin authentication** by verifying that the sender is the legitimate origin of the data.

1.2.4 Non-repudiation

Non-repudiation ensures that an entity can't deny actions that it has done. For instance, if some agent generates some data, then it can't deny it and say that someone else did it.

1.3 Cryptography terminology

Let us now list some basic definitions related to cryptography.

1.3.1 Alphabet

First, let's start by defining what an alphabet is.

Definition 1.2 (Alphabet). *An alphabet $\mathcal{A}$ is a finite set of symbols.*

Among all alphabets, the binary alphabet is very important because it is very used when talking about cryptosystems for computers and because we can obtain any other alphabet starting from it.

Definition 1.3 (Binary alphabet). *The alphabet $\mathcal{B} = \{0, 1\}$ is called binary alphabet.*

1.3.2 Message space and plaintext

Definition 1.4 (Message space). *The message space $\mathcal{M}$ is the set of strings over an alphabet.*

For instance, if we consider the binary alphabet $\mathcal{B}$, then a message space over $\mathcal{B}$ could be the set of bit strings with at most 3 characters, namely

$$\{0, 1, 10, 11, 100, 101, 110, 111\}$$

Definition 1.5 (Plaintext). *A string in a message space $\mathcal{M}$ is called plaintext p.*

1.3.3 Ciphertext space and ciphertext

Similarly to the message space, we can define the ciphertext space as:

Definition 1.6 (Ciphertext space). *The ciphertext space $\mathcal{C}$ is the set of strings over an alphabet.*

Definition 1.7 (Ciphertext). *A string in a ciphertext space $\mathcal{C}$ is called ciphertext c.*

The alphabet on which the message space is defined can be different from the one used to define the ciphertext space.

1.3.4 Key space

Definition 1.8 (Key space). *A key space $\mathcal{K}$ is a set of elements called keys.*

The dimension of the key space is important when evaluating a cryptosystem since it defines the complexity of a brute-force attack. Nowadays a key size of 2^{80} is considered acceptable.

1.3.5 Transformations

Definition 1.9 (Encryption transformation)**.** *Let $\mathcal{K}$ a key space, $\mathcal{M}$ a message space and $\mathcal{C}$ a ciphertext space. An encryption transformation is a bijective function*

$$\mathbb{E}_e : \mathcal{M} \to \mathcal{C}$$

that given a key $e \in \mathcal{K}$ maps a message in $\mathcal{M}$ into a ciphertext in $\mathcal{C}$.

Definition 1.10 (Decryption transformation)**.** *Let $\mathcal{K}$ a key space, $\mathcal{M}$ a message space and $\mathcal{C}$ a ciphertext space. A decryption transformation is a bijective function*

$$\mathbb{D}_d : \mathcal{C} \to \mathcal{D}$$

that given a key $d \in \mathcal{K}$ maps a ciphertext in $\mathcal{M}$ into a plaintext in $\mathcal{M}$.

1.3.6 Cryptoscheme

Definition 1.11 (Cryptoscheme)**.** *A cryptoscheme is a tuple*

$$(\mathcal{A}, \mathcal{K}, \mathcal{M}, \mathcal{C}, \{\mathbb{E}_e : e \in \mathcal{K}\}, \{\mathbb{D}_d : d \in \mathcal{K}\})$$

where

- *$\mathcal{A}$ is an alphabet.*
- *$\mathcal{K}$ is a key space.*
- *$\mathcal{M}$ is a plaintext space.*
- *$\mathcal{C}$ is a ciphertext space.*
- *$\mathbb{E}$ is an encryption transformation.*
- *$\mathbb{D}$ is a decryption transformation.*

and the following properties are met:

- ***Correctness**. The correctness property ensures that each plaintext in $\mathcal{M}$ can be encrypted and decrypted to obtain the original plaintext. Formally we can write:*

$$\forall e \in \mathcal{K} \quad \exists d \in \mathcal{K} : \forall m \in \mathcal{M} \quad \mathbb{D}_d\big(\mathbb{E}_e(m)\big) = m$$

- ***Efficiency**. The efficiency property ensures that the encryption and decryption transformations are fast to compute with the right key and long to compute without the key. In other words, we can say that $\mathbb{E}$ and $\mathbb{D}$ have to be one-way functions (i.e., easy to compute in one way and hard in the other) with trapdoors (hard to compute without the correct key).*

1.4 Cryptography paradigms

Cryptoschemes can be divided into different categories depending on how they encrypt and decrypt data. We can recognise two different paradigms:

- **Symmetric encryption**.
- **Public key encryption**.

1.4.1 Symmetric encryption

Symmetric encryption uses the same key for encryption and decryption, which means that the encryption and decryption transformations $\mathbb{E}_k$ and $\mathbb{D}_k$ use the same key. When two users want to securely share some data, they have to agree on a common key which is used by the producer of the data to encrypt it and by the consumer to decrypt it. Symmetric encryption schemes can be divided into two categories:

- **Block encryption schemes** (or block cyphers). Block cyphers take a block of data of a fixed length and encrypt (or decrypt) it.
- **Stream encryption schemes** (or stream cyphers). Stream cyphers encrypt (and decrypt) continuous blocks of data of unbounded length.

The former is usually more popular than the latter which is used in very specific application like key-less car opening. In fact the former is also used to encrypt and decrypt stream of data which can be split in blocks of data and encrypted using block cyphers.

Pros and cons

The main advantages of symmetric encryption are:

- It's very **fast** and **efficient**.

The main disadvantages are:

- The **shared key has to be shared using a separate secure channel** since the communication channel on which the key is required is not secure.
- It **can't provide the non-reputation property** since data producer and consumer use the same key, hence we have no way to check if the data has been encrypted by the producer or by the consumer. Luckily the properties of confidentiality and authentication are met.
- It's **hard to handle big groups**, in fact, if each agent of the group wants to communicate with all the others, then each agent must have a unique key for each other agent. More precisely, in a group of n agents we need $\frac{n(n-1)}{2}$ keys, hence the number of keys grows quadratically with the number of agents.

1.4.2 Public key encryption

Public key encryption uses two keys:

- The public key p, which is publicly disclosed by a user.

- The private key s, which has to be securely stored and kept as a secret by the user.

What is encrypted with the private key can only be decrypted with the public key and what's encrypted with the public key can only be decrypted with the private key. Public key encryption schemes usually exploit hard mathematical problems like the **discrete logarithm problem** and the **integer factorisation problem**.

Advantages and disadvantages

The main advantages of public key encryption are:

- **Key management is more scalable** since every user has to generate its own couple of keys, hence we have a linear complexity.
- It **provides the non-reputation property** since a key is uniquely bound to a user, hence we can check if some data has been encrypted with a user's key.

The main disadvantages are:

- It's **much slower** than symmetric key encryption on any platform (usually even by a factor of 1000).
- **Longer keys are required** with respect to symmetric encryption to provide the same level of security.
- We have to **authenticate the user**, in fact, we have to associate a user with its public key.

1.5 Public key-user association

1.5.1 Digital signatures

A digital signature is a tool that allows verifying if some data m has been produced by a certain user and it hasn't been tampered with by a third party. Digital signatures are obtained using a signature transformation

$$\mathbb{S}_s : \mathcal{C} \to \mathcal{M}$$

that takes the data to sign and some secret s and generates a signature. Since the signature is obtained using a user's secret, then it must have been signed by him or her. The signature can be verified using a verification transformation

$$\mathbb{V}_p : \mathcal{M} \to \mathcal{C}$$

which takes a signature and extracts the data from which the signature has been generated. As an example, RSA uses the decryption and encryption functions as signature and verification transformations. In particular, the producer can apply $\mathbb{D}_s$ on the data m it wants to sign and those who want to verify the signature can apply the encryption function $\mathbb{E}_p$ on the signature and using the user's public key p. If the data obtained from decrypting the signature is the same as the original data then the signature is verified. This schema works because a message encrypted with a user's private key can only be decrypted with the user's public key, hence if we can successfully decrypt a signature with a user's public key p, then the signature must have been generated with that user's private key s and we are sure that the producer of the data is the one with private key p.

1.5.2 Digital certificates

Digital signatures are very useful when we want to authenticate a user, hence when we want to guarantee that a certain public key is bound to a user.

Public key infrastructure

To achieve this goal a trusted third party, called certification authority, can generate a file containing an identification of the user, its public key, some metadata and some other data about the user or the third party and then sign the document with its (i.e., the third party's) key. The signed document is the digital certificate of the user. This means that the third party guarantees that a private key is associated with a user and every other user can verify that the association is legitimate by checking the third-party signature. This only moves the problem in fact we have authenticated the user but not the third party. To solve this problem, which is recursive, we have to use a hierarchical structure where each level vouches for the level below. The structure ends with a few certification authorities that are globally trusted and vouch for themselves.

The hierarchical structure used to generate, handle and revoke certificates is called Public Key Infrastructure (PKI).

Web of trust

The web of trust proposes a distributed approach for handling public keys. In particular, different users provide guarantees for a user, hence we can be sure that a key is associated with a user if all the guarantees agree on the key-user association.

1.5.3 Identity based authentication

Identity-based authentication systems use the user's identity to derive the public key, hence no authentication is required since the key is associated with a user by definition. Note that, in general, we can't use the identity itself as a key but we have to use a function that maps the identity into an appropriate key space. The private key is instead provided by a third party called **Trusted Authority** (TA) which combines the user's identity and a master secret. Identity-based authentication is mainly based on elliptic curves.

Advantages and disadvantages

The main advantages of identity-based authentication are:

- **No digital certificates** are required.
- **Multiple private keys can be bound to the same user identity**. A key can be bound to a specific time-lapse (e.g., only in the future). Moreover, the identity of a user can be split into multiple keys, one for each of his roles to partition information access rights.

The main disadvantages are:

- A user's private key is known to the Trusted Authority. This problem is known as **key escrow** and it's a problem both because we have more points where the private key could be stolen and because the user's secret is in the hands of a third party which is therefore able to read the messages sent to a user.

- Systems for handling identity based authorisation are **more complex** and provide **worst performance**.

1.6 Attacks

An important part of assessing the security of a cryptosystem is defining the appropriate thread model hence it's important to underline the different attacks model we can face. An attacker can be classified as:

- **Passive attacker**. A passive attacker only monitors the channel on which data is exchanged. This means that a passive attacker can only break the confidentiality of a message since he or she can't tamper with the encrypted message (breaking its integrity) nor interact to authenticate as another user (breaking the authentication property).
- **Active attacker**. An active attacker is able to interact with the messages, altering, deleting or creating them and with the user, hence he or she can break all the properties of a cryptosystem.

Now that we have defined all the possible attackers we can analyse what information an attacker can use to recover a message or a key. Depending on the information available to an attacker we can define different attack models:

- **Bruteforce attacks**.
- **Ciphertext only attacks** (COAs).
- **Known plaintext attacks** (KPAs).
- **Chosen plaintext attacks** (CPAs).
- **Chosen ciphertext attacks** (CCAs).

All the attack models are based on the fundamental assumption that *the cryptoscheme used to encrypt and decrypt a communication is known in all its components, including its implementation and the platform on which it's run.* This assumption is formally stated by Kerckhoff's principle:

> **Principle 1.2** (Kerckhoff)**.** *Let* $(\mathcal{A}, \mathcal{K}, \mathcal{M}, \mathcal{C}, \{\mathbb{E}_e : e \in \mathcal{K}\}, \{\mathbb{D}_d : d \in \mathcal{K}\})$ *be a cryptosystem. The alphabet* $\mathcal{A}$*, the structure of the plaintexts (i.e., the form of* $\mathcal{M}$*) and the details of the encryption and decryption algorithms are known.*

1.6.1 Bruteforce attacks

A bruteforce attack is the easiest to implement since it only requires, given a ciphertext c, to try and decrypt c with all the possible keys in the key space. For this reason, bruteforce attacks are also called exhaustive key search attacks. Note that, given a message m and a ciphertext $c = \mathbb{E}_e(m)$ (obtained encrypting m), an attacker has to be able to understand if the plaintext obtained decrypting c is the actual message m or not. This means that an attacker must be able to understand if a decrypted message corresponds to the original plaintext. This happens because, in some cryptoschemes, we can decrypt a message a with different keys and still obtain valid messages (i.e., belonging to $\mathcal{M}$). This is at the core of perfect security cyphers since they require that neither the plaintext nor the key can be obtained from a ciphertext because an attacker is not able to recognise the correct original message among all valid decrypted messages.

1.6.2 Ciphertext only attacks

In ciphertext-only attacks (COAs), the attacker only knows the ciphertexts obtained by encrypting some plaintexts. More precisely, the attacker has some ciphertexts obtained with the same unknown key and has to recover the plaintexts or the key used for encryption (or both). In this scenario, the attacker can compare the ciphertexts to find regularities and patterns that can help in recovering the key or the plaintexts.

1.6.3 Known plaintext attacks

In known plaintext attacks (KPAs), the attacker knows the plaintexts and the corresponding ciphertexts, all obtained with the same unknown key and the goal of the attacker is to recover the key. In this scenario the attacker has more information hence it should be easier to find the key with respect to COAs.

1.6.4 Chosen plaintext attacks

In chosen plaintext attacks (CPAs), the attacker can query an encryption oracle. This means that the attacker can choose some plaintexts and ask the encryption algorithm to encrypt them, all with the same key. Note that this is different from known plaintext attacks since in that model the attacker couldn't choose the plaintext and the corresponding ciphertext.

When considering a chosen plaintext attack a cryptoscheme can meet two properties:

- One-wayness under chosen plaintext attacks.
- Indistinguishability under chosen plaintext attacks.

One-wayness under chosen plaintext attacks

A cryptoscheme meets the property of one-wayness under chosen plaintext attacks if an attacker has a negligible probability of guessing the plaintext m chosen and encrypted by the defender. Basically,

1. The defender picks a plaintext m.
2. The attacker must guess or recover m. If the attacker has a negligible probability of guessing m correctly, then the scheme is secure against OW-CPAs.

Indistinguishability under chosen plaintext attacks

A cryptoscheme meets the property of indistinguishability under chosen plaintext attacks if an attacker can't understand if a ciphertext has been generated from a message m_0 or from a different message m_1. More precisely:

1. The attacker picks two plaintexts m_0 and m_1.
2. The defender chooses a message at random among m_0 and m_1, encrypts it and sends it to the attacker.
3. The attacker has to guess if the receiver ciphertext has been generated from m_0 or from m_1. If the attacker succeeds with a probability smaller than 0.5 then the scheme meets the property of indistinguishability under chosen plaintext attacks.

Comparison between the two properties

Indistinguishability is easier to break from an attacker's point of view with respect to one-wayness, hence the former property is more powerful than the latter since if a scheme meets IND-CPA, then it's also OW-CPA. In other words, IND-CAP implies OW-CPA.

1.6.5 Chosen ciphertext attacks

In chosen ciphertext attacks (CCAs) the attacker can query an encryption and a decryption oracle. This means that the attacker can

- encrypt some plaintexts with the same key and
- decrypt some ciphertext with the same key.

Note that the key used by the oracle for encryption and decryption is the same. CCA models well an attacker able to interact with a device containing a private key.

Indistinguishability under chosen-ciphertext attacks

The process for verifying if a cryptoscheme meets the property of indistinguishability under chosen-ciphertext attacks is the same as the one seen for chosen plaintext attacks, however, in this case, the attacker can also use an oracle to decrypt some messages (different from the one generated by the defender). More precisely:

1. The attacker picks two plaintexts m_0 and m_1.
2. The defender chooses a message at random among m_0 and m_1, encrypts it and sends it to the attacker.
3. The attacker has to guess if the receiver ciphertext has been generated from m_0 or from m_1. If the attacker succeeds with a probability smaller than 0.5 then the scheme meets the property of indistinguishability under chosen ciphertext attacks.

If the attacker can perform some decryption even after having sent its messages then the model is called **adaptive chosen ciphertext attack** (CCA2).

1.6.6 Attacks models in the real world

Ciphertext-only attacks and known plaintext attacks are usually too weak to model a real-world attack since they don't consider the possibility that the attacker can use the same algorithm used by the agents that exchange messages. On the other hand, chosen ciphertext attacks and chosen plaintext attacks are more real since an attacker has usually access to the encryption and decryption device. Moreover, for real-world applications, all cyphers should be secure against chosen ciphertext attacks and in particular should be proven against the indistinguishability over chosen ciphertext attacks.

Chapter 2

History of cyphers

Since the introduction of public key cryptography is rather modern, all old cyphers are based on symmetric key encryption. In particular, we can split them into

- **Substitution cyphers**, which in turn can be divided into
  - Monoalphabetic substitution cyphers.
  - Polyalphabetic substitution cyphers.
- **Permutation cyphers**.

2.1 Substitution cyphers

2.1.1 Shift cyphers

A shift cypher is a specific case of a substitution cypher.

Definition 2.1 (Shift cypher)**.** *A shift cypher is a tuple*

$$(\mathcal{A}, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}$ *is a finite set of symbols, each of which is associated with a number starting from 0 to* $|\mathcal{A}|$.
- $\mathcal{M} = \mathcal{C}$ *are the sets of messages made of a single symbol of the alphabet* $\mathcal{A}$.
- $\mathcal{K} = \{0, 1, 2, \ldots, |\mathcal{A}| - 1\}$ *is the set of keys containing all the values from 0 to* $|\mathcal{A}| - 1$.
- $\mathbb{E}_k$ *is the encryption function defined as:*

$$p \mapsto p + k \mod |\mathcal{K}|$$

- $\mathbb{D}_k$ *is the decryption function defined as:*

$$c \mapsto c - k \mod |\mathcal{K}|$$

Let us consider the alphabet

$$\mathcal{A} = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

containing the letters of the English alphabet as an example. Each letter is associated with a number, namely $A \mapsto 0, \ldots, Z \mapsto 25$. In this situation the key space is

$$\mathcal{K} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25\}$$

whilst the encryption and decryption functions are

$$\mathbb{E}_k(p) = p + k \mod 26$$

and

$$\mathbb{D}_k(c) = c - k \mod 26$$

respectively. Basically, the shift cypher encrypts a message by moving through the letters of the alphabet (ordered using the numbers assigned to each letter) of the number of positions specified by the key. The modulo operation is required to go back to the first letter of the alphabet when the last letter is reached. Depending on the value of k we can obtain famous cyphers, in particular:

- For $k = 3$ we obtain the **Caesar's cypher**.
- For $k = 13$ we obtain the **rot13 cypher** for which the encryption function coincides with the decryption function since the key is half the length of the alphabet hence adding 13 is like subtracting 13.

Cryptanalysis

The security margin of a shift cypher is very low because an attacker can try all the possible keys with a bruteforce attack. Considering the English alphabet, the key space has a cardinality of 26, hence an attacker has to try 26 different keys, which, even by hand, takes a little time.

2.1.2 Monoalphabetic substitution cyphers

Monoalphabetic substitution cyphers use different alphabets for the plaintext and ciphertext spaces and the encryption process replaces each letter of the plaintext alphabet with a letter of the ciphertext alphabet according to a bijective map. More formally,

Definition 2.2 (Monoalphabetic substitution cypher)**.** *A monoalphabetic substitution cypher is a tuple*

$$(\mathcal{A}_m, \mathcal{A}_c, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}_m$ *is the alphabet used for the plaintext space.*
- $\mathcal{A}_c$ *is the alphabet used for the ciphertext space.*
- $\mathcal{M}$ *is the plaintext space containing the strings made of a single symbol taken from* $\mathcal{A}_m$*.*
- $\mathcal{C}$ *is the ciphertext space containing the strings made of a single symbol taken from* $\mathcal{A}_c$*.*

- *$\mathcal{K}$ is the key space made that contains all the possible bijective maps from $\mathcal{A}_m$ to $\mathcal{A}_c$.*
- *$\mathbb{E}_k : \mathcal{M} \to \mathcal{C}$ is the encryption function that uses the bijective map defined by the key to map a plaintext into the corresponding ciphertext.*
- *$\mathbb{D}_k : \mathcal{C} \to \mathcal{M}$ is the decryption function that uses the inverse of the bijective map defined by the key to map a ciphertext into the corresponding plaintext. Note that the inverse of the map exists since it's bijective.*

Note that the plaintext alphabet and the ciphertext alphabet must have the same cardinality to have a bijective map.

Since the main feature of a substitution cypher is its map, it's useful to introduce a way to represent it in a compact way. First, we have to assume that we can assign to each element of the alphabets a unique index ranging from 1 to $|\mathcal{A}|$. Thanks to this notation we can represent a map, hence a key, as a list of numbers with values from 1 to $|\mathcal{A}|$. The number in position i of the list defines the position of the letter in the ciphertext alphabet to which the i-th letter of the plaintext alphabet has to be mapped. Better said, the letter in position i of the plaintext alphabet is mapped to the letter in position $k[i]$ of the ciphertext alphabet. Say for instance that we have the following key:

$$k = (12, 11, 24, 5, 13, 10, 14, 8, 15, 4, 7, 9, 22, 20, 19, 16, 21, 26, 6, 23, 25, 1, 3, 2, 17, 18)$$

Then the letter in position 2 of the plaintext alphabet (starting from 1) is mapped to the 11th letter of the ciphertext alphabet.

Cryptanalysis

A substitution cypher is more resistant against bruteforce attacks since the key space is way larger, in fact, the cardinality of the key space is

$$|\mathcal{K}| = |\mathcal{A}_m|!$$

This observation comes from the fact that the first letter of $\mathcal{A}_m$ can be mapped to any of the $|\mathcal{A}_c|$ letters in $\mathcal{A}_c$. The second letter can be mapped to all the letters in $\mathcal{A}_c$ except the one used for the first letter. If we repeat this process on and on we obtain

$$|\mathcal{K}| = |\mathcal{A}_m| \cdot (|\mathcal{A}_m| - 1) \cdot (|\mathcal{A}_m| - 2) \cdot \ldots 1 = |\mathcal{A}_m|!$$

Although bruteforce attacks aren't feasible by hand (with $|\mathcal{A}| = 26$ we have $26! \approx 2^{88}$ keys) an attacker can still break this cypher, even in a ciphertext-only scenario. More precisely, an attacker who knows the language in which messages are written can use statistics to compute the frequency with which a symbol is present in a generic message and use this information to obtain the map. This happens because a monoalphabetic substitution cypher creates a **1-to-1 association** between letters in $\mathcal{A}_m$ and in $\mathcal{A}_c$, namely each letter in $\mathcal{A}_m$ always mapped to the same letter in $\mathcal{A}_c$. Say for instance that the messages are written in English. By analysing English texts we can compute the frequency with which each letter appears in a generic text. When we see a ciphertext, we can compute the same statistics for the symbols in the ciphertext and infer that if a symbol $\alpha \in \mathcal{A}_m$ has the same frequency as a symbol $\xi \in \mathcal{A}_c$, then the map probably associates ξ to α. If we repeat the same process for every letter in the plaintext alphabet we can obtain the full map. This attack has approximately a linear complexity since we only have to pass the ciphertext once to compute the symbols' statistics. Note that since the ciphertext is way smaller than the corpus of English

phrases the statistics obtained on the ciphertext are different from the ones computed on the English phrases, hence one might have to try different associations. When mounting frequency attacks one might also have to consider the most common bigrams (e.g., is, do, an) and trigrams (e.g., the, and) to help recognise common patterns.

2.1.3 Polyalphabetic substitution cyphers

Polyalphabetic substitution cyphers try to solve the weaknesses of monoalphabetic substitution cyphers by removing the 1-to-1 mapping between the two alphabets. More precisely, a polyalphabetic substitution cypher is defined as follows:

Definition 2.3 (Polyalphabetic substitution cypher)**.** *A polyalphabetic substitution cypher is a tuple*

$$(\mathcal{A}_m, \mathcal{A}_c, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}_m$ *is the alphabet used for the plaintext space.*
- $\mathcal{A}_c$ *is the alphabet used for the ciphertext space.*
- $\mathcal{M}$ *is the plaintext space containing the strings (called words) made of a finite sequence of symbols taken from* $\mathcal{A}_m$*.*
- $\mathcal{C}$ *is the ciphertext space containing the strings (called words) made of a finite sequence of symbols taken from* $\mathcal{A}_c$*.*
- $\mathcal{K}$ *is the key space which is a set of tuples, each containing* $L > 1$ *bijective maps, namely we have*

  $$\mathcal{K} = \{(\mu_0, \mu_1, \dots, \mu_{L-1})_i\}$$

  where μ_i *is a bijective map.*
- $\mathbb{E}_k$ *is the encryption function that maps the* i*-th letter of the plaintext using the map* $\mu_{i \mod L} \in k$
- $\mathbb{D}_k$ *is the decryption function that applies the inverse of the map* $\mu_{i \mod L} \in k$ *to the* i*-th letter of the ciphertext.*

A polyalphabetic substitution cypher is able to break the 1-to-1 association between the two alphabets because a letter is mapped to at most L different letters, depending on the map that is used. If we consider for instance $L = 3$ and the string "AAA", then the letter A could be mapped to three different letters since we use μ_0 for the first letter, μ_1 for the second and μ_2 for the third.

Cryptanalysis

A direct bruteforce approach is even less feasible with respect to monoalphabetic substitution cyphers since we don't only have to consider all possible maps but also all the possible orders in which the L maps are applied. This means that the number of keys is

$$|\mathcal{K}| = (|\mathcal{A}|_m!)^L$$

which was already big for $L = 1$ (i.e., for monoalphabetic substitution cyphers). Polyalphabetic substitution cyphers are a bit stronger than their monoalphabetic counterpart, however, they can still be attacked using statistics. In particular, we can apply the same method we have seen for monoalphabetic substitution cyphers on different chunks of the ciphertext. If we assume that the length L is known (we'll see how to obtain it later on), then we can split the ciphertext in L chunks where chunk γ contains the letters in the positions $\gamma + iL \forall i = 0, \dots$. Basically, the first chunk contains the letters in positions $0, L, 2L, 3L, \dots$, the second chunk the letters in positions $1, 1 + L, 1 + 2L, \dots$ and so on. Each chunk contains the letters encrypted with the same map, hence a letter in a chunk is always mapped to the same symbol in the ciphertext alphabet. This means that we can apply frequency statistics on each chunk and obtain the same result we got for monoalphabetic substitution cyphers. In this case, we might have to try more associations between letters in $\mathcal{A}_m$ and $\mathcal{A}_c$ since, having split the ciphertext in chunks, we have worse statistics for each block. This means that the larger the value of L, the harder it is to apply the frequency attack.

Kasisky test When doing the cryptanalysis of the polyalphabetic substitution cypher we have assumed that the number of maps L is known. Let us now explain how to obtain this value using the Kasisky test. This method is based on the observation that it's highly likely that some common bigrams are mapped to the same bigrams in the ciphertext. This is because if the bigrams are very common, then it's highly probable that they will appear in the same positions (considering the modulo), hence the same maps are used and both bigrams are mapped to the same bigram. Consider for instance the phrase *he is who he is* and $L = 5$. In this scenario the bigram *he* is mapped using the first two functions in both cases, hence *he* is always mapped to the same bigram.

In practice, we can compute the distances between all bigrams (or trigrams) that we find in the ciphertext and then compute L as the greatest common divisor of the distances between the distances of the bigrams.

Vigenère cypher

One famous example of a polyalphabetic substitution cypher is the Vigenère cypher which uses L cyclic shifts of the alphabet $\mathcal{A}$ (used for plaintext and ciphertext) as maps. Formally, the Vigenère cypher is defined as follows:

Definition 2.4 (Vigenère cypher)**.** *The Vigenère cypher is a tuple*

$$(\mathcal{A}, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}$ *is a finite set of symbols, each of which is associated with a number starting from 0 to* $|\mathcal{A}|$.
- $\mathcal{M} = \mathcal{C}$ *are the sets of messages made of a sequence of symbols of the alphabet* $\mathcal{A}$.
- $\mathcal{K}$ *is the set of keys, each being a sequence of L letters (called keyword) each of which denotes the first letter of the cyclic shift. Formally, the key space can be written as*

$$\mathcal{K} = \{(k_0, k_1, \dots, k_{L-1})_i : k_j \in \mathcal{A} \forall j = 0, \dots, L-1, \forall i \in |\mathcal{K}|\}$$

- $\mathbb{E}_k$ *is the encryption function that encrypts the* i*-th letter of the plaintext using the* $i \mod L$ *shift function defined by* k*. Formally, we have*

$$p_i \mapsto (p_i + k_{i \mod L}) \mod |\mathcal{A}|$$

- $\mathbb{D}_k$ *is the decryption function that decrypts the* i*-th letter of the ciphertext using the inverse of the* $i \mod L$ *shift function defined by* k*. Formally, we have*

$$c_i \mapsto (c_i - k_{i \mod L}) \mod |\mathcal{A}|$$

Beale cypher

The Beale cypher, also called book cypher, is a direct evolution of the Vigenère cypher obtained by using a longer key. In particular, the keyword is the first words of a book, namely the key space is the set of opening sentences of books. This is done because the longer the key, the lower the probability of having repeated bigrams, hence the hardest it is to attack the cypher. Note that a longer key also enlarges the key space, which is good in terms of bruteforce security margin but, as a downside, requires more computing power for encryption.

Even if Beale cypher might be more robust with respect to ciphertext-only attacks, it still is very weak against known or chosen plaintext attacks. This is because an attacker can query the oracle and obtain the encryption of each letter for all maps. For instance, if $L = 3$ and the attacker wants to obtain the maps he or she can simply send the string

`AAABBBCCCDDDEEEFFFGGGHHHIIIJJJKKKLLLMMMNNNOOOPPPQQQRRRSSSTTTUUUVVVWWWXXXYYYZZZ`

and obtain the mapping for each letter and for each map.

2.2 Permutation cyphers

Permutation cyphers allow us to scramble the letters of the ciphertext using a function that maps each position in the input with a position in the output. Formally, a permutation cypher is defined as follows:

Definition 2.5 (Permutation cypher)**.** *A permutation cypher is a tuple*

$$(\mathcal{A}, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}$ *is a finite set of symbols.*
- $\mathcal{M}$ *and* $\mathcal{C}$ *are the plaintext and ciphertext spaces containing strings made of a sequence of symbols of* $\mathcal{A}$*.*
- $\mathcal{K}$ *is the key space that contains all the possible permutations of* L *positions. A key is a tuple containing a permutation* π *and a length* L*.*
- $\mathbb{E}_k$ *is the encryption function that applies permutation* k *to the plaintext. If the plaintext is longer than* L *then it's split into chunks of length* L*, each chunk is encrypted separately and the encrypted chunks are then rejoined.*

- $\mathbb{D}_k$ *is the decryption function that applies the inverse of the permutation* k*, eventually splitting the ciphertext into blocks of length* L*.*

Let us now introduce a notation to represent keys, i.e., permutations. A permutation π of length L can be represented as a $2 \times L$ matrix where:

- The first row contains the indices (i.e., the positions) of the letters in the ciphertext.
- The second row contains the indices (i.e., the positions) from where we have to take a letter from the plaintext. Namely, in position i we find the index of the plaintext from which we take the letter to put in position i of the ciphertext.

This means that, if we have, as an example, the following permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 4 & 2 & 5 \end{pmatrix}$$

then

- the first letter of the ciphertext is the third letter of the plaintext,
- the second letter of the ciphertext is the first letter of the plaintext,
- the third letter of the ciphertext is the fourth letter of the plaintext,
- the fourth letter of the plaintext is the second letter of the plaintext, and
- the fifth letter of the ciphertext is the fifth letter of the plaintext.

This means that we can write

$$\pi(1) = 3 \qquad \pi(2) = 1 \qquad \pi(3) = 4 \qquad \pi(4) = 2 \qquad \pi(5) = 5$$

Given a plaintext $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$, we can compute the ciphertext as

$$\mathbf{c} = (x_{\pi(1)}, x_{\pi(2)}, x_{\pi(3)}, x_{\pi(4)}, x_{\pi(5)}) = (x_3, x_1, x_4, x_2, x_5)$$

Alternatively, we can use the following notation

$$\pi = (1342)(5)$$

to indicate that:

- The letter in position 1 of the ciphertext is taken from position 3 of the plaintext.
- The letter in position 3 of the ciphertext is taken from position 4 of the plaintext.
- The letter in position 4 of the ciphertext is taken from position 2 of the plaintext.
- The letter in position 2 of the ciphertext is taken from position 1 of the plaintext.
- The letter in position 5 of the ciphertext is taken from position 5 of the plaintext.

Note that a permutation isn't the same thing as a substitution in fact in the latter case we use all the letters of the ciphertext alphabet whereas in the former case, we use only the letters already in the plaintext and we simply change their position.

2.2.1 Cryptanalysis

Bruteforce attacks are infeasible if the value of L is large enough, in fact, the key space has a cardinality of

$$|\mathcal{K}| = L!$$

Moreover, a ciphertext-only analysis isn't very effective. In fact, we can still compute the frequency of each letter but we don't know how to map the letter to its original position. On the other hand chosen ciphertext attacks can easily break this cypher in fact an attacker can send L different letters to the oracle and check in which position each letter is sent to obtain the permutation function.

2.2.2 Advantages and disadvantages

The main advantages of a permutation cypher are:

- The key space can be quite large, in fact, there exist $L!$ possible permutations of length L. This helps when considering bruteforce attacks.
- It alters the frequency distribution of d-grams between plaintext and ciphertext.
- A ciphertext-only analysis isn't very effective.

On the other hand, the main disadvantages of a permutation cypher are:

- It doesn't alter the frequency distribution of the single letters.
- It isn't secure against known or chosen plaintext attacks.

2.3 Hill cypher

The Hill cypher is a polyalphabetic cypher. Formally, the cypher is defined as follows:

Definition 2.6 (Hill cypher)**.** *The Hill cypher is a tuple*

$$(\mathcal{A}, \mathcal{M}, \mathcal{C}, \mathcal{K}, \mathbb{E}_k, \mathbb{D}_k)$$

where

- $\mathcal{A}$ *is a finite set of symbols, each of which is associated with a number starting from 0 to* $|\mathcal{A}|$.
- $\mathcal{M} = \mathcal{C}$ *are the sets of messages made of a sequence of symbols of the alphabet* $\mathcal{A}$*. ciphertexts and plaintexts are divided into column vectors of size* m*.*
- $\mathcal{K}$ *is the set of keys. Each key is a* $m \times m$ *invertible matrix where each cell contains a value between 0 and* $|\mathcal{A}| - 1$*. The operations on the matrix have to be modulo* $|\mathcal{A}|$*.*
- $\mathbb{E}_k$ *is the encryption function defined as the matrix multiplication between the key* k *and the column vectors in which the plaintext is divided. Namely, the plaintext is divided into column vectors* p_i *of size* m *encrypted as:*

$$p_i \mapsto k \cdot p_i \mod |\mathcal{A}|$$

The resulting column vectors are then appended one to the other to obtain the full ciphertext.

- $\mathbb{D}_k$ *is the decryption function defined as the matrix multiplication between the inverted key and the column vectors in which the ciphertext is divided. Namely, the ciphertext is divided into column vectors* c_i *of size* m *and decrypted as:*

$$c_i \mapsto k^{-1} \cdot c_i \mod |\mathcal{A}|$$

The resulting column vectors are appended one to the other to obtain the full plaintext.

The Hill cypher potentially maps each symbol of the plaintext to a different symbol because the matrix multiplication doesn't simply replace values or reorder them but combines them.

2.3.1 Cryptanalysis

The Hill cypher is secure against bruteforce attacks since the key space is quite large, in fact, we have

$$|\mathcal{K}| \approx 26^{m^2}$$

matrices. Note that the cardinality of the key space isn't exactly 26^{m^2} because not all $m \times m$ matrices are invertible. Moreover, ciphertext-only attacks are impossible to achieve because the key mixes values of the plaintext and no frequency analysis is possible.

On the other hand, we can always mount a known or chosen plaintext attack since only have to solve a system of linear equations to obtain the values of the key. More precisely, given a pair (P, C) of plaintext and ciphertext, the key can be computed as

$$K = C \cdot P^{-1}$$

because

$$C = K \cdot P$$

2.4 Common characteristics and weaknesses

From the cryptanalysis of permutation and substitution cyphers we have obtained some common problems among all schemes:

- The cypher key should be long enough to withstand bruteforce attacks. Usually, a key should be at least 80-bit long for currently off-the-shelf computing machines.
- The mapping between plaintext and ciphertext letters, in the definition of the encryption and decryption transformation, should not be the same for every occurrence of the same plaintext letter (i.e., it should be position dependent). Namely, we shouldn't use functions that define a 1-to-1 mapping. A combination of substitution and permutation may be of help to avoid frequency-based COAs.
- A linear mapping (or any other efficiently invertible relation) between plaintext and ciphertext is threatened by KPAs.

- Frequency attacks exploit the redundancy of the English language lossless compression before encryption removes redundancy. Employing an uncommon (or dead) natural language may also help. Compressing the message before encrypting them can also be useful.

Chapter 3

Information theoretic security

3.1 Perfect cyphers

The goal of a cypher is to hide the content of a message from all the users except the intended receiver of the message. It's therefore interesting to understand if there exists a cypher such that when a third party sees a message, he or she can obtain no information from it. In practice, a message in a communication channel should look like a random sequence of symbols to everyone but the parts that are communicating. In other words, a perfect cypher should look unbreakable regardless of the effort put to break it. Shannon proved the existence of such a cypher, namely of a cypher resistant to ciphertext-only attacks, known plaintext attacks, chosen plaintext attacks and chosen ciphertext attacks (even adaptive chosen ciphertext attacks).

3.1.1 Statistical modelling

Let us consider the generic symmetric cypher

$$(\mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \{\mathbb{E}_k : k \in \mathcal{K}\}, \{\mathbb{D}_k : k \in \mathcal{K}\}) \tag{3.1}$$

Each item in $\mathcal{M}$, $\mathcal{K}$ and $\mathcal{C}$ can be described as a random variable P, K and C, respectively, with the probability distributions

- $Pr(\mathsf{P} = m)$ with $m \in \mathcal{M}$
- $Pr(\mathsf{K} = k)$ with $k \in \mathcal{K}$
- $Pr(\mathsf{C} = c)$ with $c \in \mathcal{C}$

We can see P, K and C as sets of objects, each associated with the probability of being picked. Moreover, we have to assume that P and K are statistically independent, hence that the key is chosen independently from the text that has to be encrypted. Thanks to these assumptions we can write the probability to observe a certain ciphertext c as

$$Pr(\mathsf{C} = c) = \sum_{k:c \in \{\mathbb{E}_k(m) \forall m \in \mathcal{M}\}} Pr(\mathsf{K} = k) \cdot Pr(\mathsf{P} = \mathbb{D}_k(c)) \tag{3.2}$$

where

- $k : c \in \{\mathbb{E}_k(m) \forall m \in \mathcal{M}\}$ is the set of keys that generate the ciphertext c when encrypting an arbitrary message m.
- $Pr(\mathsf{K} = k) \cdot Pr(\mathsf{P} = \mathbb{D}_k(c))$ is the product of the probability that key k is picked and that the plaintext obtained decrypting c with key k is obtained. In practice, the second term of this product is the probability of getting a meaningful text when decrypting c (i.e., a string in $\mathcal{M}$).

Another interesting thing we can compute is the probability of a ciphertext c knowing that we have observed a plaintext m. This probability can be computed as

$$Pr(\mathsf{C} = c|\mathsf{P} = m) = \sum_{k:m=\mathbb{D}_k(c)} Pr(\mathsf{K} = k) \tag{3.3}$$

We are saying that the probability of obtaining a certain ciphertext c having observed a plaintext m is the sum of the probabilities of picking the keys for which $\mathbb{D}_k(c) = m$. This is because c can be obtained from one of the keys that map m into c (i.e., for which $\mathbb{D}_k(c) = m$, hence $\mathbb{E}_k(m) = c$ because $m = \mathbb{D}_k(\mathbb{E}_k(m))$), hence the probability is the probability of picking one of the keys for which $\mathbb{D}_k(c) = m$ and therefore we consider the sum of probabilities. Now that we have the probabilities $Pr(\mathsf{C} = c)$ and $Pr(\mathsf{C} = c|\mathsf{P} = m)$ we can compute the probability of a message m knowing that we have observed a ciphertext c. This is the probability we are interested in since it is the probability of guessing the plaintext from the ciphertext. Thanks to Bayes' rule we can write

$$Pr(\mathsf{P} = m|\mathsf{C} = c) = \frac{Pr(\mathsf{P} = m) \cdot Pr(\mathsf{C} = c|\mathsf{P} = m)}{Pr(\mathsf{C} = c)} \tag{3.4}$$

3.1.2 Perfect cypher

Now that we have the tools, we can define a perfect cypher.

Definition 3.1 (Perfect cypher)**.** *A symmetric-key cryptosystem is perfectly secure if the ciphertext does not reveal any information about the plaintext, namely if*

$$Pr(\mathsf{P} = m|\mathsf{C} = c) = Pr(\mathsf{P} = m) \quad \forall m \in \mathcal{M} \ \forall c \in \mathcal{C} \tag{3.5}$$

The definition makes it clear that a cypher is perfect when the probability of obtaining a plaintext from a ciphertext doesn't depend on the ciphertext, namely the ciphertext holds no information. An important lemma that derives from Definition 3.1 of perfect cypher is the following:

Theorem 3.1. *A symmetric-key cryptosystem is perfectly secure if the plaintext does not reveal any information about the ciphertext, namely if*

$$Pr(\mathsf{C} = c|\mathsf{P} = m) = Pr(\mathsf{C} = c) \quad \forall m \in \mathcal{M} \ \forall c \in \mathcal{C} \tag{3.6}$$

Proof. Replacing Equation 3.4 into the Definition 3.1 of perfect cypher we get:

$$\frac{Pr(\mathsf{P} = m) \cdot Pr(\mathsf{C} = c|\mathsf{P} = m)}{Pr(\mathsf{C} = c)} = Pr(\mathsf{P} = m) \quad \forall m \in \mathcal{M} \ \forall c \in \mathcal{C} \tag{3.7}$$

$$\frac{Pr(\mathsf{C} = c|\mathsf{P} = m)}{Pr(\mathsf{C} = c)} = 1 \quad \forall m \in \mathcal{M} \ \forall c \in \mathcal{C} \tag{3.8}$$

$$Pr(\mathsf{C} = c|\mathsf{P} = m) = Pr(\mathsf{C} = c) \quad \forall m \in \mathcal{M} \ \forall c \in \mathcal{C} \tag{3.9}$$

□

Another important lemma related to perfect cyphers defines the relations between the cardinalities of the sets $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{K}$ in a perfect cypher.

Theorem 3.2. *Given a perfectly secure symmetric key cryptosystem, the following conditions hold*

$$|\mathcal{M}| \leq |\mathcal{C}| \leq |\mathcal{K}| \tag{3.10}$$

Proof. Let's start with the leftmost inequality, namely $|\mathcal{M}| \leq |\mathcal{C}|$. Let's suppose, by absurd, that the number of strings in the ciphertext space $\mathcal{C}$ is strictly smaller than the number of strings in the plaintext space $\mathcal{M}$, namely that

$$|\mathcal{C}| < |\mathcal{M}|$$

This means that two plaintexts map to the same ciphertext, as shown in Figure 3.1. This is a problem because it forces an imbalance in the probability distribution $Pr(P = m|C = c)$ because different cs might have a different number of inverse images. This means that the number of items in $\mathcal{C}$ has to be at least equal to the number of messages. Put it in another way, the encryption function has to be injective. The encryption function function has to be injective since the encryption function either

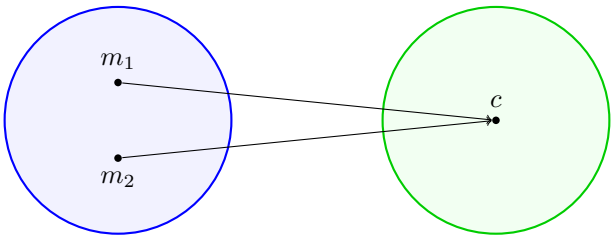


Figure 3.1: A diagram that shows a non-injective encryption function.

maps two different plaintexts to the same ciphertext or it isn't defined for some plaintexts (i.e., it can't encrypt some plaintexts). In the first case, the decryption function can't decrypt a message since it can't decide which of the multiple plaintexts that can originate it is the correct one whereas in the latter case, the encrypting function would be useless since it works only on some inputs.

Let's now move to the second inequality, namely $|\mathcal{C}| \leq |\mathcal{K}|$. Let's start by saying that the probability of observing a ciphertext has to be strictly positive ($Pr(\mathsf{C} = c) > 0$), otherwise we can alter the definition of the ciphertext space (i.e., we can define a new $\mathcal{C}$ with only the ciphertext that can be generated). Thanks to Theorem 3.1 we know that

$$Pr(\mathsf{C} = c) = Pr(\mathsf{C} = c|\mathsf{P} = m) \quad \forall m \in \mathcal{M}\ \forall c \in \mathcal{C} \tag{3.11}$$

and because we've shown that $Pr(\mathsf{C} = c) > 0$, then it must also be true that

$$Pr(\mathsf{C} = c|\mathsf{P} = m) > 0 \quad \forall m \in \mathcal{M}\ \forall c \in \mathcal{C} \tag{3.12}$$

This means that for every ciphertext c and for every plaintext m there must be at least one key $k \in \mathcal{K}$ such that c has been obtained encrypting m with key k, namely that

$$\mathbb{E}_k(m) = c \tag{3.13}$$

Since it exists at least one key for each c that satisfies Equation 3.13, then the number of keys is at least equal to the number of ciphertexts, hence we have $|\mathcal{K}| \geq |\mathcal{C}|$. □

Finally, we are ready to state Shannon's theorem which defines the conditions under which a cryptoscheme is perfect.

Theorem 3.3 (Shannon)**.** *Let*

$$(\mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \{\mathbb{E}_k : k \in \mathcal{K}\}, \{\mathbb{D}_k : k \in \mathcal{K}\})$$

denote a symmetric key cryptosystem where the keys are picked independently of plaintext values and $|\mathcal{M}| = |\mathcal{C}| = |\mathcal{K}|$*. The cryptosystem is perfectly secure if and only if:*

1. *Every key is used with probability* $\frac{1}{|\mathcal{K}|}$*.*
2. *For each couple of message and ciphertext there exist a unique key that encrypts the message into the ciphertext, namely*

$$\forall (m, c) \in \mathcal{M} \times \mathcal{C} \quad \exists! k \in \mathcal{K} : \mathbb{E}_k(m) = c$$

Proof. Let us now consider the if part first. As a hypothesis, we know that

1. $Pr(\mathsf{K} = k) = \frac{1}{|\mathcal{K}|}$.
2. $\forall (m, c) \in \mathcal{M} \times \mathcal{C} \quad \exists! k \in \mathcal{K} : \mathbb{E}_k(m) = c$.

We can expand the probability $Pr(\mathsf{C} = c)$ of seeing a ciphertext as the probability that c is generated from m, or equivalently, that m has been obtained decrypted c. Namely, we get

$$Pr(\mathsf{C} = c) = \sum_{k : c \in \{\mathbb{E}_k(m), \forall m \in \mathcal{M}\}} Pr(\mathsf{K} = k) Pr(\mathsf{P} = \mathbb{D}_k(c)) \tag{3.14}$$

$$= \sum_{k : c \in \{\mathbb{E}_k(m), \forall m \in \mathcal{M}\}} \frac{1}{|\mathcal{K}|} Pr(\mathsf{P} = \mathbb{D}_k(c)) \qquad \text{hypothesis 1} \tag{3.15}$$

$$= \frac{1}{|\mathcal{K}|} \sum_{k : c \in \{\mathbb{E}_k(m), \forall m \in \mathcal{M}\}} Pr(\mathsf{P} = \mathbb{D}_k(c)) \tag{3.16}$$

$$= \frac{1}{|\mathcal{K}|} \sum_{m \in \mathcal{M}} Pr(\mathsf{P} = m) \qquad \text{hypothesis 2} \tag{3.17}$$

Note that the sum over all keys has been replaced with the sum over messages since we initially considered the keys for which at least one message can be encrypted into c. Since, for each message there exists only one key such that the message is mapped to c, then we are summing over all messages. Since we are considering the sum of the probabilities of all messages, we get 1 as the sum, hence we have

$$Pr(\mathsf{C} = c) = \frac{1}{|\mathcal{K}|} \tag{3.18}$$

We can now use this value to compute the probability $Pr(\mathsf{P} = m | \mathsf{C} = c)$, which is what we need to check that the cypher is perfect.

$$Pr(\mathsf{P} = m | \mathsf{C} = c) = \frac{Pr(\mathsf{C} = c | \mathsf{P} = m) Pr(\mathsf{P} = m)}{Pr(\mathsf{C} = c)} \qquad \text{Bayes' rules} \tag{3.19}$$

$$= \frac{Pr(\mathsf{C} = c | \mathsf{P} = m) Pr(\mathsf{P} = m)}{\frac{1}{|\mathcal{K}|}} \tag{3.20}$$

Since, for hypothesis 2, each ciphertext is mapped to one ciphertext, then when we observe a message m, the probability of obtaining c is the probability of picking the key that maps m to c, hence we have

$$Pr(\mathsf{C} = c|\mathsf{P} = m) = \frac{1}{|\mathcal{K}|} \tag{3.21}$$

If we replace this result we obtain

$$Pr(\mathsf{P} = m|\mathsf{C} = c) = \frac{\frac{1}{|\mathcal{K}|}Pr(\mathsf{P} = m)}{\frac{1}{|\mathcal{K}|}} \tag{3.22}$$

$$= Pr(\mathsf{P} = m) \tag{3.23}$$

And we've shown that the cypher is perfect.

Let us now consider the only if part. Now we have, as a hypothesis:

1. $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{M}|$.
2. $Pr(\mathsf{P} = m|\mathsf{C} = c) = Pr(\mathsf{P} = m)$.

Now we want to prove that $\mathbb{E}_k$ is injective. Let's start by proving that, given an arbitrary pair (m, c) of message and ciphertext, there must exist at least one key k such that $\mathbb{E}_k(m) = c$. If it wasn't so, then we would have $Pr(\mathsf{P} = m|\mathsf{C} = c) = 0$, since we have a ciphertext which is not obtained by any key but $Pr(\mathsf{P} = m)$ since every message can be picked. This denies the hypothesis, hence there must exist at least one key that maps a message into a ciphertext. Next, we want to prove that there exists at most one key k such that $\mathbb{E}_k(m) = c$. Say that there exist two keys k_1 and k_2 such that $\mathbb{E}_{k_1}(m) = c$ and $\mathbb{E}_{k_2}(m) = c$. Because of hypothesis 1, there should exist another ciphertext c' such that $\mathbb{E}_k(m) \neq c$ for every key k and the mapping $m \mapsto_k c'$ would be impossible for every key k. Hence we must have at most one key and, since we have also proved that we must have at least one key, then we have proved that there exists only one key and $\mathbb{E}_k$ is injective. Now that we know that encryption is injective, we can write

$$Pr(\mathsf{P} = m) = Pr(\mathsf{P} = m|\mathsf{C} = c) \tag{3.24}$$

$$= \frac{Pr(\mathsf{C} = c|\mathsf{P} = m)Pr(\mathsf{P} = m)}{\mathsf{C} = c} \qquad \text{Bayes' rules} \tag{3.25}$$

Now we can consider one specific key k, hence the probability $Pr(\mathsf{C} = c|\mathsf{P} = m)$ becomes the probability of obtaining a ciphertext c from a message m is the probability of picking the lone key that maps m into c. What we get is

$$Pr(\mathsf{P} = m) = \frac{Pr(\mathsf{K} = k)Pr(\mathsf{P} = m)}{Pr(\mathsf{C} = c)} \tag{3.26}$$

Now we can simplify and obtain

$$1 = \frac{Pr(\mathsf{K} = k)}{Pr(\mathsf{C} = c)} \tag{3.27}$$

$$Pr(\mathsf{C} = c) = Pr(\mathsf{K} = k) \tag{3.28}$$

This is true for each c, which are equally distributed, hence it must be

$$Pr(\mathsf{C} = c) = \frac{1}{|\mathcal{C}|} \tag{3.29}$$

But since $|\mathcal{K}| = |\mathcal{C}|$ for hypothesis, we have

$$Pr(\mathsf{K} = k) = \frac{1}{|\mathcal{K}|} \tag{3.30}$$

□

It's important to understand that, even if we know that a cypher is perfectly secure, we can't immediately derive a way of breaking it (i.e., of finding the key or plaintext given a ciphertext).

Vernam cypher

An example of a perfect cypher is the Vernam cypher which uses a sequence of random bits as a key. This means that each time we want to send a message we have to create a random sequence of bits of the same length of the message to send, which is then used to encrypt the message using a simple XOR operation. The Vernam cypher is perfectly secure because:

- The key is chosen at random (condition 1 of Theorem 3.3).
- A different key is used for each message and each message is mapped to a different ciphertext (condition 2 of Theorem 3.3). This is true because the bit-wise XOR operation defines an algebraic group over the set $\{0, 1\}$, then for each couple (m, c) there exists a unique key such that $m \oplus k = c$. More formally, assume there exist two $k_1, k_2 \in \mathcal{K}$ such that $k_1 \neq k_2$, $m \oplus k_1 = c$ and $m \oplus k_2 = c$. This means we can write

$$m \oplus k_1 = m \oplus k_2 \tag{3.31}$$
$$k_1 = k_2 \tag{3.32}$$

  hence the hypothesis must be wrong and there must exist only one key k.

Because of these properties, this cypher is also called **One-Time-Pad** (OTP) enciphering machine. The Vernam cypher is theoretically perfect, however, it can't be used in practice since it requires the key to be physically moved from the source of the message to the destination, hence shifting the problem from software security to physical security (in the sense of securing the process used to transfer the drive in which the key is stored). More schematically, when a source Alice wants to send a message m to Bob:

1. Alice generates a random key k of the same length as the message.
2. Alice encrypts the message m using the key k, hence obtaining $\mathbb{E}_k(m)$.
3. The encrypted message is sent to Bob via a communication channel.
4. The key k is physically sent to Bob via a side channel.
5. Bob deciphers $\mathbb{E}_k(m)$ using the key k.

The problem with this cypher isn't only in the way keys are handled (since moving keys is expensive) but also in the process of generating random keys, in fact, it's very expensive to generate securely random keys. Moreover, we have to be sure that a key isn't reused, in fact, an attacker could mount a known plaintext attack which allows to obtain the key. The attacker can ask to encrypt a known text and obtain the key as $k = m \oplus c$. Moreover, if two messages are encrypted using the same key

we can obtain some information about the plaintexts knowing only the ciphertexts. In practice, say Alice sends

$$c_1 = k \oplus m_1$$

and

$$c_2 = k \oplus m_2$$

If we xor the two messages we obtain

$$\begin{aligned} c_1 \oplus c_2 &= (k \oplus m_1) \oplus (k \oplus m_2) \\ &= k \oplus m_1 \oplus k \oplus m_2 \\ &= m_1 \oplus m_2 \end{aligned}$$

3.1.3 Computationally secure cyphers

As we have seen from the Vernam cypher, perfect cyphers do exist but are practically not feasible to implement. This means that we have to shift our attention to computationally secure cyphers with a long time horizon, i.e., computationally secure cyphers that require an immense amount of time to break (ideally, many times the life of the universe). Computationally secure cyphers use a shorter fixed-length key with respect to perfect cyphers (typically 128 or 256 bits) and avoid:

- The disclosure of the plaintext from the ciphertext.
- The disclosure of the key from plaintext-ciphertext pairs.

This means that breaking the cypher requires trying 2^{256} keys (or in general $2^{input\ bit\ length}$), which is practically unfeasible even with a huge computational power.

3.2 Entropy

Entropy is a key tool for evaluating a cypher and allows quantifying the information held by a message (in general, without focusing on plaintexts, ciphertexts or keys). In particular, information is related to uncertainty, in fact, if we are uncertain about what message a source would generate, then when we receive a message we obtain the maximum information because we had no a-priori information on the message we could receive. Consider for instance a sender which picks a message among a set of n messages and sends it to a receiver across a communication channel. If the probability of picking a message is the same for all messages (i.e., $Pr(m) = \frac{1}{n}$), when the receiver sees the message, it has

- gained the maximum information, and
- lost the most amount of uncertainty since it was maximally uncertain about the message to receive, and it lost such uncertainty when the message arrives.

On the other hand, if the source picks a message m_1 with probability 1 and all the other messages with probability 0, then the receiver isn't surprised to see message m_1, hence he or she

- gains no information, and
- loses no uncertainty about the source.

From a crypto-analysis point of view, entropy measures the level of uncertainty an attacker has about either the correct plaintext or the correct key, hence it quantifies the amount of information leaked by the ciphertext.

Before giving the formal definition of entropy, let us consider an example to have an intuition about what entropy is and how it's measured. Say we have a sender, Alice, who sends a single bit to a receiver, Bob. The message sent by Alice (which can be either 0 or 1) is chosen uniformly at random, hence Bob is maximally unsure about what he will receive and when he gets the message he discovers 1 full bit of information. On the other hand, if Alice sends always 1, then Bob doesn't really care about the message he receives since he's sure that it will be 1. This means that the message sent by Alice has an entropy of 0 bits since it doesn't contain any useful information to Bob. This means that entropy, i.e., information, should be 1 (and maximum) when the source generates messages uniformly at random and 0 (and minimum) when we are sure about the message sent by the source. Now we are ready to introduce the formal definition of entropy.

Definition 3.2 (Entropy)**.** *Let* X *be a random variable which takes values in* $\{x_1, x_2, \ldots, x_n\}$ *with probability distribution* $p_i = Pr(\mathsf{X} = x_i), \forall 1 \leq i \leq n$*. The entropy of* X *is defined as:*

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

assuming conventionally that $p_i \log_2 p_i = 0$ *if* $p_i = 0$*.*

Let us now analyse the main characteristics of the entropy by highlighting what each element of the formula is:

- p_i is the probability of picking $\mathsf{X} = x_i$.
- $\log_2 p_i$ is the logarithm of the probability of picking $\mathsf{X} = x_i$. Since the probability p_i is a number between 0 and 1, then the logarithm must be non-positive. Since we want the entropy to be positive, we can consider adding a $-$ sign to the logarithm. If we take the minus outside of the sum, we obtain the formulas written in the definition. Also note that we have considered the logarithm with base 2 because it's handy in computer science since we always work with bits, however we can write equivalent formulas using different bases. Also note that the logarithm of the probability allows summing probabilities instead of multiplying them.

We can now analyse some important properties of the entropy:

- The entropy of X is always a non-negative value

$$H(\mathsf{X}) \geq 0 \tag{3.33}$$

  because it's the sum of non-negative values (considering $-\log_2 p_i$) multiplied by a positive value (the probability p_i).

- The entropy of X is 0 only if $p_j = 1$ and $p_i = 0 \forall i, i \neq j$. This confirms our initial intuition that the entropy should have been 0 when we are sure about the message sent (i.e., when the probability of a message is 1 and the others are 0).
- The entropy of X is $\log_2 n$ if all values of X have the same probability of being picked, namely if $p_i = \frac{1}{n} \forall i$.

Moreover, the properties above define the bounds for the entropy. This result is stated in the following theorem.

Theorem 3.4 (Entropy bounds). *Let* X *be a random variable which takes values in* $\{x_1, x_2, \ldots, x_n\}$. *Then it's always true that*

$$0 \leq H(\mathsf{X}) \leq \log_2 n$$

Note that, entropy also assesses by how much one can compress the representation of the information.

3.2.1 Key equivocation

An important concept related to entropy is key equivocation. Before diving into the details of key equivocation we need some useful definitions.

Definition 3.3 (Joint entropy). *Let* X, Y *be random variables over in* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_m\}$, *respectively. The joint entropy of* X *and* Y *is*

$$H(\mathsf{X}, \mathsf{Y}) = -\sum_{i=1}^{n}\sum_{j=1}^{m} Pr(\mathsf{X} = x_i, \mathsf{Y} = y_j) \log_2 Pr(\mathsf{X} = x_i, \mathsf{Y} = y_j) \tag{3.34}$$

and defines the amount of information we get by observing a pair of values (x, y).

Definition 3.4 (Conditioned entropy). *Let* X, Y *be random variables over* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_m\}$, *respectively. The entropy of* X *given that we have observed* $\mathsf{Y} = y$ *is*

$$H(\mathsf{X}|\mathsf{Y} = y) = -\sum_{i=1}^{n} Pr(\mathsf{X} = x_i|\mathsf{Y} = y) \log_2 Pr(\mathsf{X} = x_i|\mathsf{Y} = y) \tag{3.35}$$

Definition 3.5 (Conditional entropy). *Let* X, Y *be random variables over* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_m\}$, *respectively. The conditional entropy of* X *given* Y *is*

$$H(\mathsf{X}|\mathsf{Y}) = -\sum_{j=1}^{m} Pr(\mathsf{Y} = y_j) H(\mathsf{X}|\mathsf{Y} = y_j) \tag{3.36}$$

$$= -\sum_{i=1}^{n}\sum_{j=1}^{m} Pr(\mathsf{Y} = y_j) Pr(\mathsf{X} = x_i, \mathsf{Y} = y_j) \log_2 Pr(\mathsf{X} = x_i, \mathsf{Y} = y_j) \tag{3.37}$$

and defines the amount of information you get after a value of Y *has been revealed.*

Now we can use the definition above to state some important theorems.

Theorem 3.5. *Let* X, Y *be random variables over* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_n\}$*, respectively. Then it's always true*

$$H(\mathsf{X}, \mathsf{Y}) \leq H(\mathsf{X}) + H(\mathsf{Y}) \tag{3.38}$$

This means that, given two symbols their joint entropy isn't more than the sum of the entropy of the two symbols when they are observed separately. This means that if we see two messages in the same context, we can't get more information than seeing the messages separately. Equivalently, the entropy of two symbols is the same, independently of the fact that they are related or not.

Theorem 3.6. *Let* X, Y *be random variables over in* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_n\}$*, respectively. Then it's always true*

$$H(\mathsf{X}, \mathsf{Y}) = H(\mathsf{Y}) + H(\mathsf{X}|\mathsf{Y}) \tag{3.39}$$

This theorem tells us that the information we get from two symbols is the same as we get from the second symbol summed to the information we get from the first, knowing that we have seen the second.

Theorem 3.7. *Let* X, Y *be random variables over in* $\{x_1, x_2, \ldots, x_n\}$ *and* $\{y_1, y_2, \ldots, y_n\}$*, respectively. Then it's always true*

$$H(\mathsf{X}|\mathsf{Y}) \leq H(\mathsf{X}) \tag{3.40}$$

This comes from the fact that a perfect cypher doesn't leak any information, hence we don't get any information from observing a realisation of Y.

Application to cyphers

The definitions and properties are rather generic however we would like to apply them to cryptoanalysis. Let us therefore state some useful properties:

- The entropy of a plaintext given the key used to decrypt it and the corresponding ciphertext is null
$$H(\mathsf{P}|\mathsf{K}, \mathsf{C}) = 0 \tag{3.41}$$
since we can decrypt the ciphertext using the key and get the plaintext.

- The entropy of a ciphertext given the key used to encrypt it and the corresponding plaintext is null
$$H(\mathsf{C}|\mathsf{P}, \mathsf{K}) = 0 \tag{3.42}$$
since we can encrypt the plaintext the key and get the ciphertext.

- The entropy of a triple plaintext, key and ciphertext is

$$\begin{aligned} H(\mathsf{P}, \mathsf{K}, \mathsf{C}) &= H(\mathsf{P}, \mathsf{K}) + H(\mathsf{C}|\mathsf{P}, \mathsf{K}) && \text{Theorem 3.6} && (3.43)\\ &= H(\mathsf{P}, \mathsf{K}) && \text{Property 3.42} && (3.44)\\ &= H(\mathsf{P}) + H(\mathsf{K}) && \text{Theorem 3.5} && (3.45) \end{aligned}$$

- The entropy of a triple plaintext, key and ciphertext is

$$\begin{aligned} H(\mathsf{P},\mathsf{K},\mathsf{C}) &= H(\mathsf{K},\mathsf{C}) + H(\mathsf{P}|\mathsf{K},\mathsf{C}) && \text{Theorem 3.6} && (3.46)\\ &= H(\mathsf{K},\mathsf{C}) && \text{Property 3.41} && (3.47)\\ &= H(\mathsf{K}) + H(\mathsf{C}) && \text{Theorem 3.5} && (3.48) \end{aligned}$$

If we combine Properties 3.45 and 3.48 we obtain

$$\begin{aligned} H(\mathsf{P},\mathsf{K},\mathsf{C}) &= H(\mathsf{P}) + H(\mathsf{K}) && && (3.49)\\ H(\mathsf{K},\mathsf{C}) &= H(\mathsf{P}) + H(\mathsf{K}) && \text{Eqn. 3.47} && (3.50) \end{aligned}$$

Key equivocation

Now we are ready to give the definition of key equivocation.

Definition 3.6 (Key equivocation)**.** *Key equivocation is the amount of information (uncertainty) about a key obtained by the knowledge of a ciphertext, namely* $H(\mathsf{K}|\mathsf{C})$.

Thanks to Theorem 3.6 we can write

$$H(\mathsf{C},\mathsf{K}) = H(\mathsf{C}) + H(\mathsf{K}|\mathsf{C}) \tag{3.51}$$

from which we can extract

$$\begin{aligned} H(\mathsf{K}|\mathsf{C}) &= H(\mathsf{K},\mathsf{C}) - H(\mathsf{C}) && && (3.52)\\ &= H(\mathsf{P}) + H(\mathsf{K}) - H(\mathsf{C}) && \text{Eqn. 3.50} && (3.53) \end{aligned}$$

The result we have just obtained is very important since it allows us to define the key equivocation in terms of the bits of information given by plaintext, ciphertext and key. Moreover, this formula gives a bound on the information we can get from a ciphertext, having the key, independently from the cryptoscheme used. Namely, if we see a ciphertext of n bits, we still have to recover $H(\mathsf{P})+H(\mathsf{K})-n$ bits of the key. In other words, key equivocation measures how uncertain we are about a key given a ciphertext and how many bits of the key we still have to disclose. For instance, if we have

$$H(P) = 1.95 \qquad H(K) = 1.5 \qquad H(C) = 2$$

then we can compute

$$H(\mathsf{K}|\mathsf{C}) = H(\mathsf{P}) + H(\mathsf{K}) - H(\mathsf{C}) = 1.95 + 1.5 - 2 = 1.45$$

Since $H(K|C) = 1.45$ and $H(K) = 1.5$, then knowing a ciphertext reduces the entropy of the key by 0.05 bits, hence we get 0.05 bits of information for every ciphertext we get. Note however that even if this formula tells us how many bits of information we get from a ciphertext, it doesn't explain how to obtain such information.

If we consider perfect cyphers, then the entropy of ciphertext, plaintext and key is maximum since messages and keys are picked and generated at random. If we call H_{max} the maximum value of the entropy we get

$$\begin{aligned} H_{\text{perfect cypher}}(\mathsf{K}|\mathsf{C}) &= H_{\text{perfect cypher}}(\mathsf{P}) + H_{\text{perfect cypher}}(\mathsf{K}) - H_{\text{perfect cypher}}(\mathsf{C}) && (3.54)\\ &= H_{max} + H_{max} - H_{max} && (3.55)\\ &= H_{max} && (3.56) \end{aligned}$$

This means that even by observing a ciphertext we can't get any information from the ciphertext because the entropy of the key is the same when not observing the ciphertext.

3.2.2 Spurious keys and unicity distance

If a message doesn't have the maximum entropy value, it might help in finding some bits of the key. This is due to redundancy in many languages used to write messages, in fact, even PDFs have a structure that with redundancies. What we want to do now is to quantify how many bits of the key we can get from a message (i.e., from plaintext). Since we want to exploit redundancies in a language we first have to formally define redundancies.

Definition 3.7 (Language redundancy)**.** *Let L be a language, $|\mathcal{M}|$ the number of symbols of the language and H_L the entropy per letter of L. The redundancy of language L is*

$$R_L = 1 - \frac{H_L}{\log_2 |\mathcal{M}|} \tag{3.57}$$

Redundancies tell us the percentage of text that is redundant in a generic phrase, hence we can use it to understand what is the best a compression algorithm can do to reduce the size of a text. If we consider, for instance, the English language where $|\mathcal{M}| = 26$ and $H_{English} = 1.25$ (it has been estimated that $1 \leq H_{English} \leq 1.5$) then we get

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

This means that 10MB of English text may be encoded in 2.5MB since 75 % of the text is redundant.

Spurious keys

If we consider a language with meaning, some ciphertexts can be decrypted only by a subset $\mathcal{K}_{meaningful}$ of the whole set of keys $\mathcal{K}$. Namely, only the keys in $\mathcal{K}_{meaningful}$ return a meaningful message when decrypting a ciphertext. The keys belonging to $\mathcal{K}$ but not to $\mathcal{K}_{meaningful}$ are called spurious keys, namely

$$\mathcal{K}_{spurious} = \mathcal{K} \setminus \mathcal{K}_{meaningful}$$

In other words, the spurious keys are those that do not decrypt a ciphertext into a valid (i.e., meaningful) plaintext. Let us now consider a cypher

$$(\mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \mathbb{E}_k, \mathbb{D}_k)$$

where each ciphertext word is composed of n symbols and the number of keys is the same as the number of ciphertexts, namely $|\mathcal{K}| = |\mathcal{C}|$. If we consider a ciphertext $c \in \mathcal{C}$ and we call $|\mathcal{K}_{meaningful}(c)|$ the number of keys that decrypt c into a meaningful plaintext (i.e., $\mathbb{D}_k(c) \in \mathcal{M}, \forall k \in \mathcal{K}_{meaningful}(c)$), then we can compute the average number of spurious keys. Formally we get:

Definition 3.8 (Average number of spurious keys)**.** *Let*

$$(\mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \mathbb{E}_k, \mathbb{D}_k)$$

be a cypher such that each ciphertext word is composed of n symbols and $|\mathcal{K}| = |\mathcal{C}|$. Let

$$\mathcal{K}_{meaningful}(c) = \{k : \mathbb{D}_k \in \mathcal{M}\}$$

The average number of spurious keys of the cypher is

$$\bar{s}_n = \sum_{c \in \mathcal{C}} Pr(\mathsf{C} = c)(|\mathcal{K}_{meaningful}(c)| - 1) \tag{3.58}$$

Note that, as the length of the ciphertext increases (i.e., for $n \to \infty$) we can obtain

$$\bar{s}_n \geq \frac{|\mathcal{K}|}{|\mathcal{M}|^{nR_L}} - 1 \tag{3.59}$$

As we can see, the number of spurious keys decreases as the length of the ciphertext increases. This is helpful for an attacker for which it's ideal when $\bar{s}_n = 0$.

Unicity distance

Definition 3.9 (Unicity distance)**.** *The unicity distance is the length of ciphertext words (i.e., the number of ciphertexts) $n = n_0$ such that the number of spurious keys is equal to zero, i.e. such that $\bar{s}_n = 0$. The unicity distance can be approximated as*

$$n_0 \approx \frac{\log_2 |\mathcal{K}|}{R_L \log_2 |\mathcal{M}|} \tag{3.60}$$

More importantly, **the unicity distance n_0 is the number of ciphertext symbols we provide to a bruteforcer to be reasonably sure that the first meaningful output is the right plaintext**. This means that n_0 represents the number of ciphertexts required by an attacker to do an exhaustive key search in a ciphertext-only attack. Note that if $R_L = 0$, i.e., the plaintext is random, n_0 goes to infinity.

Let us consider an example to understand the meaning of unicity distance. Consider for instance the a substitution cypher with $|\mathcal{M}| = 26$ and $|\mathcal{K}| = 26!$. This means that $R_L \approx 0.75$ and n_0 is

$$n_0 \approx \frac{\log_2 26!}{0.75 \cdot \log_2 26}$$

In a generic modern symmetric-key cypher (which encrypts bit-strings) we may have that $|\mathcal{M}| = |\mathcal{C}| = 2$, and $|\mathcal{K}| = |\{0,1\}|^l$, for some bit length $l \gg 1$, while the plaintext language is English. Assuming $R_L \approx 0.75$ (which is an under-estimate as we encode English letters with ASCII). The unicity distance is: $n_0 \approx \frac{l}{R_L} \approx \frac{4l}{3}$. If we would be able to compress the plaintext in a perfect manner, to have $R_L \approx 0$, then n_0 would go to infinity.

Modern cyphers encrypt plaintexts with redundancy even when some lossless compression technique is applied. Both symmetric and asymmetric cyphers usually add to the plaintext some redundancy to counter active attacks like KPAs, CPAs and CCAs. Note that, the considerations about the unicity distance are valid only if the threat model defines a passive attacker (i.e., COA).

Chapter 4

Block cyphers

4.1 Introduction

Most of the symmetric cyphers in use nowadays are block cyphers. Shannon theorised that block cyphers should have good statistical properties, and that, to achieve them a cypher should be built as an iterative application of confusion and diffusion. Let us, therefore, define confusion and diffusion:

> **Definition 4.1** (Confusion)**.** *Confusion ensures that the relationship between ciphertext, plaintext and key is non-predictable and as hard as possible to define. Confusion makes it so that every bit of the key used to encrypt a plaintext influences the correspondence between plaintext and ciphertext in a non-predictable way.*

Confusion makes frequency analysis impossible and can be implemented as a very complex substitution using a predefined table.

> **Definition 4.2** (Diffusion)**.** *Diffusion ensures that the statistical distribution of groups of plaintext letter frequencies should be as flat as possible. Namely, the ciphertext should look like random text.*

In practice, flipping a bit of plaintext yields a completely different and unpredictable ciphertext. The diffusion property makes statistical analysis impossible and is implemented using permutations which ensure that the single letters have the same distribution before the permutation but digrams and trigrams are spread out and divided. Note that if a cypher has poor diffusion, then known-plaintext attacks are possible. Thanks to these definitions we can say that a block cypher can be obtained iterating a combination of

- **Substitutions**, which ensure the confusion property because we scramble the digits of the plaintext in a non-predictable way.
- **Permutations**, which ensure the diffusion property because we move each digit of the plaintext in a different position.

4.2 General block cyphers structure

Block cyphers operate on blocks or chunks of text of a fixed size. This means that a block cypher takes as input a message

$$\mathbf{m} = (m_1, m_2, \ldots, m_n) \in \mathcal{M}$$

of size n with $m_i \in \{0, 1\}$ and a key

$$\mathbf{k} = (k_1, k_2, \ldots, k_m) \in \mathcal{K}$$

with $k_i \in \{0, 1\}$ to produce a block of ciphertext

$$\mathbf{c} = (c_1, c_2, \ldots, c_n) \in \mathcal{C}$$

of the same size n of the plaintext through a function $\mathbb{E}_k$. The cypher also decrypts a block of ciphertext $\mathbf{c}$ through the function $\mathbb{D}_k$ to obtain a block of plaintext of the same length. The size of the block cypher (i.e., of the block in input to $\mathbb{E}_k$ and $\mathbb{D}_k$) is the same for encryption and decryption and usually spans between 64 and 256 bits.

A message is usually bigger than a cypher block, hence it is split into chunks of the correct size and each chunk is encrypted (or decrypted) using $\mathbb{E}_k$ (or $\mathbb{D}_k$). The **mode of operation of a cypher** defines the scheme used to encrypt the whole message, namely how to combine the results obtained from encrypting (or decrypting) a single block, i.e., a single chunk (more on this later). For instance one could combine a ciphertext $\mathbf{c}_i$ to a plaintext $\mathbf{m}_{i+1}$ before encrypting $\mathbf{m}_{i+1}$.

If the message length is not a multiple of the block length n, we can pad the message and obtain a length divisible by n. Some padding options are:

- Add as many known values as desired at the end of the message (e.g., pad with all zeros).
- Add the length of the message at the end of the message.
- Add the length of the padding at the end of the message.

4.2.1 Purposes

A block cypher has to ensure:

- **Mathematical security**. A cypher has to ensure the blending of plaintext and ciphertext (i.e., confusion) and statistical flatness (i.e., diffusion). Moreover, the cypher has to be analysable.
- **Efficiency**. A cypher should have good figures of merit for encryption and decryption on every platform. This means that the cypher should be fast and efficient.

4.2.2 High level structure

As we have said, a block cypher is implemented as an iteration of permutations and substitutions. The result of each operation (either a permutation or a substitution or any other operation) is called **cypher state**. The initial cypher state, in the case of encryption, is the plaintext while the last cypher state is the ciphertext. A sequence of operations applied to a state is called **round**. The operations in a round usually involve a cypher key. Since the encryption transformation is made of many rounds, it could be useful to use different keys, which are generated starting from the encryption key, for each round. In particular, the key is expanded in r round keys without

changing the entropy of the key (i.e., no information about the key is lost when expanding it). The key expansion procedure is called **key schedule**. Note that the key schedule has to be reversible, which means that, given some round keys, we can obtain the encryption key back. This is necessary to ensure that the key information is not lost in the expansion but it's also dangerous since an attacker might want to get the original key from observing some round keys. Also keep in mind that the key schedule might generate round keys of different lengths, depending on the encryption transformation.

In general, a block cypher:

1. Is fed with a plaintext and a key.
2. Expands the key using a key schedule.
3. Combines each round key with the cypher state in a sequence of iterations (or rounds).

Depending on the encryption and decryption transformations, we have two different classes of block cyphers:

- **Feistel cyphers**. A Feistel cypher splits the cypher state into two parts and acts on each part separately. Moreover, the structure of the encryption and decryption transformation is the same.
- **Substitution Permutation Networks** (SPNs). A Substitution Permutation Network works on the whole cypher state and uses two different structures for encryption and decryption. An SPN applies:
  - A non-linear function, called **substitution box** (S-box), providing confusion. An S-box is represented as a lookup table.
  - A **linear function** providing diffusion. Some examples are bit-wise permutations or pairs of rotate and xor operations.
  - The addition of a **part of the key schedule**.

4.3 Feistel network

A Feistel network is a general framework for building a symmetric cypher. To build a Feistel network, we only have to specify a non-linear function that combines a key and a block of text. A Feistel network splits a cypher state $\mathbf{m}$ (i.e., a block or chunk) into two parts $\mathbf{L}$ and $\mathbf{R}$ of the same length $\frac{n}{2}$ (even if there also exist unbalanced Feistel networks) and applies different operations on each part for a finite number of rounds. More precisely, the cypher is initialised with the state $\mathbf{m} = (\mathbf{L}_0, \mathbf{R}_0)$ and r rounds are applied ($r-1$ identical round and a final round) to obtain the ciphertext $\mathbf{c} = (\mathbf{R}_r, \mathbf{L}_r)$. Note that the output has the left and right parts swapped since the last round swaps the left and right parts. Each round takes the left and right parts and

1. Computes $L \oplus \mathcal{F}(k_i, R)$ where
   - $\mathcal{F}$ is a non-linear (and possibly non-invertible) function.
   - k_i is the i-th round key obtained from the key schedule.

   The value computed is used as the right part for the next round, namely

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

2. Uses the right part as the left part for the next round, namely

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

Algorithm 1 shows an implementation of a Feistel cypher and Figure 4.1 shows a graphical representation of a network. The same algorithm can be used for encryption and for decryption, in fact, we can feed the cypher with the output of the cypher itself and obtain the original message back. The only caveat is that keys should be used in reverse order in decryption with respect to the order used for encryption. Note this property is true regardless of the function $\mathcal{F}$ we decide to use. Let us now show that the same structure can be used both for encryption and decryption. We know that the right and left parts at round i can be obtained as

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

and

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

Because the xor operation is invertible, we can also write, from the first equality,

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

Now we can replace $R_{i-1} = L_i$, obtained from the second equation to obtain

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

This means that, given (L_{r-1}, R_{r-1}) we can compute (L_{r-2}, R_{r-2}) and repeat the same operation until we reach (L_0, R_0). If this isn't enough, let us consider a Feistel network with two rounds. When encrypting a message:

1. We start from
$$(L, \quad R)$$
2. We compute
$$(R, \quad L \oplus \mathcal{F}(k_0, R))$$
3. We compute
$$(R \oplus \mathcal{F}(k_1, L \oplus \mathcal{F}(k_0, R)), \quad L \oplus \mathcal{F}(k_0, R))$$
which is the ciphertext obtained from (L, R).

Let us now feed the ciphertext to the same Feistel network, passing keys in reverse order. We get:

1. We start from
$$(R \oplus \mathcal{F}(L \oplus \mathcal{F}(R, k_0), k_1), \quad L \oplus \mathcal{F}(R, k_0))$$
2. We get
$$L \oplus \mathcal{F}(R, k_0)$$
on the left side and
$$R \oplus \mathcal{F}(L \oplus \mathcal{F}(R, k_0), k_1) \oplus \mathcal{F}(L \oplus \mathcal{F}(R, k_0), k_1) = R \tag{4.1}$$
on the right side.

3. We get
$$L \oplus \mathcal{F}(R, k_0) \oplus \mathcal{F}(R, k_0) = L$$
on the left side and
$$R \tag{4.2}$$
on the right side. This means that we have obtained the original plaintext back.

This works for any number of rounds. A visual representation of what we have just explained is shown in Figures 4.2a and 4.2b.

Algorithm 1 A Feistel cypher.

procedure FEISTEL((L, R), k)
  for $i \in \{0, \dots, r-2\}$ **do**
    $t \leftarrow L$
    $L \leftarrow R$
    $R \leftarrow t \oplus \mathcal{F}(k_i, R)$ ▷ $L_i \leftarrow R_{i-1}, R_i \leftarrow L_{i-1} \oplus \mathcal{F}(k_i, R_{i-1})$
  end for
  $R \leftarrow L \oplus \mathcal{F}(k_{r-1}, R)$
  return (R, L)
end procedure

4.3.1 DES

The Data Encryption Standard (DES) cypher uses a Feistel network. Other examples of cyphers that use a Feistel network are

- Blowfish.
- Twofish.
- CAST5.

DES modifies a little the Feistel framework by adding an initial and final permutation of the state (where the final permutation is simply the opposite of the initial one, hence it moves the bits in their original positions). This isn't done for security reasons but simply to make the hardware layout less complex. A permutation is in fact implemented in hardware as a scramble of wires. Structurally, DES is a Feistel network with:

- **16 rounds**.
- **64-bit input size**.
- **56-bit key**. The key is actually 64 bits long but it contains 8 parity bits, hence the useful information is stored in 56 bits. In practice, the 64-bit key is obtained by splitting the 56 bits key into 8 blocks of 7 bits and computing the parity bit of each block.
- A **selection and permutation key schedule**. An important feature of the DES key schedule is that the last round key is the same as the first, namely $k_0 = k_{15}$.

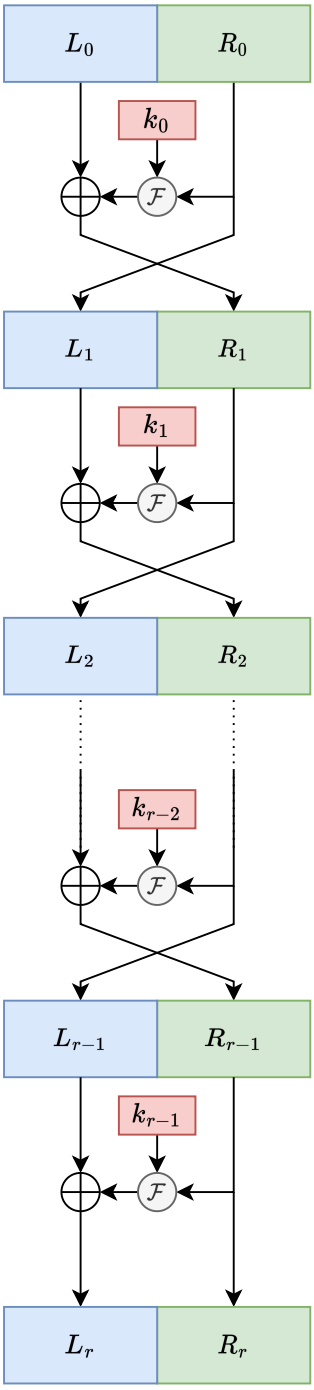


Figure 4.1: A Feistel network.

DES round

DES uses the Feistel framework, hence we simply have to define the non-linear function $\mathcal{F}$ which is combined with the round key (which is 48 bits long in DES). The function $\mathcal{F}$ is defined as

$$\mathcal{F}(k_i, R_{i-1}) = \text{Pbox}(\text{Sbox}(k_i \oplus \text{Ebox}(R_{i-1}))) \tag{4.3}$$

where

- Sbox is a substitution function.
- Pbox is a permutation function.
- Ebox is an expansion function.

Substitution box A substitution box, implemented using 8 lookup tables, maps a 48-bit word into a 32-bit word. In particular, the input of the Sbox is split in 8 chunks $b_0, \ldots, b_5$ of 6 bits each and the i-th chunk b_i is mapped using the i-th lookup table S_i which maps 6 bits into 4 bits. Each lookup table is a 4×16 table and

- The first and last bits of the Sbox input are used to index the lookup table's row.
- The second, third, fourth and fifth bits of the Sbox input are used to index the lookup table's column.

Formally, a substitution box can be implemented as in Algorithm 2.

Algorithm 2 A substitution box.

procedure S-BOX($i_0, i_1, i_2, i_3, i_4, i_5$, b)
  $r \leftarrow 2 \cdot i_0 + i_5$
  $c \leftarrow 8 \cdot i_1 + 4 \cdot i_2 + 2 \cdot i_3 + i_4$
  return $S_b[r, c]$
end procedure

Permutation box A permutation box allows shuffling the characters in the cypher state. A permutation box is implemented as a table where in position i it is specified the position of the input bit to put in position i of the output. Namely, we are saying that

```
out[i] = in[Pbox[i]]
```

The same mechanism is used for the initial and final permutations (outside the Feistel network).

Expansion box An expansion box expands its input from 32 to 48 bits by replicating some bits. Note that the expansion function is fixed.

Properties

The reasons behind some design choices done when creating the DES protocol haven't been disclosed, however, we know that some properties hold.

Complementation property The complementation property states that

Theorem 4.1 (DES complementation property)**.** *Inverting the bit values of the input plaintext m and key k yields a ciphertext equal to the bit-wise inversion of the result of* $\mathrm{DES}(k, m)$*, namely*

$$\mathrm{DES}(k, m) = \overline{\mathrm{DES}(\bar{k}, \bar{m})} \quad \forall m \in \mathcal{M}, \forall k \in \mathcal{K}$$

This property makes chosen plaintext attacks easier. Assume that we can collect two pairs

$$(m_1, c_1 = \mathrm{DES}(k, m_1))$$

and

$$(\overline{m_1}, c_2 = \mathrm{DES}(k, \overline{m_1}))$$

If we apply the complementation property we can write

$$\begin{aligned} c_2 &= \mathrm{DES}(k, \overline{m_1}) \\ &= \overline{\mathrm{DES}(\bar{k}, \overline{\overline{m_1}})} \\ &= \overline{\mathrm{DES}(\bar{k}, m_1)} \end{aligned}$$

By applying the bit-wise inversion we can obtain $\overline{c_2} = \mathrm{DES}(\bar{k}, m_1)$. This means that we have the message m_1, we can test both k and $\bar{k}$, hence we have to try half the keys and the complexity of a bruteforce attack goes from 2^{56} to 2^{55}.

Weak keys Some keys yield a very important property which mines the security of DES. Weak keys are keys such that if we encrypt the plaintext twice using the same key (i.e., the weak key) we obtain the plaintext. Namely,

$$\mathrm{DES}(k, \mathrm{DES}(k, m)) = m \quad \forall m \in \mathcal{M}$$

Semi-weak keys Some key pairs (k_1, k_2) behave like weak keys, in fact, if we encrypt a message with k_1 and then we encrypt the result with k_2 we obtain the original message. Namely,

$$\mathrm{DES}(k_2, \mathrm{DES}(k_1, m)) = m \quad \forall m \in \mathcal{M}$$

DES group Another important fact is that DES is not a group. Formally:

Theorem 4.2 (DES group)**.** *Given a plaintext m and a pair of keys k_1, k_2, the set of DES bijective transformations is not closed under composition:*

$$\forall m \ \nexists k_3 : \mathrm{DES}(k_3, m) = \mathrm{DES}(k_1, \mathrm{DES}(k_2, m))$$

This means that encrypting a plaintext m by applying DES twice with two different keys, is not the same as encrypting m once with a third key k_3.

Double DES

Thanks to the computational power currently available DES isn't secure anymore. Double DES (2DES) tries to improve DES security by doubling the length of the key. The 2DES key k is split

into two keys k_1 and k_2 which are used to encrypt the input by applying DES twice. The output is therefore computed as

$$2\text{DES}(k_1, k_2, m) = \text{DES}(k_1, \text{DES}(k_2, m))$$

Double DES, having a key of length 2^{112} should be secure, however, it's still possible to attack it. To show how, let's start by writing

$$c = \text{DES}(k_1, \text{DES}(k_2, m))$$

Now we can decrypt from both sides of the equation to obtain

$$\text{DES}^{-1}(k_1, c) = \text{DES}(k_2, m)$$

This means that we can compute $\text{DES}(k_2, m)$ for all possible keys k_2, namely we compute

$$A_i = \text{DES}(k_{2,i}, m) \quad \forall i \in \{0, \dots, 2^{56} - 1\}$$

We can then compute $\text{DES}^{-1}(k_1, c)$ for all possible keys k_1, namely we compute

$$B_i = \text{DES}^{-1}(k_{1,i}, c) \quad \forall i \in \{0, \dots, 2^{56} - 1\}$$

Now we can check if we can find any couple (A_i, B_j) such that $A_i = B_j$ and when we find a match we find a key $k = (k_{(1,j)}, k_{(2,j)})$. This means that we simply have to try 2^{57} keys, which is better than bruteforcing 2^{112} keys. Moreover, since bruteforcing DES has a cost of 2^{55}, double DES isn't really better than DES.

Triple DES

If applying DES twice isn't enough, maybe doing it three times is. Triple DES (3DES) uses a key three times longer than the standard DES key, namely

$$3DES(k_1, k_2, k_3, m) = DES(k_1, DES^{-1}(k_2, DES(k_3, m)))$$

The 3DES key $k = k_1.k_2.k_3$ can be built using three different options:

- **3DES3**. All DES keys that compose the 3DES key are different.
- **3DES2**. Keys k_1 and k_2 are independent and k_3 is equal to k_1.
- **3DES1**. All keys are the same. This option is not secure (encryption with k_3 and decryption with k_2 cancel out) and is used only for back compatibility.

Triple DES can be attacked through a known plaintext attack using the same man-in-the-middle approach seen in double DES with a complexity of 2^{112}.

DES-X

DES-X is yet another variant of DES which increases the complexity of a known plaintext attack using a technique called pre-whitening. The idea is to use three independent keys, the first, called k, has a size of 56 bits while the others, k_1 and k_2, are 64 bits long. A ciphertext is obtained as

$$c = \text{DES-X}(k, k_1, k_2, m) = k_2 \oplus \text{DES}(k, m \oplus k_1)$$

The plaintext is obtained from a ciphertext as

$$m = \text{DES-X}^{-1}(k, k_1, k_2, c) = k_1 \oplus \text{DES}^{-1}(k, c \oplus k_2)$$

The main advantage of DES-X is that it's almost as fast as DES but it has a much longer key (184 bits against 56 bits for DES) and a way better security margin (2^{112} against 2^{55}).

4.3.2 DES analysis

DES can be analysed using advanced techniques like:

- **Linear cryptanalysis**. Linear cryptanalysis finds approximated linear relations (namely, Boolean equalities) among some bits of the plaintext and some bits in input to the last round of the cypher. It recovers, in a subsequent step, the values of some bits of the secret key.
- **Differential cryptanalysis**. Differential cryptanalysis finds how differences between two input plaintexts propagate within the cypher up to the beginning of the last round. In a subsequent step, this knowledge is exploited to obtain the values of some bits in the last sub-key.

4.4 Modes of operation

Until now we have defined a block cypher as a component that takes as input a message of a fixed length n and encrypts or decrypts it and we have called $\mathbb{E}_k$ and $\mathbb{D}_k$ such transformations. For instance, a Feistel Network is a way of implementing both $\mathbb{E}_k$ and $\mathbb{D}_k$. Not all messages are however of length n, hence we have to define how longer messages are handled. For starters, let us assume, without loss of generality, that the length of the message is a multiple of n. This isn't a problem because if not so, we can still pad the message by adding, for instance, a known fixed character of the length of the message.

Messages longer than a block are handled using modes of operation, which are ways to apply the encryption and decryption transformation to each chunk in which the message is split. More precisely, when we have to deal with a message of length kn, we split it into k blocks of length n and we apply $\mathbb{E}_k$ (or $\mathbb{D}_k$) using different schemes called modes of operations.

4.4.1 Electronic Code Book

The simplest mode of operation is called Electronic Code Book (ECB) and applies the encryption or decryption transformation to each block independently. This means that:

1. The message $\mathbf{m}$ is split into blocks $\mathbf{m}_1, \ldots, \mathbf{m}_k$ of size n.
2. Each block is encrypted using the transformation $\mathbb{E}_k$ to obtain $\mathbf{c}_i = \mathbb{E}_k(\mathbf{m}_i)$.
3. The ciphertexts are concatenated to obtain the whole ciphertext $\mathbf{c} = \mathbf{c}_1||\mathbf{c}_2||\ldots||\mathbf{c}_k$.

A graphical representation of an ECB is shown in Figure 4.3.

Advantages and disadvantages

The main advantages of the ECB mode of operation are:

- It's **very easy to implement**.
- It's **very fast**.
- It's **easily parallelisable** since every cypherblock can be computed independently.
- One bit of error in a block is propagated to a single block since every encryption (or decryption) is independent.

The main disadvantages are:

- If the same message is encrypted twice (i.e., two ciphertexts are generated from the same plaintext and the same key), using the same key, the ciphertext is the same.
- Because each block is encrypted independently an attacker can still recognise patterns in the ciphertext. In fact, the same block is mapped always to the same cypherblock. A famous example is the encryption of the Linux logo. If we encrypt the Linux logo using an ECB, we get an image which is different (i.e., has different colours) but we can still recognise the penguin. This is because blocks of pixels of the same colour pixels are mapped to the same block of pixels.
- It suffers from insertion attacks. This means that we can insert chunks of text to the message or to the ciphertext (of length n) since it will not impact how the other blocks are encrypted or decrypted, hence the insertion will not be noticed.

By analysing the disadvantages of ECB we can notice that the main issue is that all blocks are handled independently. This means that the only field in which this mode of operation can be applied is where the message is always shorter than a block.

4.4.2 Cypher Block Chaining

The Cypher Block Chaining (CBC) tries to solve ECB's problems by combining, with a xor operation, the previous ciphertext generated with the next message to encrypt. If we call $\mathbf{c}_0$ the value that has to be combined with the first message, we obtain a block of ciphertext as

$$\mathbf{c}_i = \mathbb{E}_k(\mathbf{c}_{i-1} \oplus \mathbf{m}_i)$$

The first cypherblock $\mathbf{c}_0$ is initialised with a random and public value, called initialisation vector (IV). If the key is used only once the IV can be fixed, otherwise we have to pick a different IV every time. Decryption works symmetrically, hence a plaintext block can be obtained as

$$\mathbf{m}_i = \mathbb{D}_k(\mathbf{c}_i) \oplus \mathbf{c}_{i-1}$$

In fact, we can write

$$\begin{aligned}\mathbf{m}_i &= \mathbb{D}_k(\mathbf{c}_i) \oplus \mathbf{c}_{i-1} \\ &= \mathbb{D}_k(\mathbb{E}_k(\mathbf{c}_{i-1} \oplus \mathbf{m}_i)) \oplus \mathbf{c}_{i-1} \\ &= \mathbf{c}_{i-1} \oplus \mathbf{m}_i \oplus \mathbf{c}_{i-1} \\ &= \mathbf{m}_i\end{aligned}$$

A graphical representation of CBC is shown in Figure 4.4.

Advantages and disadvantages

The main advantages of CBC are:

- A block isn't always encrypted to the same cypherblock. This is because the encryption of a block depends on the block but also on the previously encrypted blocks.
- When decrypting a ciphertext we only need the current ciphertext and the previous one (but we don't need the result of another decryption), hence the decryption process can be parallelised.

- A one-bit change to the ciphertext corrupts the corresponding plaintext block and inverts the corresponding bit in the next plaintext block. Further error propagation is avoided (CBC is self-recovering for intra-block errors).
- Insertions or deletions might be avoided since adding or removing a block might modify what comes after. This however depends on the position where the block is added or removed. If we add for instance a block at the end, then the change might not be detected.

The main disadvantages are:

- Different from decryption, encryption can't be parallelised.
- No recovery against synchronisation errors is possible. If a bit is inserted or added or lost from the ciphertext string, then all subsequent blocks are garbled.
- An adversary can alter a ciphertext block in such a way as to arbitrarily modify the following plaintext block. In particular, flipping a bit of the ciphertext flips the bit in the same position of the next decrypted plaintext. This however destroys the corresponding plaintext block.
- Reusing the same IV or key on two messages yields identical ciphertexts up to the first difference in plaintexts.

4.4.3 Counter mode

Counter mode (CTR) is one of the most used modes of operation. It is based, as the name suggests on a counter which is initialised with a value called initialisation vector (IV) which is randomly chosen and publicly disclosed. The counter is incremented for each block to encrypt and it's used to encrypt and decrypt the message or the ciphertext. More precisely, the counter is used as input of the encryption function $\mathbb{E}_k$ and the result is xored with the message to encrypt to obtain the ciphertext. Formally we can write,

$$\mathbf{c}_i = \mathbb{E}_k(IV + i) \oplus \mathbf{m}_i$$

Decryption happens in a similar way, in fact, the counter is passed to the decryption function $\mathbb{D}_k$ (which is the same as the encryption function, i.e., $\mathbb{D}_k = \mathbb{E}_k$) and the result is xored with the ciphertext to obtain the plaintext. Namely, we get

$$\mathbf{m}_i = \mathbb{E}_k(IV + i) \oplus \mathbf{c}_i$$

A graphical representation of the CTR mode is shown in Figure 4.5. It's important that the initialisation vector isn't reused in two different messages, otherwise the same block might be mapped to the same ciphertext block (hence obtaining the same result obtained for ECB). In this case it's a little bit harder to obtain the same result because the two blocks must have the same index i, however it's still is a vulnerability.

Advantages and disadvantages

The main advantages of CTR are:

- An error on one bit in the message (or in the ciphertext) effects only one bit in the ciphertext (or in the plaintext). This means that an error in one block is not propagated to successive blocks. This is because the i-the ciphertext block is computed using only the i-th message

block and the counter. This is an advantage for error recovery but also a disadvantage since it can be exploited by an attacker. More precisely, an attacker that is able to guess the encrypted message can change some bits in the ciphertext to obtain specific bits in the plaintext, when the ciphertext is decrypted.

- Only the encryption primitive is needed, namely $\mathbb{E}_k = \mathbb{D}_k$.
- Encryption and decryption can be parallelised since we don't have dependencies between blocks.
- We can encrypt or decrypt a block independently from the others since we simply have to compute the value of the counter.

The main disadvantages of CTR are:

- The initialisation vector can't be reused.
- An attacker can insert or delete blocks in specific positions (e.g., at the end) without being detected.

Message Authentication Codes and Galois Counter Mode

Even if the counter mode is very popular it sill doesn't provide message integrity (i.e., an attacker can change a message or a ciphertext without being noticed). To solve this problem we have to use Message Authentication Codes computed using symmetric keys and hash functions.

Another option for enforcing message integrity is using authenticated mode of operations like the **Galois Counter Mode** (GCM).

4.5 Stream-based mode of operations

A stream-based mode of operation handles a message $\mathbf{m}$ as a stream of blocks. In practice, the cypher generates a key stream $k_1, k_2, k_3, \ldots$ as long as the number of blocks to encrypt (or decrypt). The i-th ciphertext block is obtained xoring the i-th key k_i and the i-th plaintext block $\mathbf{m}_i$, namely

$$\mathbf{c}_i = k_i \oplus \mathbf{m}_i$$

The i-th ciphertext block is instead obtained xoring the i-th key k_i and the i-th ciphertext block $\mathbf{c}_i$, namely

$$\mathbf{m}_i = k_i \oplus \mathbf{c}_i$$

The different stream-based modes of operation define the way the key stream is generated. Each key k_i is as long as a block to encrypt or decrypt.

4.5.1 Cypher Feedback mode

A Cypher FeedBack mode (CFB) requires

- A n-bit encryption function $\mathbb{E}_k$.
- A n-bit shift register as input of $\mathbb{E}_k$.
- A n-bit shift register as output of $\mathbb{E}_k$.

To encrypt a message:

1. The input register is initialised with an initialisation vector.
2. The block (which has length n) is split into chunks of length j.
3. The bits in the input shift register are fed to the encryption function and the encrypted bits are stored in the output register. Now the encryption function isn't used until the whole block of n bits hasn't been encrypted.
4. The j leftmost bits of the output register are xored with one chunk of j bits of the message.
5. The input register is shifted to the left by j positions and the ciphertext generated, which also has length j, is xored with the input register (which means inserting the ciphertext bits in the input register).
6. Go back to step 4 using the next chunk of j message bits.

To decrypt a message:

1. The input register is initialised with an initialisation vector.
2. The block to decrypt (which has length n) is split into chunks of length j.
3. The bits in the input shift register are fed to the decryption function (which is the same as the encryption function) and the decrypted bits are stored in the output register. Now the encryption function isn't used until the whole block of n bits hasn't been encrypted.
4. The j leftmost bits of the output register are xored with one chunk of j bits of the ciphertext to obtain a chunk of j bits of the plaintext.
5. The input register is shifted to the left by j positions and the ciphertext is xored with the input register (which means inserting the ciphertext bits in the input register).
6. Go back to step 4 using the next chunk of j message bits.

Figure 4.6 shows how CFB works.

Advantages and disadvantages

The main advantages of CFB are:

- We use the encryption primitive for decryption, hence we need only one primitive.
- Decryption can be parallelised.
- The transmitted information comes in the form of arbitrarily size data
- Used with $j = 1$, a one-bit de-synchronisation is automatically recovered $n + 1$ positions after the inserted or deleted bit.
- Bit errors will corrupt the plaintext block at the same bit positions. The corrupted cypher block will then be fed to the input register and cause bit errors in the plaintext for as long as the erroneous bits stay in the input register. After that, the system recovers, and all following bytes are decrypted correctly.

The main disadvantages are:

- The self-recovery process is less efficient than other modes of operation in bringing back the system into functioning.
- It is subject to insertion or deletion attacks that systematically spoil the block synchronisation boundaries.

4.5.2 Output Feedback mode

The Output FeedBack mode (OFB) works similarly to CFB but the input register is fed back with the content of the output register instead of the generated ciphertext. Figure 4.7 shows how OFB works.

Advantages and disadvantages

The main advantages of OFB are:

- The transmitted information comes in the form of arbitrarily size data
- There is no need of a block cypher decryption primitive.
- Decryption can be parallelised.
- The encryption process can be partially parallelised as the values of output register can be pre-computed.
- In OFB, the bit errors in the decrypted ciphertext block (or segment) occur in the same bit positions as in the ciphertext block (or segment) and the other bit positions are not affected.

The main disadvantages of OFB are:

- It is subject to malicious bit insertion and deletion into the ciphertext, thus spoiling the synchronisation of the block (or segment) boundaries.

4.6 Substitution Permutation Networks

A Substitution Permutation Network is a framework for building an encryption (and a decryption) transformation, namely $\mathbb{E}_k$ and $\mathbb{D}_k$. An SPN applies the same set of operations (called round) multiple times. Each round of a SPN is split into three parts:

1. A **substitution** part. This stage applies a non-linear transformation to the cypher state. The transformation can either be seen as an algebraic function or a lookup table. The definitions are equivalent, however, using one definition for the hardware implementation might yield a faster circuit.
2. A **permutation** part. This stage applies a linear transformation that changes the positions of the bits in the cypher state. Instead of a bit-wise permutation, it is common for SPNs to perform a xor-linear mixing of the bits (e.g., via a permutation matrix).

3. A **key mixing** part. This stage mixes the cypher state with the key, usually applying a xor operation. If the key is added via a nonlinear operation (e.g., modulo addition) the cypher is defined as a product cypher.

Each part of the SPN works on the whole cypher state. Moreover, in an SPN, the encryption transformation is always different from the decryption transformation, namely

$$\mathbb{E}_k \neq \mathbb{D}_k$$

Note that:

- The substitution and key mixing stages ensure Shannon's confusion property (as defined in Theorem 3.3).
- The permutation stage ensures Shannon's diffusion property (as defined in Theorem 3.3).

4.6.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) cypher is one of the most used and secure symmetric encryption algorithms and it's based on an SPN. In particular, AES

- Encrypts blocks of 128 bits which are seen as a 4×4 matrix of bytes ($8 \cdot 4 \cdot 4 = 128$).
- Supports three different key lengths, namely 128, 192 and 256 bits.

AES's popularity comes from the fact that it

- has a simple round structure,
- is immune to linear and differential cryptanalysis (hence only bruteforcing is effective), and
- can be efficiently implemented in software and hardware.

Moreover, differently from DES (which is the protocol that AES was designed to replace), AES's design choices and inner mechanisms are well documented and explained.

An AES encryption round is made of the following steps:

1. The 128-bit input block is represented as a 4×4 byte matrix.
2. The round key is added to the state matrix via a **AddRoundKey** transformation.
3. A round is repeated for a number of times that depends on the length of the key (9 times for 128-bit keys, 11 times for 192-bit keys, 13 for 256-bit keys). Each round applies, in whatever order, four transformations:
   - A **SubBytes** transformation.
   - A **ShiftRows** transformation.
   - A **MixColumns** transformation.
   - A **AddRoundKey** transformation.
4. One round, different from the previous, is applied. This round applies the same transformations as the previous rounds, except for the MixColumns one. This is because said transformation doesn't add anything to the security margin of the cypher when applied in the last round.

A graphical representation of the encryption scheme is shown in Figure 4.8a. As we have seen, AES uses in its encryption and decryption module four transformations

- **SubBytes**.
- **ShiftRows**.
- **MixColumns**.
- **AddRoundKey**.

which take as input a state matrix (and optionally a key) and output another state matrix. Let us now further specify each of these transformations.

SubBytes

The SubBytes transformation is a substitution transformation, namely a non-linear bijective function which maps a block of 8 bits (i.e., a cell in the state matrix) into another block of 8 bits. Since every byte can be handled independently from the others, we can apply the transformation in any order. A SubBytes transformation can be implemented as a lookup table however we can also give an algebraic meaning to it. To understand what's the meaning behind this transformation let us write a byte (i.e., the content of a state matrix cell)

$$i = i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0$$

as a polynomial in the field $(\mathbb{F}_{2^8}, \oplus, \odot)$, namely a polynomial

$$I(x) = i_7 x^7 \oplus i_6 x^6 \oplus i_5 x^5 \oplus i_4 x^4 \oplus i_3 x^3 \oplus i_2 x^2 \oplus i_1 x^1 \oplus i_0 \mod (x^8 \oplus x^4 \oplus x^2 \oplus x \oplus 1)$$

where the bits are the coefficients of the polynomial. Note that the modulo operation is required since the multiplication of two polynomials of degree 7 is a polynomial of degree 14. However, we want to work with polynomials of degree 7 (because we want to work with bytes) hence we need the modulo operation with a polynomial of maximum degree 8 to get a polynomial of degree 7. Since we are working in a field, we can compute the inverse of a polynomial $I(x)$, which is the polynomial $I^{-1}(x)$ such that

$$I(x) \odot I^{-1}(x) = 1$$

and 1 is the identity of the field (i.e., of the $\odot$ operation). The multiplicative inverse can be interpreted back as a byte to obtain i^{-1}. Having obtained the inverse of the input byte, namely i^{-1} we can compute the output of the substitution transformation as

$$o = a \cdot i^{-1} \oplus b$$

where

- a is a constant 8×8-bit matrix defined by AES.
- b is a constant 8-bit vector defined by AES.
- $\cdot$ is the matrix multiplication.

This transformation is resistant to linear and differential cryptanalysis because

- it's hard to approximate the polynomial inversion operation with linear relations, and
- the equation for computing the output is hard to solve algebraically (because every output bit depends on all the bits of i^{-1}).

ShiftRows

The ShiftRows transformation rotates each row of the state matrix by a fixed number of positions. More precisely, row i (with the uppermost row being row 0) is shifted to the left by i positions. This transformation is fast to implement both in hardware (it's just a matter of crossing wires) and software (it can be implemented as a bit-wise rotation). By changing the positions of the bytes, this transformation is used to satisfy the diffusion property.

MixColumns

The MixColumns transformation combines the bytes of a column of the input (or cypher state) matrix to obtain a new column. More precisely, each column I_c is represented as a vector of four bytes, namely

$$I_c = (i_{0c}, i_{1c}, i_{2c}, i_{3c})$$

As always we can see a column as a polynomial

$$I_c(x) = i_{0c} + i_{1c}x + i_{2c}x^2 + i_{3c}x^3 \mod (x^4 + 1)$$

over the ring $(\mathbb{F}_{2^8}[x], +, \cdot)$ since each i_{xc} is a byte, hence the polynomial has coefficients ranging from 0 to $2^8 - 1$. The column is then multiplied by a fixed polynomial

$$C(x) = 2 + x^1 + x^2 + 3x^3$$

to obtain the new column

$$O_c(x) = C(x) \cdot I_c(x) \mod (x^4 + 1)$$

Note that the polynomial $C(x)$ has to be invertible if we want this transformation to be invertible. One can verify that in fact it exists $C^{-1}(x)$ such that

$$C(x) \cdot C^{-1}(x) \equiv 1 \mod (x^4 + 1)$$

and it is

$$C^{-1}(x) = 14 + 9x^1 + 13x^2 + 11x^3 \mod (x^4 + 1)$$

The MixColumns transformation can also be implemented on the ring $(\mathbb{F}_{2^8}, \oplus, \odot)$. In this case we write the bytes of a column $I_c = i_0 i_1 i_2 i_3$ as a vector

$$\mathbf{I}_c = \begin{pmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \end{pmatrix}$$

and the output bytes (i.e., the bytes that will be part of the new column) are computed as

$$\begin{pmatrix} o_0 \\ o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \end{pmatrix} = \begin{cases} 2i_0 \oplus 3i_1 \oplus 1i_2 \oplus 1i_3 \\ 1i_0 \oplus 2i_1 \oplus 3i_2 \oplus 1i_3 \\ 1i_0 \oplus 1i_1 \oplus 2i_2 \oplus 3i_3 \\ 3i_0 \oplus 1i_1 \oplus 1i_2 \oplus 2i_3 \end{cases}$$

Note that each value of the column is multiplied either by a power of 2 or by a number which is close to a power of 2 (i.e., can be obtained by adding or subtracting 1 to a power of 2), then this operations can be implemented using a shifter and a xor adder (which are both fast in hardware).

AddRoundKey

The AddRoundKey is a bit-wise xor between the bits in the matrix and the bits in the corresponding position of the round key (which is 128 bits long and can be interpreted as a 4×4 byte matrix).

The bit-wise xor is equivalent to the sum operation $\oplus$ on the group $\mathbb{F}_2$, namely the sum modulo 2. The bitwise xor is also an addition over $\mathbb{F}_{2^8}$. This means that the MixColumns and AddRoundKey transformations can be swapped since the MixColumns transformation can also be seen as an operation over $\mathbb{F}^{2^8}$, provided that $MixColumn^{-1}(K)$ is used as the key.

Decryption

The AES decryption round has the same structure of the AES encryption one but the data flows in the opposite direction and the inverse of the transformations previously defined is used, namely:

- **InvSubBytes** which applies the inverse of the SubBytes transformation.
- **InvShiftRows** which rotates the bytes in to the right.
- **InvMixColumns** which computes the same multiplication of the MixColumns transformation but using $C^{-1}(x)$ instead of $C(x)$.

A graphical representation of the decryption scheme is shown in Figure 4.8b.

Properties

The most important properties of AES are:

- One bit flip is diffused over the cypher state after just two rounds.
- The best-known cryptanalysis attack has a complexity of 2^{128} for AES-128, hence AES is completely immune to linear and differential cryptanalysis.
- The best known plaintext attack has a complexity of $2^{126.1}$ for AES-128, $2^{189.7}$ for AES-192 and $2^{254.4}$ for AES-256.
- The best-known plaintext, related key attack (i.e., the plaintext is encrypted using similar keys) has a complexity of 2^{176} for AES-192 and $2^{99.5}$ for AES-256. AES-128 is immune to this attack since the key is as large as the cypher state, thus there are no collisions of different key values onto the same intermediate state values.

T-tables

The transformations not involving the key can be fused together in a single transformation, usually implemented as a lookup table. More precisely, we obtain four lookup tables T_0, T_1, T_2 and T_3. The t-tables are computed as follows:

$$T_0[i] = \begin{bmatrix} S[i] \odot 02 \\ S[i] \\ S[i] \\ S[i] \odot 03 \end{bmatrix} \qquad T_1[i] = \begin{bmatrix} S[i] \odot 03 \\ S[i] \odot 02 \\ S[i] \\ S[i] \end{bmatrix} \qquad T_2[i] = \begin{bmatrix} S[i] \\ S[i] \odot 03 \\ S[i] \odot 02 \\ S[i] \end{bmatrix} \qquad T_3[i] = \begin{bmatrix} S[i] \\ S[i] \\ S[i] \odot 03 \\ S[i] \odot 02 \end{bmatrix}$$

The round is then computed column-wise (the input column is $I = (i0, i1, i2, i3)$ top to bottom in the state matrix) as

$$O = T_0[i_0] \oplus T_1[i_1] \oplus T_2[i_2] \oplus T_3[i_3] \oplus K \tag{4.4}$$

Key scheduling

The key schedule expands the key into $r+1$ keys, where r is the number of rounds. Before analysing the key schedule algorithm, let us notice that a key can be seen as a s words of 32 bits where

- $s = 4$ for AES-128.
- $s = 6$ for AES-192.
- $s = 8$ for AES-256.

In particular:

- The first keys are filled with the original user key.
- The rest of the key schedule K is filled column-wise, using the following algorithm:

  for $i = s, \ldots, 4(r+1) - 1$ **do**
  if $i \equiv 0 \mod s$ **then**
  $K[i] \leftarrow K[i-s] \oplus S[K[i-1] \lll 8]) \oplus RCON[\frac{i}{s}]$
  else if $s = 8 \wedge i \equiv 4 \mod s$ **then**
  $(K[i] \leftarrow K[i-s] \oplus S[K[i-1]])$
  else
  $K[i] \leftarrow K[i-s] \oplus K[i-1]$
  end if
  end for

  where RCON is an array of round-dependent 32-bit constants.

The key scheduling is invertible knowing at least s consecutive words of the key material (i.e., of the round keys).

4.7 Cryptanalysis of block cyphers

Cryptanalysis means building polynomial time tests to see if a block cypher outputs something which looks not random, hence breaking the cypher. As a result, cryptanalysis methods can be used to derive a key. The cryptanalysis of block cyphers can be used to understand the security of a protocol. In particular, it allows to test the robustness against known plaintext attacks, which are the hardest to defend against. In some cases, we can also consider related ciphertext attacks, namely attacks in which we can use two ciphertexts that differ by a single bit. This type of attack is weaker than known plaintext attacks but it still represents a good reference model. We can identify different techniques for doing cryptanalysis:

- **Algebraic** cryptanalysis.
- **Statistical** cryptanalysis. An important example of statistical cryptanalysis technique is **linear cryptanalysis**.
- **Differential** cryptanalysis.

In any case, cryptanalysis techniques aim at recovering the key with fewer operations than a brute-force attack. cryptanalysis is done considering known plaintext attacks and not ciphertext only attacks since if a cryptoscheme is secure against the former, then it's also secure against the latter. As a result, we will consider known-plaintext, related-plaintext (i.e., plaintext of which we control one bit) or chosen-plaintext attacks. The second type is weaker than controlling the whole plaintext.

4.7.1 Algebraic cryptanalysis

In algebraic cryptanalysis, a block cypher is represented as a vector of Boolean functions (or equations), one for each possible couple of plaintext-ciphertext, in the field $(\mathbb{Z}_2, \oplus, \wedge)$ where

- $\oplus$ is the xor operation.
- $\wedge$ is the and operation.

In this field, squaring a value returns the value itself since the square of a value is the and between the same value. If we check the truth table of the and operation we immediately notice that $0 \wedge 0 = 0$ and $1 \wedge 1 = 1$. The key bits are the equations' unknowns whereas the known terms are ciphertext and plaintext bits. Given enough equations, obtained from couples of ciphertext and plaintext, the goal is to solve the system of equations, hence to compute the key bits. These equations are huge, in fact, considering AES, each equation is made of $128 + 128 = 256$ variables, hence we can get 2^{256} possible equations since every variable can get either value 0 or 1. To efficiently perform algebraic cryptanalysis we have to split the process in different instances and repeat the cryptanalysis on a reduced set of equations for a small number of times.

Say a cypher uses only linear operations to obtain a ciphertext. This means that the cypher can be approximated with a system of linear equations and, since all equations are linear, it'd be easy to solve it and obtain the key. In particular, a completely linear cypher can be deterministically broken in polynomial time, i.e., $\mathcal{O}(\lambda^3)$ with λ number of key bits. For this reason, most block cyphers use s-boxes to introduce non-linearities in the cypher. We can easily check if a function in $(\mathbb{F}_2, \oplus, \wedge)$ is linear (the same is true for whatever ring) if

$$f(x \oplus y) = f(x) \oplus f(y) \quad \forall x \in \mathbb{F}_2, \forall y \in \mathbb{F}_2$$

This means that a function f is not linear if the statement above is false, hence if

$$\exists (x, y) \in (\mathbb{F}_2 \times \mathbb{F}_2) : f(x \oplus y) \neq f(x) \oplus f(y)$$

This result is very important since it tells us that a non-linear function might still show some linear behaviour, hence we might find values for which $f(x \oplus y) = f(x) \oplus f(y)$. Linear analysis exploits this fact to approximate a non-linear system as a linear one such that we can easily solve it. More precisely, we want to find a linear approximation of the non-linear part of the cypher which better approximates the behaviour of the true non-linear function. In other words, we want to find the linear approximation that behaves like the real function most of the times. The approximation we are seeking should be key-independent and should be able to

- Bypass all rounds of the cypher but one, i.e., the one for which the linear relation holds.
- Recover the last round key.

As we have understood, we want to approximate a non-linear S-box (which is publicly known and independent of the cypher itself) with a linear relation. A good choice is using linear equations since they are easy to manage.

Trivial cypher

Let us consider the trivial cypher to understand why a completely linear cypher can be broken in polynomial time by algebraic cryptanalysis. We consider an S-box with four input bits and four outputs as in Figure 4.9 bits to understand how linear cryptanalysis works, however the results we'll obtain hold for any S-box.

Algebraic approach Now that we know the structure of the S-box, we can write an equation for each output bit to obtain

$$\begin{cases} C_0 = P_2 \oplus k_2 \oplus k_1 \oplus k_0 \\ C_1 = P_3 \oplus k_3 \oplus k_0 \oplus k_1 \\ C_2 = P_0 \oplus k_0 \oplus k_3 \oplus k_2 \\ C_3 = P_1 \oplus k_1 \oplus k_2 \oplus k_3 \end{cases}$$

If we move the known parameters (i.e., ciphertext and plaintext) on the left-hand side of each equation we obtain

$$\begin{cases} P_2 \oplus C_0 = k_2 \oplus k_1 \oplus k_0 \\ P_3 \oplus C_1 = k_3 \oplus k_0 \oplus k_1 \\ P_0 \oplus C_2 = k_0 \oplus k_3 \oplus k_2 \\ P_1 \oplus C_3 = k_1 \oplus k_2 \oplus k_3 \end{cases}$$

We can now ask an oracle to encrypt a message $P_0P_1P_2P_3$, obtain $C_0C_1C_2C_3$ and use these values to solve the equation for k_0, k_1, k_2 and k_3.

Note that this can be applied also to purely linear feedback shift registers. As a result, a LFSR is broken if it doesn't introduce any non-linearity.

Statistical approach We have successfully broken this cypher using algebraic cryptanalysis but, just for the sake of showing it, we can do the same using a statistical approach. Let us consider the first equation which puts into correspondence the input P_2 with the output C_0,

$$C_0 = P_2 \oplus k_2 \oplus k_1 \oplus k_0$$

We can say that this equation holds with probability 1 since it's part of the cypher, hence it's always works and holds. Since C_0 and P_2 can only be 1 or 0, either the relation

$$r : C_0 = P_2$$

or the relation

$$r' : C_0 \neq P_2$$

holds. In particular, r holds if $C_0 \oplus P_2 = 0$, namely if $k_2 \oplus k_1 \oplus k_0 = 0$, which holds when

- $k_2 = k_1 = k_0 = 0$, or
- $k_0 = k_1 = 1$ and $k_2 = 0$, or
- $k_0 = k_2 = 1$ and $k_1 = 0$, or
- $k_1 = k_2 = 1$ and $k_0 = 0$.

The relation r' holds when $C_0 \oplus P_2 = 1$, hence if $k_2 \oplus k_1 \oplus k_0 = 1$, which holds when

- $k_2 = k_1 = k_0 = 1$, or
- $k_0 = k_1 = 0$ and $k_2 = 1$, or
- $k_0 = k_2 = 0$ and $k_1 = 1$, or
- $k_1 = k_2 = 0$ and $k_0 = 1$.

Let us consider the first relation. If we know that the first ciphertext's bit is the same as the third plaintext's bit, then we know that the key bits k_0, k_1 and k_2 can be in one of the four combinations we have listed. Since with three bits we can have $2^3 = 8$ possible combinations, we have basically halved the key space since we know that only the four listed keys out of the eighth combinations are possible. As a result, we have halved the key space just by looking at a single bit. The same result holds for the relation r'. Note that this is true since $C_0 = P_2 \oplus k_2 \oplus k_1 \oplus k_0$ holds with probability 1, hence either r or r' hold with probability 1 (but not together, they hold with probability one when conditioned on the observation of P_2 and C_0).

4.7.2 Statistical cryptanalysis

Statistical cryptanalysis aims at finding the likelihood that a bit of the key has a certain value. Equivalently, we want to find out the likelihood, given a ciphertext to have a plaintext under a certain key. Ideally, when building a block cypher, we want this probability to be 0.5, since it means that each bit is picked at random. An attacker wants instead to find a bias ε to add to or remove from the probability of each key-bit of having a certain value. Note that the bias can't be too small since we'd require too many samples to compute the error done when computing the probability. In practice, say $\varepsilon = \frac{1}{2^{3000}}$, then we'd require around 2^{3000} samples to estimate the error, however we don't have that many samples (AES uses 128 bit inputs, hence we have only 2^{128} possible inputs). Long story short, we can't apply this analysis if ε is too small, hence even a cypher is secure even if the probability of a key bit is not ideal but the bias it's very small.

Statistical cryptanalysis also allows to compare different couples of plaintext-ciphertext. We can take one plaintext m_1, flip one bit in the plaintext to obtain m_2, compute the output of m_1 and m_2 and then compute the change of the outputs. By looking the differences in output and conditioned by the probabilities taking place we want to understand what is the probability of a key-bit of having a certain value. In this case we ask ourselves what is the likelihood, given a bit-flip in input, to get a bit-flip in a specific place in output. If the cypher is a pseudo-random permutation, all the possible output bits should be looking alike. Basically, we are computing the probability of an output change given an input change, conditioned by the key value. The ideal value for this probability should be $\frac{1}{\textbf{block size}-1}$ where the -1 is justified by the fact that at least one bit should change. This means that we should ideally have one output change.

4.7.3 Linear cryptanalysis

One important example of statistical cryptanalysis is linear cryptanalysis. Let us apply this technique to a cypher called *simple cypher* to better understand what linear cryptanalysis is. Statistical cryptanalysis, hence linear cryptanalysis, is easy on purely linear cyphers. For this reasons, cyphers have to include non-linear transformations that make these techniques harder. In particular, non-linearities prevent efficient closed form solutions of the equations we write to model the cypher. If we remember the statistical cryptanalysis we did on the trivial cypher, adding non-linearities means that the probability of a relation between a ciphertext and a plaintext bit, conditioned by an observation of the plaintext and the ciphertext bits, is smaller than 1. Since we can't approximate the whole cypher in one shot (i.e., on all rounds), we have to

1. exploit some biases to provide a key independent linear approximation of a portion of the cypher,

2. use the approximation in place of all but one rounds of the cypher,

3. use the approximation to recover the last round key since, having the relation between an input (approximated with the linear relations) and its output of a round, a single round is broken, and

4. repeat removing the last round.

For this procedure to be successful, we have to find a bypass (i.e., the set of linear relations) from the plaintext to the input of the last round which holds with probability more than $\frac{1}{2}$ and is key independent. Also note that peeling away a round diminishes the strength of the cypher. As a result, the key part is getting a good cryptanalysis on the last round and the next steps are easier.

The easiest way of finding the best linear relation (i.e., equation) that approximates the considered S-box is by enumerating all possible equations and take the one which makes fewer mistakes. In general we have that a relation between a plaintext $m = x_0...x_{n-1}$ and a ciphertext $c = y_0 \dots y_{n-1}$ can be written as

$$\bigoplus_i x_i = \bigoplus_i y_i$$

Since we are working in $\mathbb{F}_2$, it holds $-1 =_2 1$ hence we can write

$$(\bigoplus_i x_i) \oplus (\bigoplus_i y_i) = c$$

with $c \in \mathbb{F}_2$. Let us consider, for now on, an S-Box with 4 inputs and 4 outputs. This means that we have $2^4 \cdot 2^4 \cdot 2 = 2^9$ possible combinations, hence 2^9 possible tentative equations. Note that we can do an assumption over the value of c and compute the equation considering only ciphertext and plaintext. In our example, after having assumed $c = 0$, we obtain

$$a_0x_0 \oplus a_1x_1 \oplus a_2x_2 \oplus a_3x_3 = b_0y_0 \oplus b_1y_1 \oplus b_2y_2 \oplus b_3y_3 \quad a_i, b_i \in \{0, 1\}$$

Since we are not sure that $c = 0$, we can say that this holds with some probability $\frac{1}{2} + \epsilon$. As a result, $c = 1$ holds with probability $\frac{1}{2} - \epsilon$. We will consider $c = 0$ just because it's easier to manipulate the equation, however the same reasoning can applied if we assume $c = 1$. Now we can consider every possible assignment of the values a_i, b_i and evaluate how well such equation approximates the S-Box. Given a model (i.e., an assignment for the values a_1, a_2, a_3, a_4 and b_1, b_2, b_3, b_4), we

1. pick one input $[x_0, x_1, x_2, x_3]$,

2. give an input $[x_0, x_1, x_2, x_3]$ to the S-Box and compute the output $[y_0, y_1, y_2, y_3]$,

3. check if the equation $a_0x_0 \oplus a_1x_1 \oplus a_2x_2 \oplus a_3x_3 = b_0y_0 \oplus b_1y_1 \oplus b_2y_2 \oplus b_3y_3$ holds using the outputs of the S-Box,

4. increment a counter *ctr* if the equation holds,

5. if we have checked each of the 16 possible inputs, stop, otherwise pick a new input and go back to step 2.

If we repeat this analysis for every equation and then keep the equation that does fewer mistakes (i.e., with the highest *ctr*) we obtain the best approximation for the S-Box. In other words, we have chosen the best model that approximates the S-Box, i.e., the best values of a_i, b_i by counting how many times the relation holds with respect to all possible (m, c) pairs. Finally, we want to identify the bias (with respect to the 0.5 probability), which is computed as

$$\varepsilon = \frac{ctr - 8}{16}$$

where *ctr* counts how many times the relation holds (i.e., how many outputs are correctly computed by the linear model). Note that, since the relation $0 = 0$ always holds, the bias is in this case always $\varepsilon = \frac{1}{2}$, but we are not interested in this case. Ideally, we would like to have a table (which contains the bias for all possible combinations of ciphertext and message) filled with all 0s, however even a table filled with 0s and 2s is still a good enough solution.

To show how linear cryptanalysis can be applied to an S-Box, let us consider an S-Box derived from DES. The S-Box we consider has four input bits and 4 output bits. The S-Box is shown in Table 4.1.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

Table 4.1: A 4×4 S-Box. To use this S-Box we take the column which has in the first raw the symbol in input $(x_0x_1x_2x_3)$. The output is the value in the second row and in the same column.

The results of the evaluation of the equations can be shown in a table, called bias table, which contains

- an input equation, i.e., $a_0x_0 \oplus a_1x_1 \oplus a_2x_2 \oplus a_3x_3$, for each row, and
- an output equation, i.e., $b_0y_0 \oplus b_1y_1 \oplus b_2y_2 \oplus b_3y_3$, for each column.

Namely, each row defines a value for a_is and each column a value for b_is. A cell in the bias table contains the value $ctr - \frac{\text{box size}}{2}$, where the box size, in our example is 16 (since we are considering a 4×4 S-Box). The bias of a model can be computed as

$$\epsilon = \frac{v}{\text{SBox size}} = \frac{ctr - \frac{\text{SBox size}}{2}}{\text{SBox size}}$$

where v is the value of the cell corresponding to that model. Table 4.2 is the bias table of the S-Box in Table 4.1.

Note that the bias table contains only even values because the xor operation is symmetrical. For the same reason, the values v and $-v$ represent the same result. By looking at the bias table (4.2) we can see that the best models are those with value 6 or -6 (because the 8 is obtained for the model $0 = 0$, which has no meaning). One of the best models has $a_0a_1a_2a_3 = 8 = 1000$ and $b_0b_1b_2b_3 = F = 1111$, hence it's the model

$$x_0 = y_0 \oplus y_1 \oplus y_2 \oplus y_3$$

Simple cypher

Let's now focus on a more complex cypher than the trivial cypher. This cypher takes a 16 bit input and a 80-bit key (so that it's bruteforce resistant) and computes the ciphertext using 4 rounds without key scheduling (16 bits of the key are used in each round). The simple cypher is built using a SPN design (i.e., it's a substitution permutation network) with:

- A round key addition performed through bit-wise xor.
- Four identical S-Boxes with four input bits each. Each S-Box is the same as the one in Table 4.1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	-2	-2	0	0	-2	6	2	2	0	0	2	2	0	0
2	0	0	-2	-2	0	0	-2	-2	0	0	2	2	0	0	-6	2
3	0	0	0	0	0	0	0	0	2	-6	-2	-2	2	2	-2	-2
4	0	2	0	-2	-2	-4	-2	0	0	-2	0	2	2	-4	2	0
5	0	-2	-2	0	-2	0	4	2	-2	0	-4	2	0	-2	-2	0
6	0	2	-2	4	2	0	0	2	0	-2	2	4	-2	0	0	-2
7	0	-2	0	2	2	-4	2	0	-2	0	2	0	4	2	0	2
8	0	0	0	0	0	0	0	0	-2	2	2	-2	2	-2	-2	6
9	0	0	-2	-2	0	0	-2	-2	-4	0	-2	2	0	4	2	-2
A	0	4	-2	2	-4	0	2	-2	2	2	0	0	2	2	0	0
B	0	4	0	-4	4	0	4	0	0	0	0	0	0	0	0	0
C	0	-2	4	-2	-2	0	2	0	2	0	2	4	0	2	0	-2
D	0	2	2	0	-2	4	0	2	-4	-2	2	0	2	0	0	2
E	0	2	2	0	-2	-4	0	2	-2	0	0	-2	-4	2	-2	0
F	0	-2	-4	-2	-2	0	2	0	0	-2	4	-2	-2	0	2	0

Table 4.2: The bias table of the S-Box in Table 4.1.

- A permutation performed putting the i-th bit of the j-th S-Box into the j-th bit of the i-th Sbox.

The cypher has the following structure

1. One round which applies round key addiction, the S-Boxes and a permutation.
2. One round which applies round key addiction, the S-Boxes and a permutation.
3. One round which applies round key addiction, the S-Boxes and a permutation.
4. One round which applies round key addiction and the S-Boxes.
5. One final round key addiction.

Figure 4.10 shows the simple cypher that we have just described.

To model the whole cypher we need two things:

- Compute the bias of more than one S-Box. Namely, after finding an approximation for one round and the approximation of another round we have to define a way to sum up the biases of the two rounds.
- Deal with the effects of the intermediate key addictions.

Let us consider the first problem, first. The solution to this problem is given by the pile-up lemma. Let's start by stating the pile-up lemma for one bit.

Lemma 4.1 (Pile-up (bit)). *Let Z_1 and Z_2 be two random variables over $\{0,1\}$ with distri-*

butions

$$Pr(Z_1 = z_1) = \begin{cases} p_1 & \text{if } z_1 = 0 \\ 1 - p_1 & \text{if } z_1 = 1 \end{cases}$$

and

$$Pr(Z_2 = z_2) = \begin{cases} p_2 & \text{if } z_2 = 0 \\ 1 - p_1 & \text{if } z_2 = 1 \end{cases}$$

Assuming that Z_1 and Z_2 are independent, we can write the probability of the joint distribution

$$Pr(Z_1 = z_1, Z_2 = z_2) = \begin{cases} p_1 p_2 & \text{if } z_1 = 0, z_2 = 0 \\ (1 - p_1)p_2 & \text{if } z_1 = 1, z_2 = 0 \\ p_1(1 - p_2) & \text{if } z_1 = 0, z_2 = 1 \\ (1 - p_1)(1 - p_2) & \text{if } z_1 = 1, z_2 = 1 \end{cases}$$

Then the probability of $Z_1 \oplus Z_2 = 0$ is

$$\begin{aligned} Pr(Z_1 \oplus Z_2 = 0) &= Pr(Z_1 = 0, Z_2 = 0) + Pr(Z_1 = 1, Z_2 = 1) \\ &= p_1 p_2 + (1 - p_1)(1 - p_2) \end{aligned}$$

From a cryptanalysis point of view we have merged two Boolean relations and given their probabilities we have computed the probability of the result of the relation. Since we want to compute biases, we can rewrite the probabilities p_1 and p_2 as

$$p_1 = \frac{1}{2} + \epsilon_1$$

and

$$p_2 = \frac{1}{2} + \epsilon_2$$

with

$$-\frac{1}{2} \leq \epsilon_1, \epsilon_2 \leq \frac{1}{2}$$

The probability of $Z_1 \oplus Z_2 = 0$ can be rewritten as

$$\begin{aligned} Pr(Z_1 \oplus Z_2 = 0) &= Pr(Z_1 = 0, Z_2 = 0) + Pr(Z_1 = 1, Z_2 = 1) \\ &= p_1 p_2 + (1 - p_1)(1 - p_2) \\ &= p_1 p_2 + 1 - p_1 - p_2 + p_1 p_2 \\ &= 2p_1 p_2 - p_1 - p_2 + 1 \\ &= 2p_1 p_2 - \frac{1}{2} - \epsilon_1 - \frac{1}{2} - \epsilon_2 + 1 \\ &= 2\left(\frac{1}{2} + \epsilon_1\right)\left(\frac{1}{2} + \epsilon_2\right) - \epsilon_1 - \epsilon_2 \\ &= 2\left(\frac{1}{4} + \frac{1}{2}\epsilon_1 + \frac{1}{2}\epsilon_2 + \epsilon_1\epsilon_2\right) - \epsilon_1 - \epsilon_2 \\ &= \frac{1}{2} + \epsilon_1 + \epsilon_2 + 2\epsilon_1\epsilon_2 - \epsilon_1 - \epsilon_2 \\ &= \frac{1}{2} + 2\epsilon_1\epsilon_2 \end{aligned}$$

This formula gives us a way to compute the bias of a relation between two bits. Note that as always we are not interested in the $\frac{1}{2}$ in the probability but only in the bias $2\epsilon_1\epsilon_2$. Let us call this bias $\epsilon_{1,2}$, namely

$$\epsilon_{1,2} = 2\epsilon_1\epsilon_2 \tag{4.5}$$

The lemma can be generalised to an arbitrary number of random variables.

Lemma 4.2 (Pile-up)**.** *Let $Z_1, Z_2, \ldots, Z_n$ be independent random variables with values in $\{0,1\}$. The probability of the relation $Z_1 \oplus Z_2 \oplus \ldots \oplus Z_n = 0$ is*

$$Pr(Z_1 \oplus Z_2 \oplus \ldots \oplus Z_n = 0) = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \epsilon_i$$

The bias in this formula can be denoted with

$$\epsilon_{1,\ldots,n} = 2^{n-1} \prod_{i=1}^{n} \epsilon_i$$

Note that we are still assuming that all bits are independent which actually, in practice, doesn't hold but we usually neglect this fact. If the variables are not independent, the lemma doesn't hold anymore. Consider for instance, Z_1, Z_2 and Z_3 with biases $\epsilon_1 = \epsilon_2 = \epsilon_3 = \frac{1}{4}$. If we compute the biases $\epsilon_{1,2}$, $\epsilon_{2,3}$ and $\epsilon_{1,3}$ we get $\epsilon_{1,2} = \epsilon_{2,3} = \epsilon_{1,3} = \frac{1}{8}$. The variables $Z_{12} = Z_1 \oplus Z_2$ and $Z_{23} = Z_2 \oplus Z_3$ can't be independent, in fact, if they were we could write

$$\begin{aligned} Z_{12} &= Z_1 \oplus Z_3 \\ &= Z_1 \oplus (Z_2 \oplus Z_2) \oplus Z_3 \qquad\qquad a \oplus a = 0, \quad a \oplus 0 = a \\ &= (Z_1 \oplus Z_2) \oplus (Z_2 \oplus Z_3) \\ &= Z_{12} \oplus Z_{23} \end{aligned}$$

and the bias would be $\epsilon_{1,3} = 2 \cdot \frac{1}{8} \cdot \frac{1}{8} = \frac{1}{32}$, which is not the case since we have computed $\epsilon_{1,3} = \frac{1}{8}$.

Let us now focus on the main idea of linear cryptanalysis. We want to model a cypher using linear relations. This is because they are useful when we combine relations which describe portions of the cypher. If we use linear relation we can keep the number of variables to a decent number (i.e., with which we can work). Let us consider a cypher, similar to the simple cypher, to understand how we can achieve this result. We consider an S-Box, the following permutation layer, a key addition and another S-Box. Figure 4.11 shows this setup.

We can now consider different models for each component and, if we consider the permutation and the key addition as a single component we get three relations:

- $r_1 : X_3 \oplus Y_1 \oplus Y_2 \oplus Y_3 = 0$ for the first S-Box.
- $r_2 : (Y_3' \oplus Y_3 \oplus K_3) \oplus (Y_1' \oplus Y_1 \oplus K_1) \oplus (Y_2' \oplus Y_2 \oplus K_2) = 0$ for the permutation and key addition layer.
- $r_3 : Y_1' \oplus Y_2' \oplus Y_3' \oplus Z_0 \oplus Z_2 = 0$ for the second S-Box.

Note that we had to choose one model for each S-Box (ideally the best one we obtained after computing the bias table) but the model for the middle part is fixed since the permutation is fixed

and linear. In other words, the second relation is always true, hence we have a bias of $\frac{1}{2}$. Note that the choice of the second S-Box model is biased on the first S-Box model. In fact, if we know that the first model has only Y_3, Y_2 and Y_1 with coefficient 1, then the inputs of the second S-Box must be coherent, hence only Y_1', Y_2' and Y_3' must have coefficient equal to 1. Once we have defined a set of inputs, we can take the best model (i.e., the one with highest bias) with respect to such inputs (namely, we consider only one row of the bias table, the one with the last three inputs equal to 1, and we take the column with the highest bias). If we compute the biases for the relations above using the pile-up lemma, we get

- $\epsilon_1 = \frac{6}{16}$
- $\epsilon_2 = \frac{1}{2}$
- $\epsilon_3 = -\frac{4}{16}$

If we add the second and third two relations member-wise we can cancel out the common terms since $X \oplus X = 0$, hence we obtain

$$\begin{aligned} Y_1' \oplus Y_2' \oplus Y_3' \oplus Z_0 \oplus Z_2 \oplus Y_3' \oplus Y_3 \oplus K_3 \oplus Y_1' \oplus Y_1 \oplus K_1 \oplus Y_2' \oplus Y_2 \oplus K_2 = 0 \\ (Y_1' \oplus Y_1') \oplus (Y_2' \oplus Y_2') \oplus (Y_3' \oplus Y_3') \oplus Z_0 \oplus Z_2 \oplus Y_3 \oplus K_3 \oplus Y_1 \oplus K_1 \oplus Y_2 \oplus K_2 = 0 \\ Z_0 \oplus Z_2 \oplus Y_3 \oplus Y_2 \oplus Y_1 \oplus K_3 \oplus K_1 \oplus K_2 = 0 \end{aligned}$$

We can now compute the bias of this new relation ϵ_{23} using the pile-up lemma as

$$\epsilon_{23} = 2\epsilon_2\epsilon_3 = -2\frac{1}{2}\frac{4}{16} = -\frac{4}{16}$$

The same process can be repeated to sum up this relation we have obtained an r_1. What we get is

$$\begin{aligned} Z_0 \oplus Z_2 \oplus Y_3 \oplus Y_2 \oplus Y_1 \oplus K_3 \oplus K_1 \oplus K_2 = X_3 \oplus Y_1 \oplus Y_2 \oplus Y_3 \\ Z_0 \oplus Z_2 \oplus (Y_3 \oplus Y_3) \oplus (Y_2 \oplus Y_2) \oplus (Y_1 \oplus Y_1) \oplus K_3 \oplus K_1 \oplus K_2 \oplus X_3 = 0 \\ Z_0 \oplus Z_2 \oplus K_3 \oplus K_1 \oplus K_2 \oplus X_3 = 0 \end{aligned}$$

The whole cypher can therefore be modelled with the relation

$$r_{123} : Z_0 \oplus Z_2 \oplus X_0 = K_3 \oplus K_1 \oplus K_2$$

As always, we can apply the pile-up lemma to compute the bias of this relation as

$$\epsilon_{123} = |2\epsilon_1\epsilon_{23}| = 2\frac{6}{16}\frac{4}{16} = \frac{3}{16}$$

Now we are able to say that if someone gives us one input bit X_3, we know that the output bits Z_0 and Z_2 will be equal to the sum of the key bits and X_3, with probability $\frac{1}{2} + \frac{3}{16}$. Namely, this equation will hold 11 times out of 16. Note that the order in which we sum each part up doesn't matter.

Breaking the simple cypher In the example above we have considered a small cypher. Now we want to repeat the same type of analysis on a bigger cypher as the simple cypher. What we are trying to achieve is **whole cypher approximation**, namely, starting from the plaintext bits, we:

1. Pick the plaintext bits that have a better (i.e., higher) bias for the first model.
2. Draw a path to the last-but-one round. The path is built considering what bits are set to 1 in the previous output. This means that the inputs of a component have to match the outputs of the component before. As a result, we choose for each non-linear component the best model considering the output bits of the component before. Also note that when we do whole cypher approximation, we might find layers with multiple S-Boxes, hence we have to pile-up more biases.
3. Stop right before the last round.

Now that we have everything we need to analyse the simple cypher, it's finally time to practically break it. To better understand the models used to approximate the cypher let us call:

- $U_{i,j}$ the j-th input bit (from 1 to 16, from bottom to top) of the i-th S-Box layer (considering that the S-Boxes used in one round are identical).
- $V_{i,j}$ the j-th output bit (from 1 to 16, from bottom to top) of the i-th S-Box layer (considering that the S-Boxes used in one round are identical).
- $K_{i,j}$ the j-th bit (from 1 to 16, from bottom to top) of the i-th round key.

Remember that in our example the S-Boxes are all the same and their bias table is shown in Table 4.2. Let us consider for instance the S-Box in position $(B, 4)$, namely the one modelled by

$$r_1 : X_0 \oplus X_2 \oplus X_3 = Y_1$$

This model hasn't the highest bias, however it has the advantage of having just one output. This is an advantage since one output bit in a permutation layer will not diffuse, hence it will not influence more than an S-Box in one round. As a result, we only have to pick one S-Box model for the next rounds instead of multiple one. This is particularly useful in early rounds. The bias of this model can be computed through the bias table as

$$\epsilon_{r_1} = \left| \frac{4-8}{16} \right| = \frac{1}{4}$$

We also have to model the first key addition round, which is simply modelled as

$$U_{1,j} = P_j \oplus K_{1,j} \quad \forall j \in \{1, \ldots, 16\}$$

where P_j is the j-th plaintext bit. Note that the output of the key addition round is represented as U because it's the input of the S-Box This relation has bias $\frac{1}{2}$ since it's always true. If we rewrite r_1 using our notation we obtain

$$r_1 : V_{1,4(5-b)-3} \oplus U_{1,4(5-b)-1} \oplus U_{1,4(5-b)} = V_{1,4(5-b)-2}$$

With b number of S-Box in a layer (from 1 to 4, from top to bottom). For instance, if we consider the third S-Box we get

$$r_1 : U_{1,5} \oplus U_{1,7} \oplus U_{1,8} = V_{1,6}$$

If we sum the addition layer and the S-Box we get the relation

$$\begin{aligned} V_{1,6} &= U_{1,5} \oplus U_{1,7} \oplus U_{1,8} \\ V_{1,6} &= (P_5 \oplus K_{1,5}) \oplus (P_7 \oplus K_{1,7}) \oplus (P_8 \oplus K_{1,8}) \end{aligned}$$

which has the same bias $\epsilon_1 = \frac{1}{4}$ since the relation of the key addition round is always true. We have basically solved the first round. Now we can follow the wires exiting from the third S-Box of the first round to the second key addition and the second S-Box. Once again we can model the key addition round as

$$U_{2,a} = V_{1,b} \oplus K_{2,j}$$

Note that in this case we are also including the permutation layer, hence the output bit in one position might go to another position. For instance, if we consider the first S-Box (the one on top), we get

$$U_{2,7} = V_{1,10} \oplus K_{2,j}$$

Since we are considering the third S-Box, we get

$$U_{2,6} = V_{1,6} \oplus K_{2,6}$$

This means that the output of the first layer's third S-Box goes into the second layer's third S-Box. We should then find an approximation for the third S-Box of the second layer. Since only the second input is on (i.e., only X_1), we have to choose the best model from Table 4.2 considering row 4, namely the one for $X_1 = \ldots$. One of the relations with more bias is the one in position $(4, 5)$, which corresponds to

$$X_1 = Y_1 \oplus Y_3$$

and has bias

$$\epsilon_2 = -\frac{1}{4}$$

If we translate the relation r_2 applied on the third S-Box of the second layer in our notation we get

$$U_{2,6} = V_{2,6} \oplus V_{2,8}$$

We can now apply the pile-up lemma and merge the approximation of the key addition and S-Box layers to obtain

$$\begin{aligned} U_{2,6} &= V_{2,6} \oplus V_{2,8} \\ V_{1,6} \oplus K_{2,6} &= V_{2,6} \oplus V_{2,8} \end{aligned}$$

Having obtained a model for the first and second layer we can once again apply the pile-up lemma and merge the two layers to obtain the relation r_{12} as

$$\begin{aligned} 0 &= V_{1,6} \oplus K_{2,6} \oplus V_{2,6} \oplus V_{2,8} \oplus V_{1,6} \oplus (P_5 \oplus K_{1,5}) \oplus (P_7 \oplus K_{1,7}) \oplus (P_8 \oplus K_{1,8}) \\ 0 &= (V_{1,6} \oplus V_{1,6}) \oplus K_{2,6} \oplus V_{2,6} \oplus V_{2,8} \oplus P_5 \oplus K_{1,5} \oplus P_7 \oplus K_{1,7} \oplus P_8 \oplus K_{1,8} \\ 0 &= K_{2,6} \oplus V_{2,6} \oplus V_{2,8} \oplus P_5 \oplus K_{1,5} \oplus P_7 \oplus K_{1,7} \oplus P_8 \oplus K_{1,8} \end{aligned}$$

As a result, the first two rounds are modelled by the relation

$$r_{12} : K_{2,6} \oplus P_5 \oplus K_{1,5} \oplus P_7 \oplus K_{1,7} \oplus P_8 \oplus K_{1,8} = V_{2,6} \oplus V_{2,8}$$

The bias of this relation can be computed by applying the pile-up lemma. What we get is

$$\epsilon_{12} = 2\epsilon_1\epsilon_2 = 2\frac{1}{4}\left(-\frac{1}{4}\right) = -\frac{1}{8}$$

Note that the relation r_{12} depends only on the input bits, the key and the outputs of the last layer we have analysed, but not on the inner variables. This means that we have removed two linear layers and we know that this approximation is true more than half of the time. Now we have two output bits which are on and these bits are the inputs of different S-Boxes, in particular, the first and the third S-Boxes of the third layer (or round). This means that we have to model two different S-Boxes. Let's start with the first one. The only input bit which is on is the second, this means that we have to look for a model which looks like

$$X_1 = \cdots$$

By inspecting Table 4.2 we notice that we can take, as we did for the previous layer, relation $(4, 5)$, which corresponds to

$$X_1 = Y_1 \oplus Y_3$$

and has bias

$$\epsilon_3 = -\frac{1}{4}$$

Translating this relation in our cypher we obtain

$$r_3 : U_{3,14} = V_{3,16} \oplus V_{3,14}$$

Moreover, we also have to model the key addition round, which is represented, in the case of the first S-Box, as

$$V_{2,8} \oplus K_{3,14} = U_{3,14}$$

If we put these two relations together, we obtain the model for the key addition layer and the S-Box, which is

$$r_3 : U_{3,14} \oplus V_{3,16} \oplus V_{3,14} \oplus V_{2,8} \oplus K_{3,14} \oplus U_{3,14} = 0$$

By simplifying the term we obtain

$$r_{3,1} : V_{3,16} \oplus V_{3,14} \oplus V_{2,8} \oplus K_{3,14} = 0$$

If we do the same operation for the third S-Box (choosing the same model for the S-Box as before), we get the relation

$$r_{3,3} : V_{3,6} \oplus V_{3,8} \oplus V_{2,6} \oplus K_{3,6} = 0$$

As for the previous model, $r_{3,3}$ has a bias of

$$\epsilon_3 = -\frac{1}{4}$$

These approximations must be true at the same time, hence this time we are piling up two biases. In practice, if we pile-up the relations $r_{3,1}$ and $r_{3,3}$ we get

$$r_3 : V_{3,6} \oplus V_{3,8} \oplus V_{2,6} \oplus K_{3,6} \oplus V_{3,16} \oplus V_{3,14} \oplus V_{2,8} \oplus K_{3,14} = 0$$

Applying the pile-up lemma, we obtain an overall bias of

$$\begin{aligned} \epsilon'_3 &= 2\epsilon_3\epsilon_3 \\ &= 2\left(-\frac{1}{4}\right)\left(-\frac{1}{4}\right) \\ &= \frac{1}{8} \end{aligned}$$

Now we can pile up the relation r_3 modelling the third layer with the relation r_{12} modelling the layers before to obtain the relation r_{123} as

$$r_{123} : r_{12} \oplus r_3 = 0$$

If we simplify common terms we get

$$r_{123} : V_{3,6} \oplus V_{3,8} \oplus V_{3,16} \oplus V_{3,14} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{1,8} \oplus K_{3,6} \oplus K_{3,14} \oplus K_{2,6} \oplus K_{1,5} \oplus K_{1,7} = 0$$

As always we can compute the bias of the new relation using the pile-up lemma which returns

$$\begin{aligned} \epsilon_{123} &= 2\epsilon_{12}\epsilon'_3 \\ &= 2\left(-\frac{1}{8}\right)\frac{1}{8} \\ &= -\frac{1}{32} \end{aligned}$$

Finally, we have reached the last round of S-Boxes. As before we have to consider two S-Boxes, in particular, the first and the third, since the outputs of the first and third S-Boxes of the third round are mixed up by the permutation layer. Let's consider the key addition round first. If we model each key addition with the usual formula, we obtain

$$\begin{cases} U_{4,16} = V_{3,16} \oplus K_{4,16} \\ U_{4,14} = V_{3,8} \oplus K_{4,14} \\ U_{4,8} = V_{3,14} \oplus K_{4,8} \\ U_{4,6} = V_{3,6} \oplus K_{4,6} \end{cases}$$

If we replace these relations in r_{123} we get

$$r'_{123} : (U_{4,6} \oplus K_{4,6}) \oplus (U_{4,14} \oplus K_{4,14}) \oplus (U_{4,16} \oplus K_{4,16}) \oplus (U_{4,8} \oplus K_{4,8}) \oplus P_5 \oplus P_7 \oplus P_8 \oplus \sum_K = 0$$

where $\sum_K$ is the xor of the key bits, hence

$$\sum_K = \oplus K_{1,8} \oplus K_{3,6} \oplus K_{3,14} \oplus K_{2,6} \oplus K_{1,5} \oplus K_{1,7}$$

The same relation can be rewritten by adding all the key bits in $\sum_K$ to obtain

$$r'_{123} : U_{4,6} \oplus U_{4,14} \oplus U_{4,16} \oplus U_{4,8} \oplus P_5 \oplus P_7 \oplus P_8 \oplus \sum_K = 0$$

where

$$\sum_K = \oplus K_{1,8} \oplus K_{3,6} \oplus K_{3,14} \oplus K_{2,6} \oplus K_{1,5} \oplus K_{1,7} \oplus K_{4,14} \oplus K_{4,16} \oplus K_{4,8} \oplus K_{4,6}$$

This relation holds with the same bias as the relation r_{123} since the relations of the key addition round are always true. We can therefore write

$$\epsilon'_{123} = -\frac{1}{32}$$

Since our goal is to obtain a relation which is independent of the key, we would like to remove the key bits from r'_{123}. To achieve this result, we have to remember that we are playing a fair game, hence the defender (i.e., the actor which is encrypting some message) isn't changing the key (which is reasonable to assume since messages are encrypted using a session key which is used for many messages). This means that the key is fixed, namely it's a constant, even if we don't know it. If the key bits are constant, then their sum is also constant. Since a sum of bits can be either 1 or 0, we can say that the sum of the key bits is 0 with the bias ϵ'_{123} and it's equal to 1 with bias $-\epsilon'_{123}$. But since we are interested in the magnitude of the bias and not in its sign, then we can take one of the two values and use it. We'll use $\sum_K = 0$ since it's easier to work with. As a result, we can write the relation

$$r''_{123} : U_{4,6} \oplus U_{4,14} \oplus U_{4,16} \oplus (U_{4,8} \oplus P_5 \oplus P_7 \oplus P_8 = 0$$

which holds with bias

$$\epsilon'_{123} = -\frac{1}{32}$$

This intuition allows us to forget the key bits altogether and the key stops having an influence on the model. We only have to remember to test for plus and minus the bias.

Finally, we have a model of what's before the last layer of S-Boxes which doesn't depend on the key. This means that we can imagine that the portion of the cypher before the last layer doesn't exist anymore. We can use the model r'_{123} to obtain the last round-key (i.e., the fifth round-key) without even approximating the last S-Boxes. In particular, we want to exploit the fact that, in our example, given three input bits we know the sum of the output bits more often than we should (i.e., with a bias bigger than 0). In practice, we can just guess what is needed to get from the ciphertext (which is known) to the input of the last layer's S-Boxes. This means guessing the key bits of the last round (i.e., the fifth). More precisely, we have to:

1. Guess the key bits of the fifth round key. In our case we only have to guess the key bits $K_{5,16}, \ldots, K_{5,13}$ and $K_{5,5}, \ldots, K_{5,8}$ since only the first and third S-Boxes have non-zero inputs.

2. Obtain the input of the last (i.e., fifth) key addition round by applying the guess to the ciphertext.

3. Compute the inverse of the S-Boxes, which is possible since we have the S-Box's output bits (because they have been computed in the previous step) and the S-Box implementation is known. This computation returns the inputs to the S-Boxes of the fourth round.

If the key guess is correct and we have a lot of ciphertexts, the relation between the inputs obtained through this process and the one obtained by applying r'_{123} to the input bits should be holding true with the probability of $\frac{1}{2} \pm \epsilon'_{123}$. The assumption, which usually holds in practice, is that if we get the key guess wrong, this correspondence doesn't hold with that probability. To be more precise, the procedure for guessing the last round-key works as follows:

1. Take the ciphertext, which is known.

2. Call partial subkeys the key bits of the last round-key (i.e., the fifth round-key) influenced by the output bits of our approximation of the cypher.

3. Collect $N \geq \frac{1}{\epsilon^2}$ plaintext, ciphertext pairs. Note that we need more than $\frac{1}{\epsilon^2}$ couples and not $\frac{1}{\epsilon}$ because we are estimating a bias, not an error. We would need the latter in case we were estimating the error.

4. Assume that the partial subkey is made of L bits. For each partial subkey value, $sk_i \in \{0, 1, \ldots, 2^{L-1}\}$, count how many plaintext-ciphertext pairs make the relation r'_{123} of the last round hold. Let us call N_{sk_i} this number. To understand if $r_{123'}$ holds, we have to use the ciphertexts to feed the inverse of the S-Boxes activated on the last layer and compute $U_{4,j}$.

5. Compute the biases $N_{sk_i} - 1$ for each sk_i value.

6. Select the partial subkey value with the highest bias.

Figure 4.12 shows the complete analysis we did.

Now we simply have to repeat the whole process using different input bits until we are able to recover every key bit. Once we have the last round-key, we can remove the last round and repeat the whole process again. Note that the more layers we remove, the higher will be the bias since we have fewer layers and each layer reduces the bias.

As a result, since removing rounds makes cryptanalysis easier, we can add rounds to make a cypher more secure. This however comes at the cost of efficiency.

4.7.4 Differential cryptanalysis

The idea behind differential cryptanalysis is the same as the one behind linear cryptanalysis. In fact, we can build a relation to model a part of the cypher which holds with some bias. This time, instead of looking at the values, we look at the changes. In particular, we change a key bit in the input and we try to understand which bit should change in an intermediate state. If the cypher were ideal, a one-bit change in the input should return a randomly changed output, hence the probability of seeing any of the possible changes would be $\frac{1}{2^{block\ size}}$. In formulas, we can write that the probability of seeing a difference in the output d_{out} knowing that a certain difference in input d_{in} has happened should be $\frac{1}{2^n}$, namely

$$Pr(d_{out}|d_{in}) = \frac{1}{2^n}$$

Differential cryptanalysis can be done using the same technique we described for linear cryptanalysis but using differences instead of bit values. In particular,

- we use the xor to merge relations between differences,
- we can remove duplicate differences since $A \oplus A = 0$,
- chain variables using the xor operation.

Before describing differential cryptanalysis in detail, let us define the notation we will use. Let

$$X' = [X'_0, X'_1, X'_2, X'_3]$$

and

$$X'' = [X''_0, X''_1, X''_2, X''_3]$$

be two states. The difference between X' and X'' is written as

$$\Delta X = [\Delta X_0, \Delta X_1, \Delta X_2, \Delta X_3]$$

where ΔX_i is the bit-wise xor between X'_i and X''_i. A couple of input and output differences $(\Delta X, \Delta Y)$ is called **differential**. In an ideal cypher, any differential appears with a probability of

$\frac{1}{2^n}$, hence in a non-ideal cypher this probability is not $\frac{1}{2^n}$. In particular, we will consider the garden variety of differential cryptanalysis for which we have that

$$Pr(\Delta Y = b|\Delta X = a) > \frac{1}{2^n}$$

Another type of differential cryptanalysis only looks for differentials for which

$$Pr(\Delta Y = b|\Delta X = a) = 0$$

hence for which we don't see a bit flip pattern when we change something in the input. This type of cryptanalysis is called impossible cryptanalysis.

Linear transformations

Before considering non-linear layers, let us apply differential cryptanalysis to linear transformations. In linear transformations, a difference in the input is propagated to the output unchanged. This means that the differential probability is 1 since given a certain input we know what output bits will change. Permutations just change what bits are involved in the differential but not the probability of a differential. Key bits aren't a problem. If we have a given difference in input to an add-key round, we have the same difference in output since $\Delta X = X \oplus K$ and $\Delta Y = K \oplus Y$. If we sum these differentials we get $X \oplus K \oplus K \oplus Y = X \oplus Y$ and the key vanishes. As a result, we can forget about key additions since keys would be removed anyways unless we are adding keys with something which is not an xor.

Non-linear transformations

When considering non-linear transformations like S-Boxes, we have to find what is the effect of an S-Box on a linear difference. This can be done exhaustively since S-Boxes are small, hence we can try every possible difference in input and look at every possible difference in output and count how many times they happen. This time we have to build a table of probabilities that a given bit-flip in input gives me a given bit-flip in output, instead of biases. In practice, considering an S-Box with four inputs, we have to:

1. Choose a difference in input ΔX.
2. Choose a difference in the output ΔY.
3. Initialise a counter $ctr \leftarrow 0$.
4. Pick all the possible values in input $I = (i_0, i_1, i_2, i_3)$:
   (a) Compute the output values passing I through the S-Box.
   (b) Apply the difference $I \oplus \Delta X$ and compute the output values passing $I \oplus \Delta X$ through the S-Box.
   (c) If the difference of $SBox(I)$ and $SBox(I \oplus \Delta X)$ is ΔY, increment the counter ctr.
5. Store the differential probability as $\frac{ctr}{16}$.
6. Repeat from step 1 until all possible differentials have been considered.

As a result, we obtain a table similar to the bias table but containing probabilities. If we consider the S-Box in Table 4.3 (which is the same as Table 4.1 which we used for the simple cypher) and we apply the process above, we obtain the Table 4.4. Let us understand, as we did for linear cryptanalysis, how to understand what relation is best. Different from linear cryptanalysis, zeros are a problem since we can use them for impossible cryptanalysis. Twos aren't a problem since if we have two values that give us a difference and we swap the roles they give us the same difference, hence the least we can get is 2 in this table. Everything above 2 is a problem and the closer it is to 16, the bigger the problem. The value 8 in position $(B, 2)$ is a big problem for the cypher since it means that if we flip the bits X_0, X_2 and X_3 in input we get a flip of bit Y_2 in output half of the time. Also, remember that the values of the first row and column are useless since they refer to no change in input or no change in output.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

Table 4.3: A 4×4 S-Box. To use this S-Box we take the column which has in the first raw the symbol in input $(x_0x_1x_2x_3)$. The output is the value in the second row and in the same column.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
A	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
B	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
C	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
D	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
E	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
F	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

Table 4.4: The probability table of the S-Box in Table 4.3. Each row represents a difference in input ΔX and each column represents a difference in output ΔY.

Once we have obtained the probability table, the process is exactly the same as we described for linear cryptanalysis. This time, since we want to compute the probability of a sequence of relations, we can simply multiply each probability (under the assumption that the random variables used to model the SBoxes are independent), namely

$$p_{1,\dots,n} = \prod_{i=1}^{n} p_i$$

After approximating the cypher before the last layer we get in the output of the approximation some differences which hold with a probability greater than $\frac{1}{2}$. Now we can

1. guess one portion of the key,
2. invert the ciphertext by applying the guessed key to the ciphertext,
3. apply the inverted S-Box to the ciphertext xored with the guessed key for getting the difference in input to the S-Box,
4. compare the results obtained from the inverted S-Box to the differences obtained by passing the plaintext to the approximation and store in a variable $diff$ how many times the latter differences are the same as the former ones,
5. repeat from step 1 using a different key guess until all keys have been considered, and
6. keep the key for which the differential approximation (i.e., the approximation of the part before the last round) holds exactly with the predicted probability.

Note that in this case, we can't use any plaintext-ciphertext couple but we have to consider plaintexts with a specific difference between the plaintext bits. This means that we are considering a related plaintext attack instead of a known plaintext attack which means that the attacker model is stronger in the case of differential cryptanalysis.

The whole procedure for recovering the last round key can be summed up as follows:

1. Collect a significant number of plaintext-ciphertext pairs with known differences among the plaintexts.
2. For each plaintext-ciphertext pair:
   (a) Initialise *count* to zero.
   (b) For each possible value of the partial sub-keys:
      i. Invert the effect of the partial sub-keys on the affected ciphertext bits.
      ii. Invert the last S-Box and check if the differential approximation holds.
      iii. If the approximation holds, increment *count*.
3. Check which partial sub-key makes the differential approximation hold exactly with the predicted probability.
4. Lather, rinse and repeat for all the key bits you need to extract.

Also remember that since this time we are approximating a probability $Pr() = p$, we need $\frac{1}{p}$ samples. For this reason, it looks like differential cryptanalysis is stronger than linear cryptanalysis (since we need fewer samples) but we also need related inputs. As a result, the two techniques usually achieve similar results.

4.7.5 Cyphers design criteria

Now that we know how to attack cyphers, we can define some guidelines to design good cyphers. We've seen that a cypher should follow the confusion and diffusion principle. Now we have a quantitively way of establishing these properties, in fact:

- Confusion means that the bias of a given input-output pair should be none.
- Diffusion considers more bits, namely can be seen as the number of S-Boxes affected by an input.

Another important part of a cypher is key addition. We have always considered xor addition but this isn't the only solution and we have also seen that using a non-xor addition makes differential cryptanalysis harder. An alternative to xor addition is modulo 2^n addition for some $n > 1$. Note that the addition modulo 2^n is not linear modulo 2. In fact, if we consider two bits in input (p_1, p_0) and two key bits (k_1, k_0), the ciphertext is computed as

$$(p_1 \oplus k_1 \oplus (p_0 \wedge k_0), \quad p_0 \oplus k_0)$$

where $p_0 \wedge k_0$ is the carry and as we can see it's computed using a non-linear operation.

To establish the security of a cypher we can use **reduced round cryptanalysis**. Say we know that it's impossible to apply a cryptanalysis technique to a cypher because it has too many rounds. We can remove some rounds and apply the same techniques to the reduced cypher. This gives us an idea of how the difference between the reduced and full cypher strengthens the cypher. For instance, if we know that a cypher is broken at 6 rounds (e.g., AES for related key attacks), we can say that

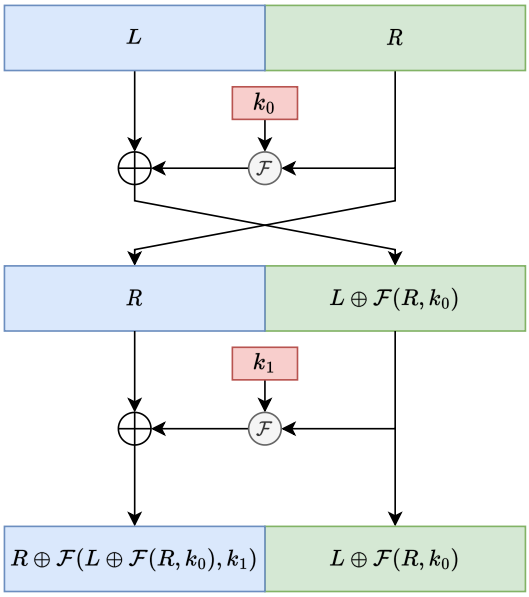
- if the full cypher has 7 rounds, we might break it in the future, but
- if the full cypher has 20 rounds, we can consider it secure.

DES

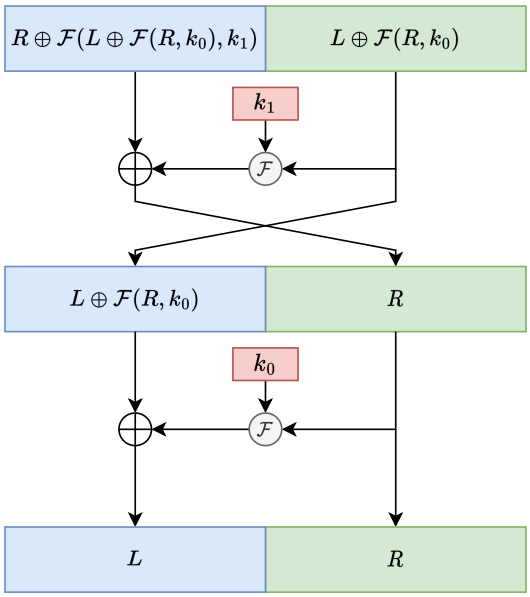
DES is broken under linear and differential cryptanalysis. Currently, the best differential cryptanalysis requires 2^{47} plaintext-ciphertext couples for the whole cypher (which is made of 16 rounds). The best linear cryptanalysis requires 2^{43} couples for the whole cypher (which is made of 16 rounds). These attacks are relevant but not catastrophic since an exhaustive search is 2^{56}. Note that triple DES doesn't change these attacks which are still possible with the same number of samples since the inner DES is not shielded.

AES

AES has been designed after linear and differential cryptanalysis, hence it has been designed to be resistant to these techniques. In particular, the biases are so small that we would require more material than we actually can feed to the cypher to apply linear or differential cryptanalysis. Moreover, AES's S-Boxes are 4-uniform, which means that by looking at the differential probabilities table we see only 2s, 0s and one 4. Moreover, the propagation path of the boolean expression of the diffusion layer, it's done so that activating a box will always activate four more in the next round. This means that even if we have a small bias, the simultaneous events we have to consider increase rapidly, which makes the analysis much harder.

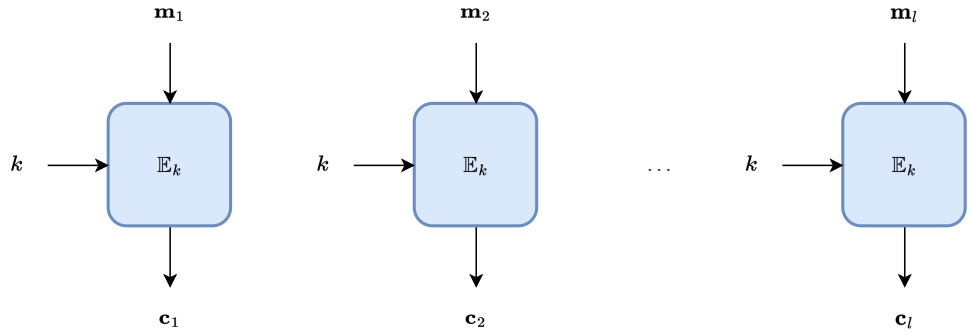


(a) An example of Feistel network used for encryption.

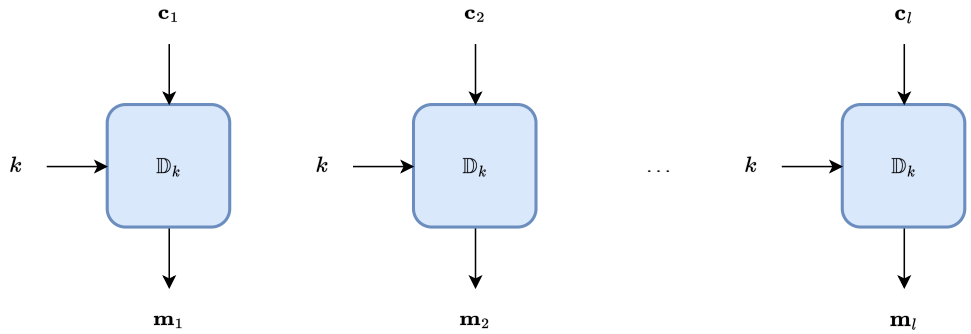


(b) An example of Feistel network used for decryption.

Figure 4.2: An example of Feistel network.

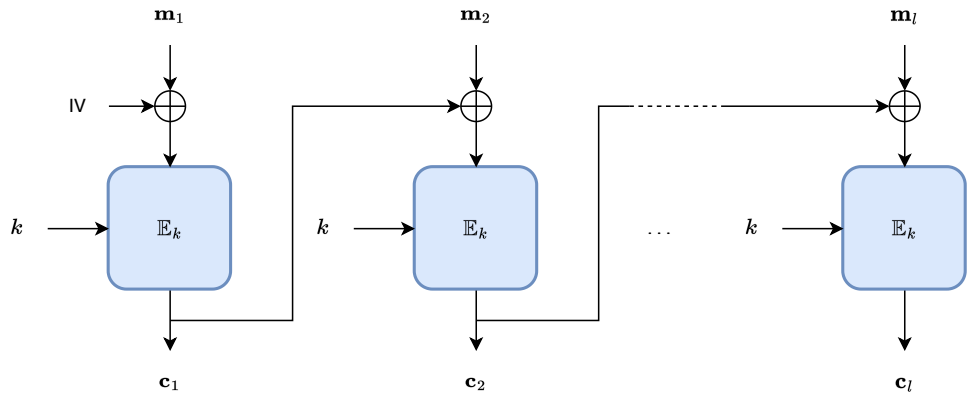


(a) An Electronic Code Book for encryption.

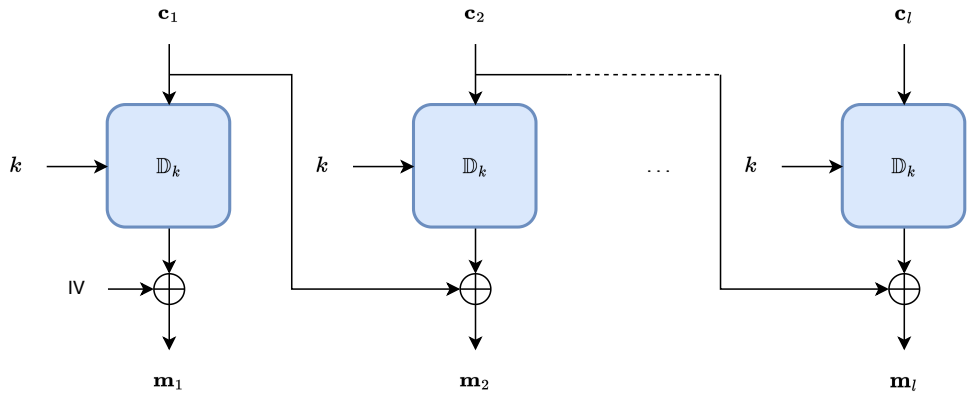


(b) An Electronic Code Book for decryption.

Figure 4.3: An Electronic Code Book.

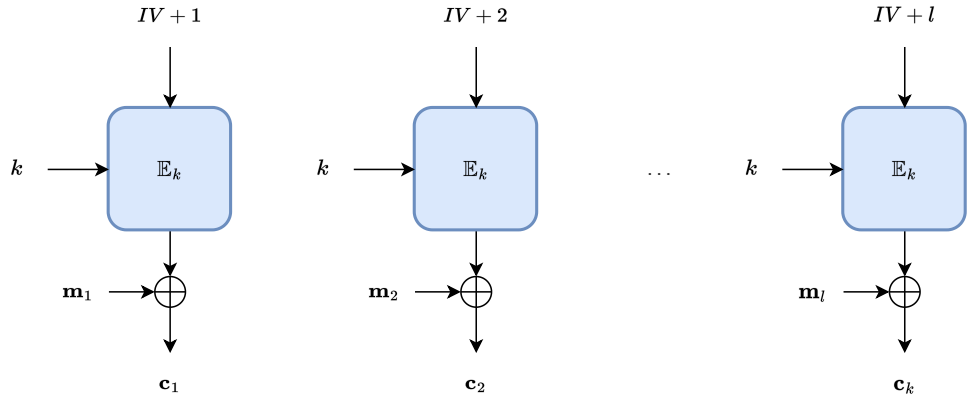


(a) A Control Block Chaining for encryption.

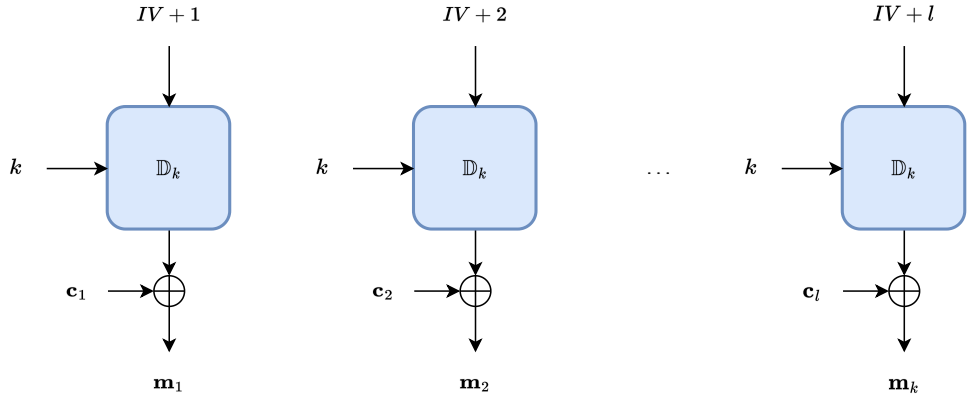


(b) A Control Block Chaining for decryption.

Figure 4.4: A Control Block Chaining.

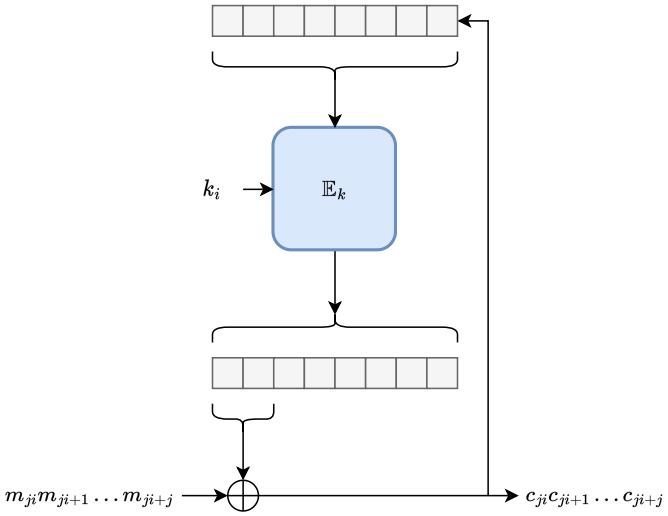


(a) A counter mode used for encryption.

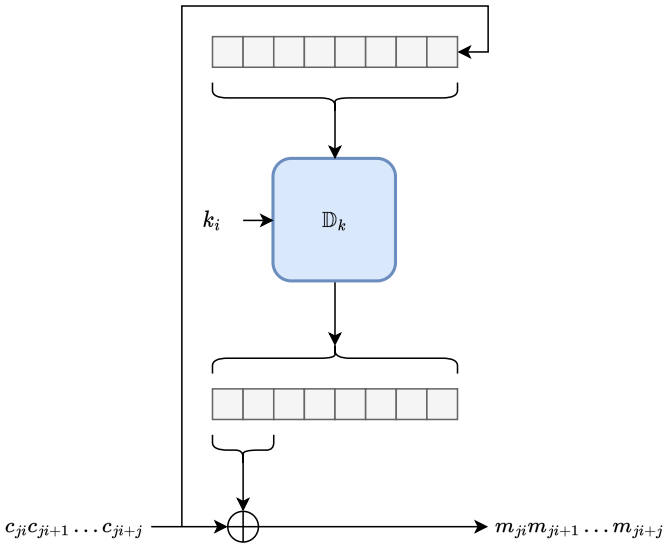


(b) A counter mode used for decryption.

Figure 4.5: A counter mode.

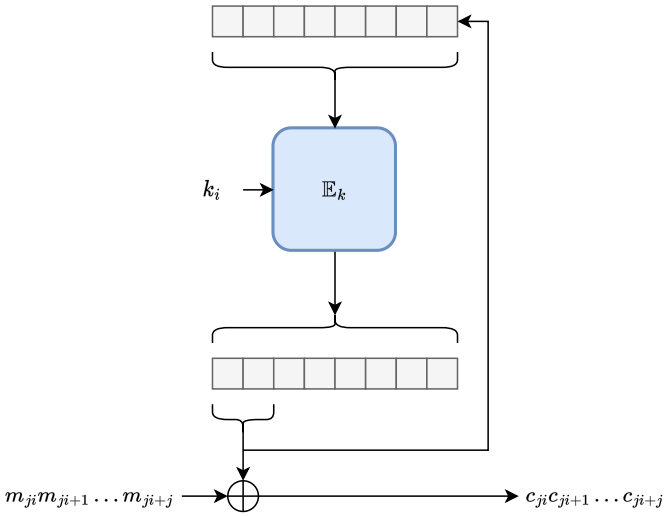


(a) Cypher FeedBack mode for encryption.

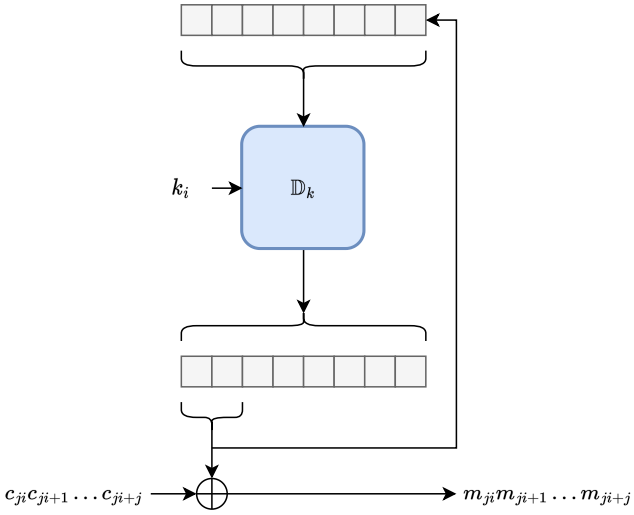


(b) Cypher FeedBack mode for decryption.

Figure 4.6: Cypher FeedBack mode.

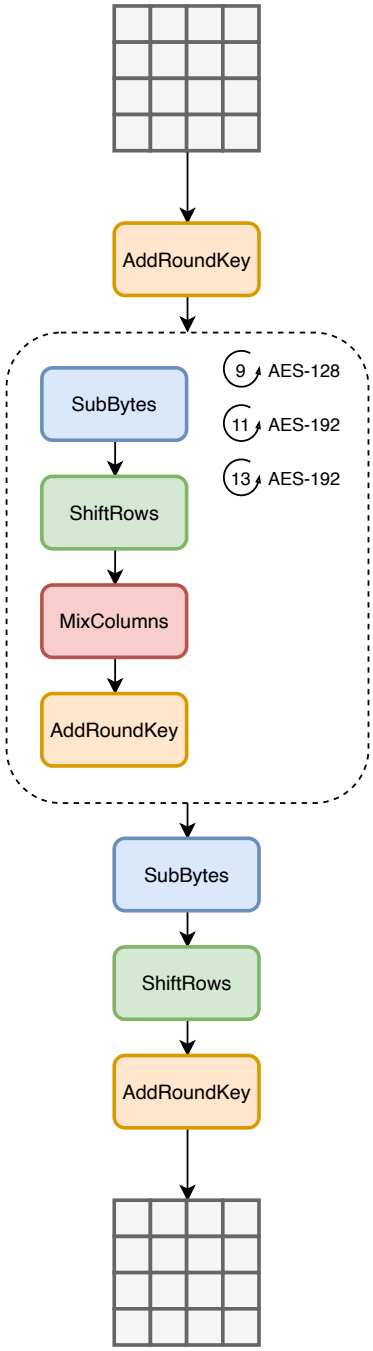


(a) Output FeedBack mode for encryption.

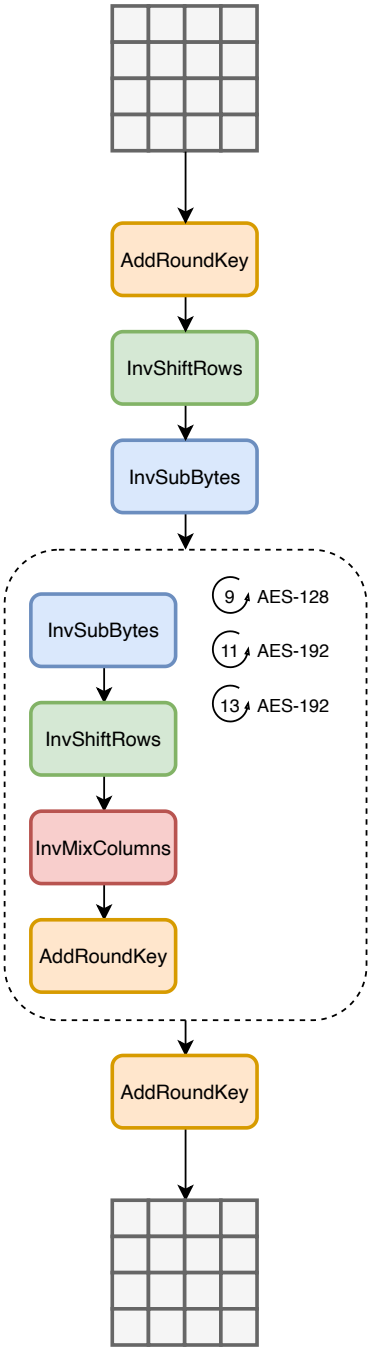


(b) Output FeedBack mode for decryption.

Figure 4.7: Output FeedBack mode.



(a) The Advanced Encryption Standard scheme used for encryption.



(b) The Advanced Encryption Standard scheme used for decryption.

Figure 4.8: The Advanced Encryption Standard scheme

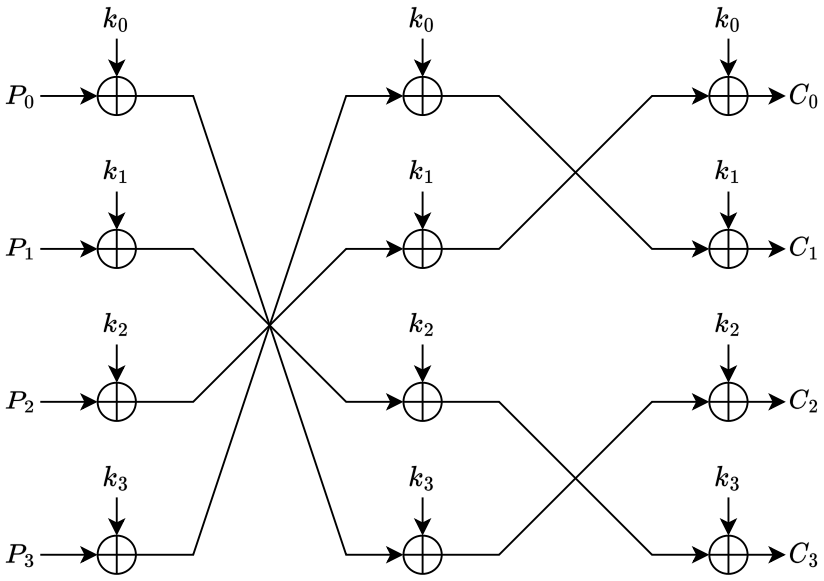


Figure 4.9: The trivial cypher

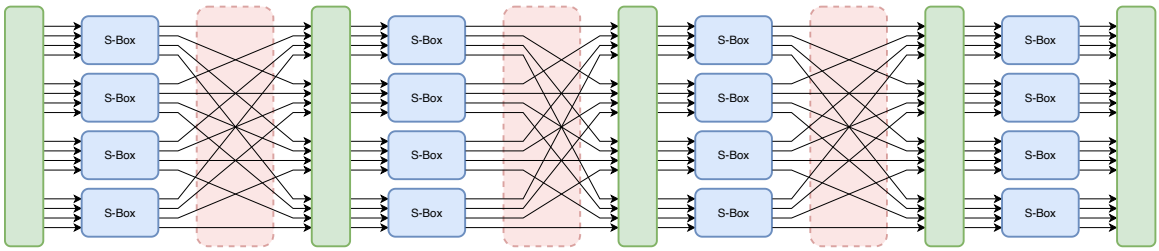


Figure 4.10: The simple cypher.

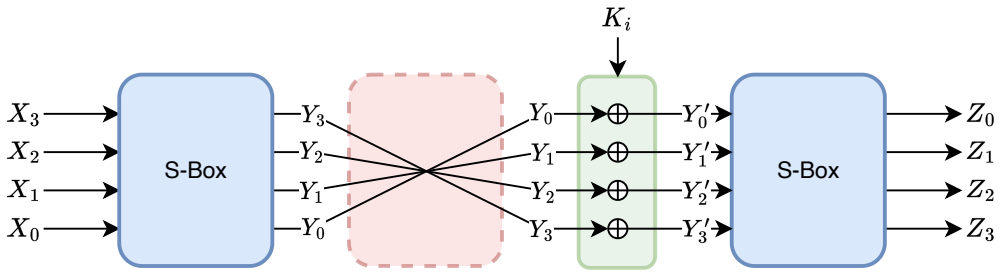


Figure 4.11: The cypher used for explaining how to use the pile-up lemma.

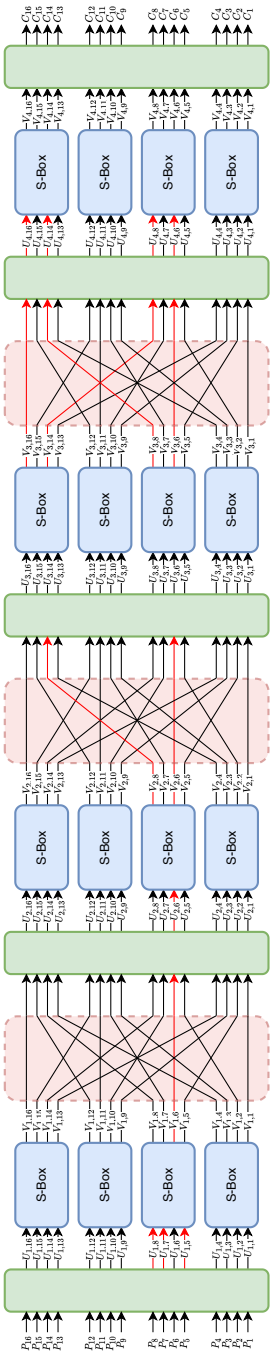


Figure 4.12: The path followed by the example in the simple cypher.

Chapter 5

Stream cyphers

5.1 General structure

Stream cyphers, unlike block cyphers, work on a sequence of bits of undefined length. The key, which is in fact a keystream (i.e., a sequence of bits of undefined length), is generated one bit at a time. The generated bit is combined (usually xored but other options are also available) with the next bit to encrypt (or decrypt) to obtain the next ciphertext bit (or plaintext bit). More schematically:

1. One bit of the plaintext stream (ciphertext stream) has to be encrypted (decrypted).
2. One bit of the keystream is generated.
3. The bit to be encrypted (decrypted) is xored with the keystream bit.

Since stream cyphers work on streams of bits, the message to be encrypted (or decrypted) doesn't need any padding. A stream cypher mimics a One Time Pad (as we have seen when discussing perfect cyphers). The only difference between an OTP and a stream cypher is that the key produced by a stream cypher has a periodicity, hence after a while, we will see the same pattern (or sequence) seen in the past. Formally, each bit of the plaintext is encrypted as

$$c_i = k_i \oplus m_i$$

and each ciphertext is decrypted as

$$m_i = k_i \oplus c_i$$

where k_i is a bit of the keystream.

Stream cyphers were usually adopted in multimedia applications like music or movie streaming because they needed very limited hardware resources and had limited memory requirements. Nowadays block cyphers have become so cheap that even such applications use them instead of stream cyphers.

Note that, if we consider communication between two parties, the keystreams of the sender (who encrypts the message) and receiver (who decrypts the message) have to be synchronised, in fact, they have to use the same sequence of key bits to encrypt and decrypt the message. To achieve this goal, a keystream is generated using L memory elements which are initialised with a secret key $k \in \mathcal{K}$ known only by the sender and receiver and then used to generate the keystream.

Stream cyphers can be divided into two categories:

- **Synchronous stream cyphers**.
- **Asynchronous stream cyphers**.

5.1.1 Synchronous stream cyphers

In synchronous stream cyphers the keystream is generated as a function of the cypher key k and of the memory elements $s_0, s_1, \ldots, s_L$, independently of any previous plaintext or ciphertext digit, i.e.

$$k_i = f(k, s_0, s_1, ..., s_L)$$

This ensures that when a bit is erroneously flipped, the error affects only a bit of the ciphertext and isn't propagated to every successive bit of the ciphertext. On the flip side, synchronisation is crucial for a correct encryption and decryption since we can't use the input to fix any error. Moreover, because bit flips aren't propagated, an active adversary can use insertion, deletion or substitution (replay) of ciphertext digits to get predictable changes on the deciphered plaintext.

5.1.2 Asynchronous stream cyphers

In asynchronous stream cyphers, the keystream is generated as a function of the cypher key k and a finite number of previous ciphertexts. Namely, given a key $k \in \mathcal{K}$ and an initial state $S_0 = (s_{L-1}, \ldots, s_0)$ the keystream is obtained as

$$k_i = f(k, S_i, S_{i-1}, ...)$$

with

$$S_i = (c_{i+L-1}, c_{i+L-2}, \ldots, c_{i+1}, c_i)$$

Since asynchronous stream cyphers use the ciphertext to generate the key, then an erroneous digit in the input affects at most L digits of the output.

The decrypting endpoint synchronises after receiving L ciphertext digits. This means that it's easier to recover if digits are dropped or added to the ciphertext stream. For this reason, asynchronous stream cyphers are also called self-synchronising cyphers.

As for synchronous stream cyphers, an active adversary can use insertion, deletion or substitution (replay) of ciphertext bits to get predictable changes on the decrypted plaintext.

5.2 Linear feedback shift registers

A Linear Feedback Shift Register (LFSR) is a keystream generator implemented as a circuit with L states that, at each clock cycle, are linearly combined to generate the next bit state. More precisely, the circuit is made of L states $s_t, s_{t+1}, \ldots, s_{t+L-1}$ and, at each clock cycle:

1. The bit s_t is used to encrypt or decrypt the next plaintext or ciphertext bit.
2. A linear combination of the state bits $s_t, s_{t+1}, \ldots, s_{t+L-1}$ is computed to obtain s_{t+L}, namely

$$s_{t+L} = c_L s_t \oplus c_{L-1} s_{t+1} \oplus \cdots \oplus c_1 s_{t+L-1} = \sum_{i=0}^{L-1} c_{L-i} s_{t+i}$$

   Since s_{t+L} is computed as a linear combination, we need some coefficients that multiply the values of the states. In this case, the coefficients c_i are binary coefficients and are xored with

the corresponding state bit, hence when the coefficient is 0, the corresponding state bit is not used for computing the next state s_{t+L}. Note that the coefficient c_L must always be 1, otherwise, we could use a generator with one bit less since s_t wouldn't be used for generating s_{t+L}.

3. Every state bit is shifted to the right (i.e., s_t is removed from the memory and s_L enters the memory).

Formally, the state bit to be inserted in the cypher state is updated using the recurrence relations

$$s_L = \sum_{i=1}^{L} c_i \cdot s_{L-i}$$

$$s_{t+L} = \sum_{i=1}^{L} c_i \cdot s_{t+L-i} \quad t \geq 0$$

where s_{t+L} is the bit that will shifted into s_{t+L-1}. A graphical representation of a LFSR is shown in Figure 5.1. A linear feedback shift register has many good properties:

- It's easy to implement in hardware since it requires only a few bits of memory, some and gates and some xor gates.
- It can be demonstrated that it generates a keystream with long period. Note that, since we have L states, the maximum period of this generator is $2^L - 1$ (since when the state is made of all zeros, then it won't change).
- It generates a keystream with good statistical properties (i.e., it's pseudo-random).
- We can use algebra to analyse it.

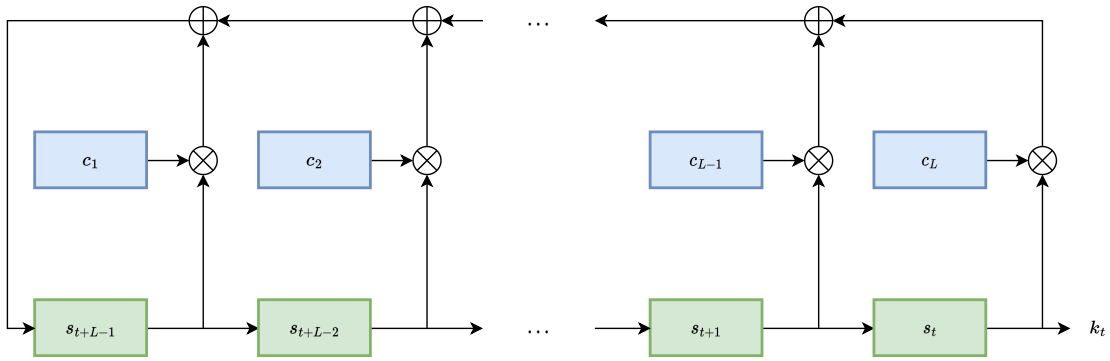


Figure 5.1: A Linear Feedback Shift Register.

Since an LFSR has binary cells, we can interpret each cell as a coefficient of a polynomial of degree $L-1$ in $\mathbb{F}_2[x]$, namely we can interpret the whole state (i.e., the memory) as a polynomial

$$s(x) = \sum_{i=0}^{L-1} s_{t+L-1-i} x^i$$

or

$$s(x) = \sum_{i=0}^{L-1} s_{L-1-i} x^i$$

if we call s_0, s_1, s_2, s_{L-1} the memory cells (where the rightmost cell is s_0) without considering the time t. The weights c_i can also be interpreted as coefficients of a polynomial of degree L in $\mathbb{F}_2[x]$ to obtain

$$c(x) = x^L + \left(\sum_{i=1}^{L-1} c_i x^i \right) + 1$$

which is called **feedback polynomial**. Alternatively, one can use the characteristic polynomial, which is:

$$g(x) = x^L c(x^{-1}) = x^L + \sum_{i=1}^{L-1} c_i x^{L-i} + 1 \in \mathbb{F}_2[x]$$

Thanks to this new representation, we can write the bits in the next time instant as

$$s'(x) = s(x) \cdot x \mod c(x)$$

Note that we require the term c_L of the feedback polynomial to be 1 because we want $c(x)$ to have degree L so that we can use $c(x)$ to compute the shift $\mod c(x)$.

5.2.1 Period

The period of the generator is at most $2^L - 1$ since it uses L states and when the state with all zeros is reached, then the generator can't move from it. More in general, the period of an LSFR is the smallest positive integer N such that

$$s_{t+N} = s_t \quad \forall t \geq 0$$

5.2.2 Characterisation of LFSR output sequences

Before introducing an important theorem, let us give some definitions regarding polynomials.

Definition 5.1 (Irreducible polynomial)**.** *A polynomial with degree L is irreducible if it cannot be written as the multiplication of polynomials with a degree less than L.*

Definition 5.2 (Primitive polynomial)**.** *A polynomial with degree L is primitive if each of its roots say r, is able to generate every element in $\mathbb{F}_{2^L} \setminus \{0\}$ as:*

$$r^n$$

and

$$n = 0, ..., 2^L - 1$$

A primitive polynomial is also an irreducible polynomial.

Theorem 5.1 (Characterisation of LFSR output sequences)**.** *Given an LFSR with L memory cells and:*

- *A non-zero initial state $(s_{L-1}, \ldots, s_1, s_0)$.*
- *A feedback network $(c_1, \ldots, c_{L-1}, c_L)$ with $c_L \neq 0$.*

then

- *If $c(x)$ is irreducible then the LFSR produces a keystream with period N, where N is a divisor of the maximum possible period $2^L - 1$. N is the smallest integer such that $C(x)$ is a factor of $x^N + 1$.*
- *If $c(x)$ is primitive then the LFSR produces a keystream with period $N = 2^L - 1$.*

5.2.3 LFSR as pseudo-random number generator

A LFSR with a primitive feedback polynomial has good statistical properties, hence it can be used as a pseudo-random number generator. In particular, the following theorem holds.

Theorem 5.2 (Golomb)**.** *A sequence of k consecutive 0s (or 1s) is called a run of length k. Then:*

- *In every N-bit sequence, the number of zeros is nearly equal to the number of ones.*
- *In every N-bit sequence, $\frac{1}{2}$ the runs have length one, $\frac{1}{4}$ have length two, $\frac{1}{8}$ have length three and, in general, $\frac{1}{2^i}$ runs have length i.*
- *For each run length, there are equally many runs of 0's and of 1's.*
- *Counting the number of digits matching between a period of the sequence and an infinite sequence from the same LFSR yields 0 if the single period is aligned with a period of the infinite sequence, and a constant k otherwise.*

To prove that an LFSR is a good PRNG we can apply a lossless compression algorithm to the output of the LFSR and, if we obtain a small reduction in size (i.e., the size of the LFSR output is similar to the size of the compressed LFSR output), then the original material (i.e., the LFSR output) has maximum entropy, hence it has good statistical properties.

5.2.4 Composition of stream cyphers

LFSR-based stream cyphers can't be used for cryptographic purposes because they apply a linear transformation, hence they are subject to known plaintext attacks. In particular, assume that the length L of an LFSR (i.e., the number of states) is known and that we can obtain $2L$ plaintext, ciphertext pairs $(m_0, c_0), (m_1, c_1), \ldots, (m_{2L-1}, c_{2L-1})$. Since we have the first L ciphertext bits then we can compute $s_0, \ldots, s_{L-1}$ as

$$s_i = c_i \oplus m_i$$

This is true because the first L states aren't influenced by the feedback circuit, hence we can obtain them directly from the ciphertext, plaintext couples. Long story short, we can compute the initial

state of the LFSR. Now we have to compute the weights of the network, hence the structure of the LFSR. Since we have $2L$ couples (m_i, c_i) we can write down a system with L equations using the recursive equation and obtain

$$s_{j+L} = \sum_{i=1}^{L} c_i s_{j+L-i} \mod 2 \quad j \in \{0, \dots L-1\}$$

where c_i is the i-th weight of the network. Since we have a system of L equations in L unknowns $c_0, \dots, c_L$, we can compute a solution and find the structure of the LFSR.

This shows that a single LFSR can't be used for cryptographic purposes. We can however take multiple linear LFSR and mix them to obtain a more complex LFSR which can be used as a secure cypher. More precisely, we would like to obtain a cypher with the following properties:

1. A good stream cypher should have a **long period**.
2. A good stream cypher should have **good randomness properties** as defined by Golomb (5.2).
3. The output bits of a good stream cypher should be obtained in **non-linear way**.
4. A good stream cypher should combine, in a non-linear way, multiple LFSR to obtain a longer period. Note that when combining multiple LFSR we have to be sure to satisfy the other properties. Consider for instance a stream cypher that combines the outputs x_1, x_2, x_3 of three LFSR as $f(x_1, x_2, x_3) = x_1 \oplus x_2 x_3$. If we observe the second term of the sum we notice that its value is 0 three times out of four (it's 1 only when $x_2 = x_3 = 1$), hence the output of the stream cypher is x_1 75% of the times, which is bad since it gives an attacker a hint on the value of the output.

Let us now analyse some examples of good stream cyphers obtained as a combination of multiple LFSR cyphers.

Geffe generator

The Geffe generator uses three LFSRs. One LFSR, the second, is used to choose between the first and third LFSR. In practice, the output is computed as

$$x = x_1 x_2 \oplus x_2 x_3 \oplus x_3$$

or equivalently as

$$x = \begin{cases} x_1 & \text{if } x_2 = 1 \\ x_3 & \text{if } x_2 = 0 \end{cases}$$

This generator isn't however secure since the output isn't chosen uniformly at random, in fact we have $Pr(x = x_1) = Pr(x = x_3) = 0.75$.

Stop-and-go generator

In a stop-and-go generator one (or more) LFSR is used to clock the others. A variation of this generator is the alternating stop-and-go generator which is built with three LFSRs. The first LFSR is used to trigger the clock signal between the second and the third. The output is therefore obtained as $x_2 \oplus x_3$ and,

- If $x_1 = 0$, the second LFSR is clocked.
- If $x_1 = 1$, the third LFSR is clocked.

Shrinking generator

A shrinking generator uses two simultaneously clocked LFSRs in parallel and:

- If the first LFSR is 1, then the value of the second LFSR is used.
- If the first LFSR is 0, then the value of the second LFSR is discarded and no bit is generated.

A5

A5 uses three LFSR cyphers, with 64 state bits in total (distributed among the three cyphers). The cyphers are defined by the polynomials:

- $C_1(x) = x^{19} + x^5 + x^2 + x + 1$
- $C_2(x) = x^{22} + x + 1$
- $C_3(x) = x^{23} + x^{15} + x^2 + x + 1$

The output is obtained xoring the outputs of the three cyphers. This version of the protocol, called A5/1, was secretly designed for GSM mobile-phone networks, however, it was reverse-engineered and broken. A second version of the protocol, called A5/2 was subsequently designed but it has also been broken. The current version of the protocol, A5/3, uses a Feistel network with a 128-bit key and 64-bit block size. The design of A5/3 is publicly available and, even if some attacks have been published, they can't be practically executed.

5.2.5 Software-oriented stream cyphers

Some stream cyphers are mainly designed to be run in software.

Ron's cypher 4

An example of a software-oriented stream cypher is Ron's cypher 4 (RC4). This cypher was initially used for securing Wi-Fi, however, it's now broken and has been replaced by WPA2, which uses AES in counter mode for encryption and AES in CBC mode, on the last block, to provide integrity (i.e., AES-CBC is used as message authentication code).

RC4 uses keys with lengths that span from 5 to 256 bytes. The cypher state is made by

- an array S of 256 bytes, and
- two 8-bit index-pointers i and j.

The array S is initialised with the key k using the following algorithm:

for $i \in \{0, \ldots, 255\}$ **do**
  $S[I] \leftarrow i$
end for
$j \leftarrow 0$
for $i \in \{0, \ldots, 255\}$ **do**

$j \leftarrow (j + S[i] + k[i \mod keylength]) \mod 256$
SWAP$(S[i], S[j])$
end for

Encryption and decryption work as always but we xor whole bytes instead of single bits. Each byte of the plaintext is xored with a byte of the keystream. The bytes of the keystream are generated using the following pseudo-random generation algorithm:

$i \leftarrow 0$
$j \leftarrow 0$
while $True$ **do**
$i \leftarrow (i + 1) \mod 256$ ▷ each byte in S is used once after 256 iterations
$j \leftarrow (j + S[i]) \mod 256$ ▷ the output depends non-linearly from S
SWAP$(S[i], S[j])$ ▷ S is modified at each iteration
$k \leftarrow S[(S[i] + S[j] \mod 256)]$ ▷ the output is hard to link with the state S output k
end while

Chapter 6

Hash functions

6.1 Hash function

One of the most important classes of functions in cryptography is the class of hash functions. A hash function $h : \mathcal{I} \to \mathcal{D}$ is a known function (i.e., whose design is known) that provides data integrity by computing the fingerprint of its input through a non-injective function (i.e., $|\mathcal{D}| < |\mathcal{I}|$). The output of the hash function is usually called digest $d = h(i)$. Since a hash function is non-injective, it can take an input of any size and generates an output of fixed size (which is smaller than the input). This means that we have two inputs $i_1, i_2 \in \mathcal{I}$ that generate the same digest, namely

$$\exists i_1, i_2 \in \mathcal{I} : h(i_1) = h(i_2)$$

Two inputs that generate the same digest are called **collision**. If the digest size is not long enough, it's easier to find collisions, hence it's better to have a long digest. Note that this behaviour is expected since the hash is used to compute a fingerprint of some data, hence it has to reduce the input size. This behaviour isn't however a problem if the hash function is designed correctly and it's very rare to obtain a collision between two meaningful messages.

A hash function doesn't necessarily include a key since it's not built to encrypt the data, but only to provide integrity. In practice, a hash function is used to compute the digest of a message m. The message is then sent with its digest and the receiver can compute the digest on the received message and compare it with the digest received. If the two values are the same, the receiver is sure to have received the message sent by the sender.

From this introduction we have understood the basic principles of hash functions, however, we still miss a more formal definition.

Definition 6.1 (Hash function)**.** *A hash function is a tuple*

$$(\mathcal{M}, \mathcal{D}, \mathcal{K}, \mathcal{H})$$

where

- $\mathcal{M}$ *is the message space, i.e., a finite set of strings used as input of the hash function.*
- $\mathcal{D}$ *is the digest space, i.e., a finite set containing all the digests that can be generated.*
- $\mathcal{K}$ *is the key space, i.e., a finite set containing keys.*

- *$\mathcal{H}$ is the space of all hash functions. In particular, for each key $k \in \mathcal{K}$ we can identify a unique function $h_k : \mathcal{M} \to \mathcal{D} \in \mathcal{H}$.*

A couple (m, d) composed of a message $m \in \mathcal{M}$ and a digest $d \in \mathcal{D}$ is valid if $h_k(m) = d$.

6.2 Properties of unkeyed hash functions

Let us now consider hash functions that do not require a key. A secure hash function should satisfy the following properties.

Principle 6.1 (One way property)**.** *A hash function $h : \mathcal{M} \to \mathcal{D}$ is a way function if, given a digest $d \in \mathcal{D}$, it should be mathematically hard to compute $m = h^{-1}(d) \in \mathcal{M}$.*

Principle 6.2 (Weak collision resistance)**.** *An hash function $h : \mathcal{M} \to \mathcal{D}$ is weakly collision resistant if, given a message $m_1 \in \mathcal{M}$ it's mathematically hard to find another message $m_2 \in \mathcal{M}$ (with $m_1 \neq m_2$) such that $h(m_1) = h(m_2)$.*

Principle 6.3 (Strong collision resistance)**.** *A hash function $h : \mathcal{M} \to \mathcal{D}$ is strongly collision resistant if it's hard to find two messages $m_1, m_2 \in \mathcal{M}$ (with $m_1 \neq m_2$) such that $h(m_1) = h(m_2)$.*

The third problem (i.e. finding two messages that collide) is easier to solve than the second one since in the first case we can choose the messages, whereas in the second case, we are forced to find a collision with a given message. This means that a function that is strongly collision resistant is more secure than a weakly collision resistant one. In other words, strong collision resistance implies weak collision resistance which implies the one-way property. Let us now analyse these properties more in detail.

6.2.1 Random oracles

A random oracle is a conceptual model for hash functions used to represent a perfect hash function. More precisely, a random oracle is a deterministic function that generates a different random string for every possible input string, hence it can be seen as a lookup table. A random oracle isn't a true hash function and it's only used for formal proofs.

6.2.2 One-way property

Let us now further analyse the one-way property without considering a specific hash function. This property states that given a digest d it's mathematically hard to find a message $m \in \mathcal{M}$ such that $m = h(d)$. This problem is called **first preimage problem**.

Now we want to analyse what is the probability of finding, trying at random, one message that generates the desired digest. Namely, we try a series of guesses until we find a message m_i such that $h(m_i) = \bar{d} = d$. If the hash function has good randomness properties, every time we pick a message

m, it has a probability of $\frac{1}{|\mathcal{D}|}$ of having digest $h(m) = \bar{d} = d$, namely we have

$$Pr(\text{success on one pick}) = \frac{1}{|\mathcal{D}|}$$

We can use this result to compute the probability that the algorithm for solving the problem (i.e., the algorithm that picks a message until we find the correct digest) picks q messages before stopping as the probability that we don't succeed q times. The probability of succeeding when picking a message is $\frac{1}{|\mathcal{D}|}$, hence the probability of not succeeding is $1 - \frac{1}{|\mathcal{D}|}$. The probability that the algorithm requires q steps (i.e., that after q steps the algorithm hasn't found the correct digest) is therefore

$$Pr(\text{algorithm stops after } q \text{ steps}) = \left(1 - \frac{1}{|\mathcal{D}|}\right)^q$$

It's now easy to compute the probability that the algorithm doesn't stop in q steps, hence the probability that the algorithm finds at least one message in q steps. This probability is

$$Pr(\text{algorithm finds at least one } m \text{ in } q \text{ steps}) = 1 - \left(1 - \frac{1}{|\mathcal{D}|}\right)^q$$

Let us now shift our focus back on the term $\left(1 - \frac{1}{|\mathcal{D}|}\right)^q$. This term can be rewritten as

$$\begin{aligned} Pr(\text{algorithm stops after } q \text{ steps}) &= \left(1 - \frac{1}{|\mathcal{D}|}\right)^q \\ &= \left(1 + \frac{-1}{|\mathcal{D}|}\right)^{|\mathcal{D}|\frac{q}{|\mathcal{D}|}} \end{aligned}$$

If we now consider the limit of this probability for $|\mathcal{D}| \to \infty$ we obtain

$$\begin{aligned} Pr(\text{algorithm stops after } q \text{ steps}) &= \left(1 - \frac{1}{|\mathcal{D}|}\right)^q \\ &= \left(1 + \frac{-1}{|\mathcal{D}|}\right)^{|\mathcal{D}|\frac{q}{|\mathcal{D}|}} \\ &\approx e^{-\frac{q}{|\mathcal{D}|}} \end{aligned}$$

Now we can consider the limit for $\frac{q}{|\mathcal{D}|} \to 0$ to obtain

$$\begin{aligned} Pr(\text{algorithm stops after } q \text{ steps}) &= \left(1 - \frac{1}{|\mathcal{D}|}\right)^q \\ &= \left(1 + \frac{-1}{|\mathcal{D}|}\right)^{|\mathcal{D}|\frac{q}{|\mathcal{D}|}} \\ &\approx e^{-\frac{q}{|\mathcal{D}|}} \\ &\approx 1 - \frac{q}{|\mathcal{D}|} \end{aligned}$$

This means that the probability of finding at least one solution to the problem is

$$\begin{aligned} Pr(\text{algorithm finds at least one } m \text{ in } q \text{ steps}) &= 1 - \left(1 - \frac{1}{|\mathcal{D}|}\right)^q \\ &= 1 - \left(1 - \frac{q}{|\mathcal{D}|}\right) \\ &= \frac{q}{|\mathcal{D}|} \end{aligned}$$

6.2.3 Weak collision resistance

Let us now repeat the same analysis done for the one-way property for weak collision resistance. In this case, the problem we want to solve is finding a message $m_2 \in \mathcal{M}$ that has the same digest as a given message $m_1 \in \mathcal{M}$. This problem is called **second preimage problem**. Since we can't compute the inverse of the hash function, we have to pick q messages $m, m_1, \ldots, m_{q-1}$ at random (where $m_i \neq m$) and, for each message, we have to compute $h(m_i)$ and compare it with $h(m)$. Note that the given message m is also chosen at random since we must be able to solve the problem for any given message m. The scenario is identical to the one seen for the one-way property, hence we can compute the probability of finding a conflict in q tries as

$$\begin{aligned} Pr(\text{algorithm finds at least one } m \text{ out of } q \text{ messages}) &= 1 - \left(1 - \frac{1}{|\mathcal{D}|}\right)^{q-1} \\ &\approx \frac{q-1}{|\mathcal{D}|} \end{aligned}$$

In this case, we have to do $q - 1$ comparisons because one of the $q - 1$ messages is the one we are comparing all the others against.

6.2.4 Strong collision resistance

Let's repeat the same black-box analysis for the strong collision resistance property. The problem we want to solve, which is called **collision problem**, requires us to find any two messages m_1, m_2 that are mapped to the same digest. In this scenario, the algorithm picks q messages and checks if there exist two messages, among the q picked, with the same digest. Given the probability to find no conflicts, it's easy to find the probability that at least two messages have the same digest, hence we can start focusing on the probability that no two messages exist that map to the same hash. If we pick only one message, the probability of not finding a collision is 1 since we need at least two messages, hence we can write

$$Pr(\text{No collision with 1 message}) = 1$$

If we consider two messages, the probability of not finding a collision is the one we've computed for the one way property, namely we get

$$Pr(\text{No collision with 2 messages}) = 1 \cdot \left(1 - \frac{1}{|\mathcal{D}|}\right)$$

Let's consider three messages. In this case, we have to compute the probability of not finding a collision when selecting two messages and the probability of not finding a collision when picking

three messages, hence we obtain

$$Pr(\text{No collision with 3 messages}) = 1 \cdot \left(1 - \frac{1}{|\mathcal{D}|}\right) \cdot \left(1 - \frac{2}{|\mathcal{D}|}\right)$$

In general, for $q = q$, we get

$$\begin{aligned} Pr(\text{No collision with } q \text{ messages}) &= 1 \cdot \left(1 - \frac{1}{|\mathcal{D}|}\right) \cdot \left(1 - \frac{2}{|\mathcal{D}|}\right) \cdot \dots \cdot \left(1 - \frac{q-1}{|\mathcal{D}|}\right) \\ &= \prod_{i=1}^{q-1} \frac{i}{|\mathcal{D}|} \end{aligned}$$

If we remember that

$$1 - \frac{a}{x} \leq e^{-\frac{a}{x}}$$

then we can write

$$\begin{aligned} Pr(\text{No collision with } q \text{ messages}) &\leq \prod_{i=1}^{q-1} e^{-\frac{i}{|\mathcal{D}|}} \\ &= e^{-\frac{1}{|\mathcal{D}|}\sum_{i=1}^{q-1} i} \\ &= e^{-\frac{q(q-1)}{2|\mathcal{D}|}} \end{aligned}$$

Ideally, we'd want this probability to be higher than 0.5, hence we can write

$$\begin{aligned} Pr(\text{No collision with } q \text{ messages}) \geq 0.5 &\iff e^{-\frac{q(q-1)}{2|\mathcal{D}|}} \geq \frac{1}{2} \\ &\iff \frac{q(q-1)}{2|\mathcal{D}|} \leq \ln 2 \end{aligned}$$

Solving for q we get

$$0 < q \leq \frac{1}{2} + \frac{1}{4}\sqrt{\frac{1}{4} + 2|\mathcal{D}| \ln 2}$$

which is roughly

$$0 < q \leq 1.1774\sqrt{|\mathcal{D}|}$$

This means that a hash function is secure if it's computationally unfeasible to find a collision in less than $1.1774\sqrt{|\mathcal{D}|}$ calls to the hash function. If we consider an hash function with n bits, then the security margin of such a function is $\sqrt{2^n}$, namely $2^{\frac{n}{2}}$. In practice, this means that we should take $|\mathcal{D}| \geq 2^{160}$, which yields a security margin of 2^{80}. This has very important implications.

6.3 Usages of hash functions

Hash functions are used in many applications:

- **File integrity**. Since hash functions are used to compute a fingerprint of the input, we can always verify if some data has been tampered with by computing its hash and comparing it with a previously computed hash.

- **Pseudo-unique file indexing**. Given that a digest is usually short, we can use it to index files. In theory, two files might have the same hash, however, this happens so rarely that it's almost impossible to find two files with the same digest.
- **Digital signatures**. Digital signatures are usually computed using RSA, which is very expensive. This means that signing a file might be a lengthy operation. To solve this problem we can sign the hash of a file instead of the file itself, since the digest is way smaller.
- **Safe password storage**. If passwords are stored in clear on a web server, whenever the server is broken and its storage is made public, everyone will obtain the users' passwords. To avoid this problem passwords' hashes are stored instead of the password itself. In this case, the user still prompts the password and then the server computes its hash and compares it with the one stored on the server. Note that the current state-of-the-art requires a password to be stored with a salt (i.e., $h(pwd||salt)$ is stored).
- **Commitment schemes**. Similarly to digital signatures, hashes can be used to prove that one has committed to a choice. Say for instance two people have to throw a coin. Each person makes a choice and computes the hash of that choice. After throwing the coin, one can check that the other didn't change his or her choice by comparing the hash of the choice after knowing the result with the hash computed before knowing the result of the coin flip.

Even if hash functions work well for many different applications, we shouldn't use them everywhere. An example is error correction codes. Error correction codes are designed to detect an error in a message and possibly correct it. In other words, error correction codes provide integrity but the first preimage problem is easy to solve (since it's required to fix the error in the message). This means that hash functions aren't a good choice to implement an error correction code since they can't be used to fix the error in the message. In other words, hashes should be used only as error detection codes, namely to detect an error (i.e., ensuring integrity) without fixing it. Put another way, hashes should be used to detect errors in a message only when it's cheap to resend the message.

6.4 Hash functions design

As for now, we have listed the properties that hash functions should satisfy, however, we have treated them as black boxes. Now we want to understand how hash functions work internally and how to implement them. First, let us define a hash function as a function from the space $\{0,1\}^*$ of bit-strings of any length to the space $\{0,1\}^s$ of bit-strings of length s, namely

$$h : \{0,1\}^* \to \{0,1\}^s$$

Since we want to deal with inputs of undefined length, a good idea would be to use a compression function that works on inputs of fixed length, split the input into blocks and apply the compression function to each block. The input $\geqslant$ is divided into blocks (or chunks) $\bar{m}_1, \ldots, \bar{m}_r$ of size t. This means that we have to define:

- A **block compression function**. The block compression function f acts on bit-strings of fixed length $l + t$ and compresses them to obtain a bit-string of length l, namely, we have to define a function

  $$f : \{0,1\}^{l+t} \to \{0,1\}^l$$

  The first l bits passed in input to the compression function are called **state bits** and the following t bits are called **message-chunk bits**. The compression function is, in fact, given one input block $\bar{m}_i$ of length t and a state z_i of length l.

- A **way to combine the outputs of the block function**. In general, a hash function combines an input block m_i of size t with some state z_j, of size l, of the function. This means that we have to define a way to initialise and update the state of the function. The initial state z_0 is obtained from an initialisation vector whereas the next state is computed using the output of the compression function f, hence we could write

  $$z_i = c(f(z_{i-1}, m_i))$$

  where c is a function that returns the next state. Note that the collision resistance is achieved through mixing the message into the state so that a random change in the message, triggers a bit flip in every bit of the digest with probability as close as possible to $\frac{1}{2}$ (this is commonly known as the avalanche effect). Moreover, the message mixing is usually performed through a nonlinear addition (e.g., addition modulo 2^{32}) during the round function. A hash scheme behaves like a cypher block but for hash functions, we don't have a key hence the number of rounds must be higher with respect to block cyphers. As always, if the size of the string m in input to the hash function isn't a multiple of t, we can use padding to extend the input and make it of the appropriate length.

- A **final expansion function**. The final state of the function is passed to a function g that expands it since it has length l and we want a digest of size s. Namely, the digest is computed as

  $$d = g(z_{last})$$

  Most of the time, the size of the state is the same as the one of the hash function output, hence g simply is the identity function.

To sum things up, a hash function:

1. Initialises the state z_0.
2. Takes an input block $\bar{m}_i$ of length t.
3. Compresses the block using the compression function,

   $$o_i = f(z_{i-1}, \bar{m}_i)$$

4. Uses the output of the compression function to compute the next state,

   $$z_i = c(o_i)$$

5. Goes to step 2 if the message isn't over.
6. Computes the digest as $h = g(z_r)$.

6.4.1 Merkle-Damgård scheme

The Merkle-Damgård scheme is one of the most famous architectures for building hash functions. The MD scheme is very simple and uses the output of the compression function as the next state. This means that the state at time i is computed as

$$z_i = f(z_{i-1}, m_i)$$

This means that the function c is the identity function. A graphical representation of the Merkle-Damgård scheme is shown in Figure 6.1. Most of the attacks on this scheme focus on the final part of the scheme and in particular try to add a block to the scheme (i.e., append some bits at the end of the message). On the other hand, an attack on inner blocks would be very hard since modifying an inner block will change all subsequent blocks.

This design has some security flaws since, if an attacker can obtain any state of the hash function, then he or she can compute all future states and finally append some completely arbitrary blocks. This attack is called suffix attack and allows to append arbitrary content at the end of a hash, which is a big problem.

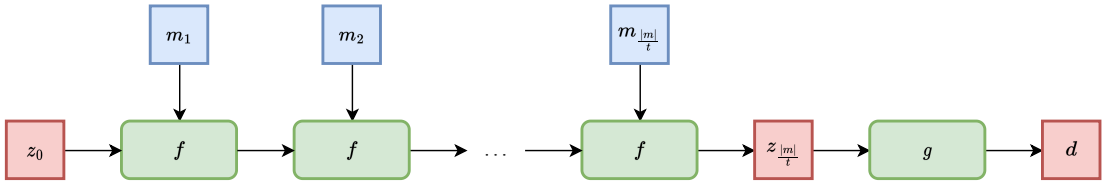


Figure 6.1: The general structure of the Merkle-Damgård scheme.

SHA-1 compression function

SHA-1 uses a Merkle-Damgård scheme. This means that we only have to analyse the implementation of the compression function to understand how SHA-1 works. The compression function takes as input a block of 160 bits, divided into 5 words of 32 bits each and produces an output of the same size. One round of the compression function changes one word in input and rearranges the others through:

- A 5-bit left rotation $\lll 5$.
- A 30-bit left rotation $\lll 30$.
- A round-dependant bit-wise function F.

These building blocks are used to compute the output of one round as shown in Figure 6.2. The whole function h is made of 80 rounds of the aforementioned function F where the output of a round is used as input for the next round.

6.4.2 Hash functions from block cyphers

A hash function scheme is similar to a block cypher, hence we can adapt block cyphers to be used as hash functions. The compression function of a hash function takes two inputs, which are a block of the message to hash and the state of the hash function, and produces one output. One building block of a block cypher also takes two inputs (usually a block of plaintext and one round key) and generates one output. This means that we have to understand if the state has to be used as a key and the message as plaintext or vice-versa and possibly how to generate the new state from the block function's output.

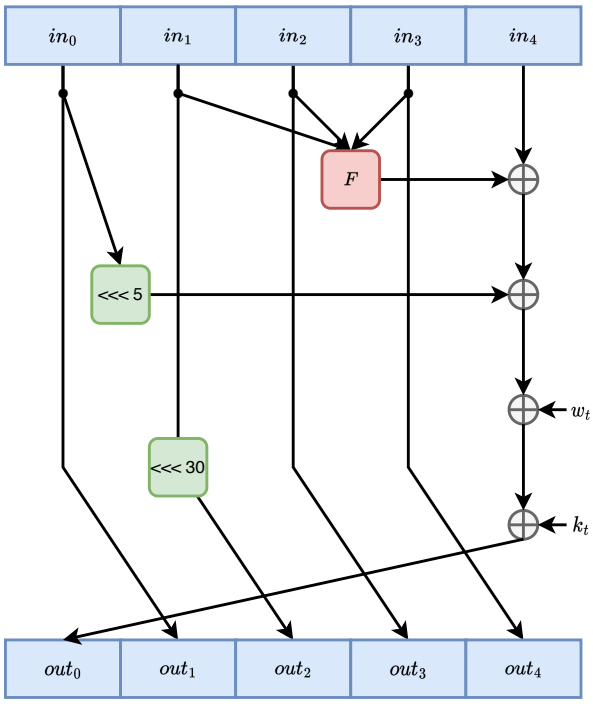


Figure 6.2: SHA-1 compression function.

Davies-Meyer scheme

The Davies-Meyer scheme is the simplest way of implementing a hash function using a block cypher. In particular:

- One block of message is used as the key of the block cypher function.
- The state of the hash is used as the plaintext of the block cypher function.
- The state is updated by xoring the output of the block cypher function and the previous state. Namely, the state at step i, i.e. z_i, is computed as

  $$z_i = \mathbb{E}_{m_i}(z_{i-1}) \oplus z_{i-1}$$

  where m_i is the i-th block of the message. This means that the compression function of the hash is

  $$f(m_i, z_{i-1}) = \mathbb{E}_{m_i}(z_{i-1}) \oplus z_{i-1}$$

The Davies-Meyer scheme is shown in Figure 6.3.

Matyas-Meyer-Oseas scheme

The Matyas-Meyer-Oseas scheme swaps message and key. Namely, the block of the message is used as input of the block cypher and the state is used as the key. As a result, the state Z_i at step i is computed as

$$z_i = \mathbb{E}_{z_{i-1}}(m_i) \oplus z_{i-1}$$

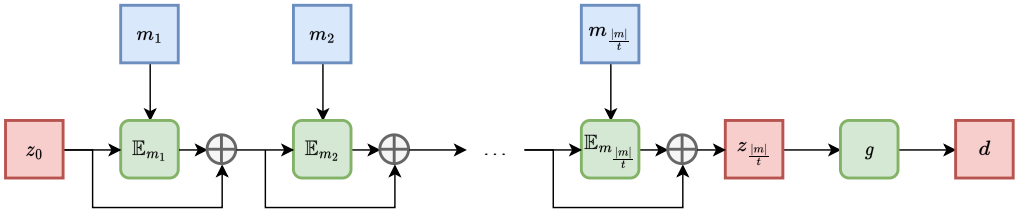


Figure 6.3: The Davies-Meyer scheme.

and the compression function is

$$f(m_i, z_{i-1}) = \mathbb{E}_{z_{i-1}}(m_i) \oplus z_{i-1}$$

The Matyas-Meyer-Oseas scheme is shown in Figure 6.4.

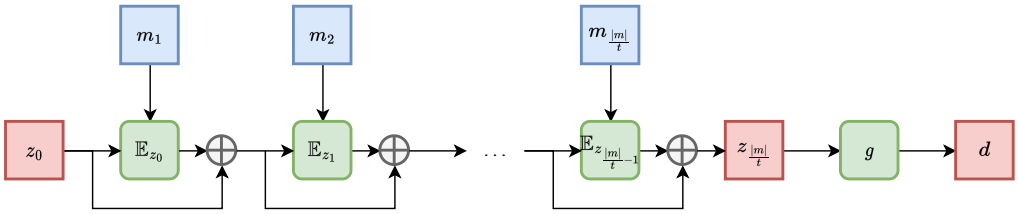


Figure 6.4: The Matyas-Meyer-Oseas scheme.

Miyaguchi-Preneel scheme

The Miyaguchi-Preneel scheme is based on the Matyas-Meyer-Oseas scheme but the new state after an iteration is computed as the xor between the message, previous state and block cypher function's output. In formulas, the state z_i at iteration i is computed as

$$z_i = \mathbb{E}_{z_{i-1}}(m_i) \oplus z_{i-1} \oplus m_i$$

and the compression function is

$$f(m_i, z_{i-1}) = \mathbb{E}_{z_{i-1}}(m_i) \oplus z_{i-1} \oplus m_i$$

The Miyaguchi-Preneel scheme is shown in Figure 6.5.

6.5 Keyed hash functions

Hash functions do not require a key, however, this doesn't provide message integrity. In fact, a man in the middle can intercept a message, change the message, recompute the hash and send the new message and relative hash to the intended sender. Adding a key to the hash computation ensures that only those having the key can compute the hash and verify the integrity of the message. More precisely, if we add a key to the hash function:

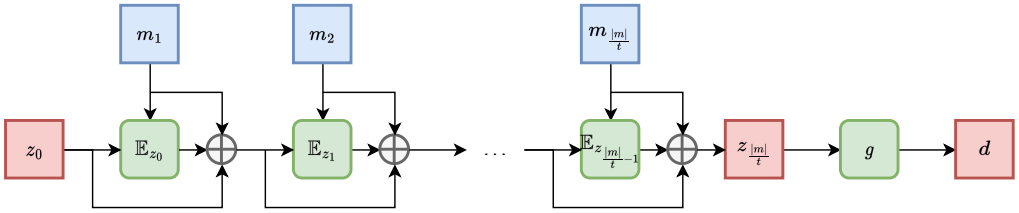


Figure 6.5: The Miyaguchi-Preneel scheme.

- A man in the middle attacker can't compute another valid couple (m', d') given a couple (m, d) and without the key used to compute $d = h_k(m)$.
- An attacker without the key can't verify the integrity of a message m when given a couple (m, d).

The key has to be included in the message to hash. If the hash function is perfect, the position of the key is irrelevant (i.e., it can be used as a suffix or prefix, too). If we use a Merkle-Damgård scheme, we have to choose where to put it. In particular, the key should never be used as a suffix or prefix since it leads to prefix and suffix attacks.

6.5.1 Key prefix attack

Let us consider a Merkle-Damgård scheme where the key is used as the prefix of the message. This means that the digest of a message m using key k is computed as

$$d = h(k||m)$$

where $||$ is the symbol for concatenation. The key prefix attack allows, given a couple (m, d) to obtain a valid pair $(m||m', d')$, without knowing the secret key. In particular, an attacker can pick an arbitrary message m' and compute

$$\begin{aligned} d' &= h(d||m') \\ &= h(h(k||m)||m') \\ &= h(k||m||m') \end{aligned}$$

which is a valid hash (but not necessarily a collision) since it contains the key at the beginning of the message to hash. This happens because the next state is computed using the previous state, hence if we control a state (i.e., the digest d, which is the last state of the hash function) we can use it to compute the next state, hence the digest obtained adding a suffix m' to the original message m. The reason why this attack works is that the key is used as a prefix, hence it's also a prefix of the message $m||m'$.

6.5.2 Key suffix attack

Let us now consider a Merkle-Damgård scheme where the key is used as a suffix of the message. This means that the digest of a message m using key k is computed as

$$d = H(m||k)$$

A key suffix attack is a bit harder than a key prefix attack however it still allows, given a couple (m, d), to reuse d as a digest for any message m' colliding with the original one. Namely, we can compute a valid digest such that

$$H(m) = H(m')$$

In other words, if we use a key as a suffix, every message m' in collision with m is also in collision with $m||k$, regardless of k. In other words, using a key as a suffix allows switching from keyed to unkeyed hash functions. In particular, if the length of the known message m is a multiple of t bits, namely

$$m = m_1||m_2||\dots||m_r$$

the attacker can consider the last block m_r of the message and the l-bit block $y = z_r$ derived from the Merkle-Damgård construction and can observe that

$$H(m) = h(y, m_r)$$

with $y = h(h(IV, m_1), \dots, m_{r-1})$ while

$$H(m||k) = h(h(y, m_r), k)$$

If the attacker obtains a message m' colliding with m, namely,

$$H(m') = H(m) \iff h(z_{r-1}, m_r) = h(z_{r-1}, m'_r)$$

the keyed digest of m' would be

$$h(m'||k) = h(h(y||m_r)||k) = d,$$

thus (m', d) is another valid message-digest pair. In short, we can write

$$\begin{aligned} d &= h(z_r, k) \\ &= h(h(z_{r-1}, m_r), k) \\ &= h(h(z_{r-1}, m'_r), k) \\ &= h(z'_r, k) \\ &= d' \end{aligned} \qquad H(m') = H(m) \iff h(z_{r-1}, m_r) = h(z_{r-1}, m'_r)$$

When m is shorter than t bits, the above collision attack cannot be mounted because part of the key k would become part of the first t-bit chunk in input to h.

6.6 Message Authentication Codes

A Message Authentication Code (MAC) is a string of bits used to ensure the integrity of a message. A MAC is usually appended to a sequence of plaintext blocks and it's used to prove to the receiver that the plaintext originated from the rightful sender and was not tampered with. Message Authentication Codes can be built using hash functions or block cyphers. A MAC built using a hash function is called HMAC. MACs are very useful since the block size of hash primitives is larger than block cypher ones, thus the schemes based on a general compression function can exhibit a better collision resistance. Moreover, the computation of a MAC should be more efficient than a hash-and-then-encrypt solution.

6.6.1 Hash Message Authentication Codes

An HMAC is a key-Hashed Message Authentication Code computed using a hash function. In particular, an HMAC is computed by applying a hash function twice. For instance, if we take SHA-2 as the hash function, we can compute the HMAC of a message m as

$$HMAC_k(m) = \text{SHA-2}((k \oplus opad)||\text{SHA-2}((k \oplus ipad)||m))$$

where

- $||$ is the concatenation.
- k is a key.
- *ipad* is the 512 bit constant `0x363636...36`.
- *opad* is the 512 bit constant `0x5c5c5c...5c`.

This formula can be split into two parts to obtain:

- $T = \text{SHA-2}((k \oplus ipad)||m)$
- $HMAC_k(m) = \text{SHA-2}((k \oplus opad)||T)$

These two components act on different domains, hence we can see them as independent hashes and even use different functions.

Note that we can build a secure HMAC even by using a hash function which is only second-preimage resistant (i.e., weak collision resistant), hence we could use (even if it's strongly discouraged) MD5.

6.6.2 Block cypher Message Authentication Codes

Since compression functions and block encryption functions have the same shape, we can use the latter as a compression function to build a MAC.

CBC-MAC

The easiest way to implement a MAC using a block cypher is by using a block cypher in CBC mode and keeping only the last encrypted block as MAC. This solution has however many issues. For instance, if d is a valid MAC for $m = m_1||\cdots||m_n$, then d is a valid MAC also for $m' = m_1||\cdots||m_n||(IV \oplus d \oplus m_1)||\cdots||m_n$. This attack exploits the same vulnerability highlighted for the hash functions.

CMAC

To solve the problem with CBC-MAC it is possible to employ a whitening step, which involves xoring the key to the whole input message, before applying the block cypher encryption. A proper way to do so has been standardised as CMAC and is currently accepted for the IPsec checksums.

Chapter 7

Public key cryptography

7.1 Introduction

Public key cryptography has a radically different approach with respect to shared key cryptography, in fact, messages aren't encrypted with a single key shared between the parties. Instead, each party has a couple of keys:

- A public key k_{pub} or k_p which is publicly available to everyone and uniquely linked to a person through a certificate.
- A private key k_{priv} or k_{sec} or k_s which is kept as a secret by each person.

The keys are mathematically linked one to the other but given a person's public key (i.e., the one publicly available to anyone) it should be hard to obtain the corresponding private key. On the other hand, once the private key is generated, it should be easy to generate the corresponding public key. The keys are generated such that

- a message encrypted with a person's private key can only be decrypted with that person's public key, and
- a message encrypted with a person's public key can only be decrypted with that person's private key.

This means that, in practice, if Alice wants to send a message to Bob she can use (i.e., encrypt the message with) Bob's public key (which she knows since it's publicly advertised by Bob) and Bob can decrypt it with his private key. Since a message encrypted with Bob's public key can only be decrypted with Bob's private one and the private key is kept as a secret by Bob, then only Bob can decrypt a message encrypted with his public key.

Public key cryptography is also known as **asymmetric key encryption** to highlight the difference with symmetric key encryption (which uses a single key).

More formally, we can define a public key cryptosystem as follows.

Definition 7.1 (Public key cryptosystem)**.** *A public key cryptosystem is a tuple*

$$\left(\mathcal{A}, \mathcal{M}, \mathcal{C}, \mathcal{K}, \{\mathbb{E}_{k_{pub}}(m) : k_{pub} \in \mathcal{K}\}, \{\mathbb{D}_{k_{priv}}(c) : k_{priv} \in \mathcal{K}\}\right)$$

where

- $\mathcal{A}$ *is an alphabet.*
- $\mathcal{M}$ *is a plaintext space.*
- $\mathcal{C}$ *is a ciphertext space.*
- $\mathcal{K}$ *is a key space.*
- $\mathbb{E}_{k_{pub}}$ *is an encryption transformation.*
- $\mathbb{D}_{k_{priv}}$ *is a decryption transformation.*

7.1.1 Usages and disadvantages

Public key cryptography is much slower than symmetric key encryption, hence it should be used only in specific cases. Some examples are:

- When we want to encrypt small secrets like identification numbers or passwords.
- When we want to encrypt a key (or a password) which can then be used for symmetric key encryption. In this way, the key is shared between the two parties in a secure way and then we can use symmetric key encryption for the rest of the communication to achieve a higher communication speed (being symmetric key encryption much faster).
- When we want to provide non-repudiation. Since a message encrypted with a person's private key can only be decrypted with the same person's public key, then we can sign a message with a person's private key such that he or she can't say it was he or she that signed that message by using the corresponding public key.
- When we need to provide data integrity and data origin authentication.
- When we need to provide entity authentication and handle authenticated key establishment protocols.

Authentication

Among all things public key cryptography can be used for, authentication is one of the most important. When dealing with authentication we have to introduce two functions $\text{sign}_{k_{priv}}(m)$ and $\text{check}_{k_{pub}}(s)$, which in general are different from the encryption and decryption functions, that are used to sign a message and to check the sign of a message. The sign function should use the signer private key whereas the check function should use the signer public key. Say Alice wants to authenticate herself to Bob, then:

1. Alice sends to Bob a message m and the same message signed with Alice's private key $\text{sign}_{k_{priv,A}}(m)$.
2. Bob checks Alice's sign using the check function with Alice's public key.

If the message is a document or a big file, Alice should hash the message and then sign the digest since public key encryption is computationally expensive. This means that Alice should send to Bob the couple

$$\langle m, \text{sign}_{k_{priv,A}}(h(m)) \rangle$$

In general, the sign and check functions are different from the encryption and decryption functions but in RSA (which is a very popular public key cryptosystem) the functions for encrypting and decrypting are used for signatures, too. This means that the authentication process works as follows:

1. Alice sends to Bob a message m and the same message encrypted with her private key $s = \mathbb{E}_{k_{priv}}(m)$. Note that in this case, we are using the private key for encrypting a message instead of using the public one. This is because if a message is encrypted with someone's private key, we are sure that it came from that person (and this is what we want to ensure when authenticating someone). In other words, in this case, we aren't ensuring confidentiality.

2. Bob decrypts the encrypted message with Alice's public key and compares it with the message. This means that the check function can be written as

$$m = \mathbb{D}_{k_{pub,A}}(s)$$

where s is the sign received by Alice and, if the message wasn't tampered with it should hold $s = \mathbb{E}_{k_{priv}}(m)$, hence

$$m = \mathbb{D}_{k_{pub,A}}(s) = \mathbb{D}_{k_{pub,A}}(\mathbb{E}_{k_{priv}}(m)) = m$$

As we have said, signing a message doesn't provide confidentiality. If we want this property to hold, Alice (i.e., the person that wants to authenticate) should also encrypt the message with Bob's (i.e., the verifier's) public key. In this context, we have two possibilities:

- **Sign then encrypt**. Only the intended receiver of the message can check the signature since the other parties can't decrypt the signature, hence they can't get the signature which has to be verified. This is the secure way of proceeding.

- **Encrypt then sign**. Everyone can verify the signature but only the indented receiver can also read the signed message. This method is used in wired transmission since it allows checking the signature first and then decrypting the message, which allows increasing the transmission speed.

7.1.2 Implementation of computational secure cyphers

Perfectly secure cyphers are practically impossible to implement, hence a public key cryptosystem can only be computationally secure. To achieve such a goal we have to exploit computationally hard problems (i.e., problems which require a non-polynomial execution time) such as:

- **The integer factorisation problem**. Solving the integer factorisation problem means finding, given a number $n = \prod_i p_i$, the factors p_i.

- **The discrete logarithm problem**. Solving the discrete logarithm problem means extracting the logarithm of a number in a cyclic group $(\langle g \rangle, \cdot)$. In other words, given $g_1 \in \langle g \rangle$ and g, find $x \in G$ such that $g_1 = g^x$ with $x \in \{0, 1, \ldots, |\langle g \rangle| - 1\}$

- **The problem of finding the shortest vector in a l-dimensional vector space**.

- **The problem of decoding a general linear code**.

The last two problems are used also for post-quantum cryptosystems, hence cryptosystems based on them should be secure even when quantum computers will be available.

These problems allow us to define a couple of public and private keys such that:

1. The public key and the private key are linked in a mathematical way.
2. The knowledge of the public key tells us nothing about the private key.
3. The knowledge of the private key allows us to decrypt messages encrypted with the public key. The encryption is also called a one-way trapdoor function since it's easy to compute with the public key but hard to invert (i.e., it should be easy to decrypt a message only with the private key).

7.2 RSA

RSA, whose name comes from the names of its inventors Rivest, Shamir and Adleman, is one of the most popular public key cryptosystems.

7.2.1 Key definition

Describing RSA means defying how the keys are built and how encryption and decryption is performed. Let's start from the key definition.

Public key

RSA's public key is obtained by picking two secure large prime numbers p and q with approximately the same number of bits and not close one to the other. The first part of the public key is obtained as the product of such prime numbers, namely

$$n = p \cdot q$$

The value n is also called **RSA public modulus**. The second part of the public key is a random number picked in the equivalence class modulo $\varphi(n)$ (where φ is Euler's totient function used to compute the number of values smaller than n and coprime with n), namely

$$e \in \mathbb{Z}^*_{\varphi(n)} = \{x : 1 \leq x \leq \varphi(n) - 1 : \gcd(e, \varphi(n)) = 1\}$$

The value e is also called **RSA public exponent**. Long story short, the RSA public key is a couple

$$k_{pub} = (n, e)$$

Private key

The first part of the RSA's private key is made of the two prime numbers p and q. The private key also contains Euler's totient function $\varphi(n)$ which can be computed using p and q as

$$\varphi(n) = (p - 1) \cdot (q - 1)$$

The last part of the private key is the multiplicative inverse, modulo $\varphi(n)$ of the public exponent e, namely

$$d = e^{-1} \mod \varphi(n)$$

Because e has been chosen in $\mathbb{Z}^*_{\varphi(n)}$, d also belongs to $\mathbb{Z}^*_{\varphi(n)}$. In a nutshell, the private key is a tuple

$$k_{priv} = (p, q, \varphi(n), d)$$

Let us highlight why all components of the private key must be kept secret. The multiplicative inverse d is easy to compute with $\varphi(n)$ (e.g., using the extended Euclid algorithm) and hard without, then $\varphi(n)$ has to be kept secret. At the same time, $\varphi(n)$ is easy to compute using having p and q since we use the formula $(p-1)(q-1)$. For this reason, p and q have to be kept secret. Note that p and q are hard to compute given only the public information n because the integer factorisation problem is hard, hence also $\varphi(n)$ and d are hard to compute given only the public key (n, e).

7.2.2 Encryption and decryption

Encryption function

The encryption function is

$$c = \mathbb{E}_{k_{pub}}(m) = m^{e \mod \varphi(n)} \mod n$$

Note that the exponent is always computed modulo $\varphi(n)$ however we will sometimes write just the exponent for brevity. Also note that, since the message is computed modulo n, hence the message has a limited length. This means that a message with more bits than the number of bits of n has to be split in chunks that are independently encrypted. In other words, the message is split in chunks which are interpreted as numbers belonging to $\mathbb{Z}_n$.

Decryption function

The decryption function is

$$m = \mathbb{D}_{k_{priv}}(c) = c^{d \mod \varphi(n)} \mod n$$

Proof

In a public key cryptosystem we have to ensure that what is encrypted with one's public key can only be decrypted with that person's private key, namely that

$$\mathbb{D}_{k_{priv}}(\mathbb{E}_{k_{pub}}(m)) = m$$

In the case of RSA, we also have to prove that, what's encrypted with one's private key can be decrypted only with that person's public key, namely that

$$\mathbb{D}_{k_{pub}}(\mathbb{E}_{k_{priv}}(m)) = m$$

Thanks to RSA's properties, demonstrating the former property is the same as demonstrating the latter since we can write

$$\begin{aligned}
\mathbb{D}_{k_{priv}}(\mathbb{E}_{k_{pub}}(m)) &= \mathbb{D}_{k_{priv}}(m^{e \mod \varphi(n)} \mod n) \\
&= (m^{e \mod \varphi(n)})^{d \mod \varphi(n)} \mod n \\
&= m^{ed \mod \varphi(n)} \mod n \\
&= (m^{d \mod \varphi(n)})^{e \mod \varphi(n)} \mod n \\
&= \mathbb{D}_{k_{pub}}(m^{d \mod \varphi(n)} \mod n) \\
&= \mathbb{D}_{k_{pub}}(\mathbb{E}_{k_{priv}}(m))
\end{aligned}$$

Since we've shown that the two cases are equivalent, we can focus on the former, hence we want to show that

$$\mathbb{D}_{k_{priv}}(\mathbb{E}_{k_{pub}}(m)) = m$$

namely that

$$(m^{e \mod \varphi(n)})^{d \mod \varphi(n)} \equiv m \mod n$$

First, let us say that, since we have defined d as the multiplicative inverse of e, i.e., as

$$d = e^{-1} \mod \varphi(n)$$

then multiplying d by e gives us 1 because

$$d \cdot e \equiv e^{-1} \cdot e \equiv 1 \mod \varphi(n)$$

Because $de \equiv 1 \mod \varphi(n)$ then we can write

$$m^{de \mod \varphi(n)} \equiv m^{1+t\varphi(n)} \equiv m \cdot m^{t\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^t \mod n$$

Since the message is independent from the public modulo n, then we have to consider two different cases

- When m and n are coprime.
- When m and n are not coprime.

Let's consider the first case. Thanks to Euler's theorem we know that, given two integers $a \in \mathbb{N}$ and $n \in \mathbb{N} \setminus \{0\}$, if they are coprime (i.e., $\gcd(a, n) = 1$), then it holds

$$a^{\varphi(n)} \equiv 1 \mod n$$

Since the assumption holds (m and n) are coprime, then we can write

$$\begin{aligned} m^{de \mod \varphi(n)} &\equiv m^{1+t\varphi(n)} \\ &\equiv m \cdot m^{t\varphi(n)} \\ &\equiv m \cdot (m^{\varphi(n)})^t \mod n \\ &\equiv m \cdot (1)^t \mod n \\ &\equiv m \cdot 1 \mod n \\ &\equiv m \mod n \end{aligned}$$

Let's now focus on the second scenario, i.e., when $\gcd(n, m) \neq 1$. In particular, let us assume, without loss of generality, that the greatest common divisor between n and m is p, hence that the message can be written as $m = u \cdot p$ for some $u \in \mathbb{Z}_n^*$. Because q isn't a factor or m, then we can state that $\gcd(q, m) = 1$ hence we can apply Euler's theorem and write

$$\begin{aligned} m^{\varphi(q)} &\equiv m^{q-1} \mod n && q \text{ is prime} \\ &\equiv 1 \mod n && a \text{ Euler's theorem} \end{aligned}$$

If we rise $m^{\varphi(n)} \mod q$ to $t \in \mathbb{Z}_{\varphi(n)}^*$ we obtain

$$\begin{aligned} (m^{\varphi(n)})^t &\equiv (m^{(q-1)(p-1)})^t \mod q \\ &\equiv (m^{q-1})^{(p-1)t} \mod q \\ &\equiv (1)^{(p-1)t} \mod q \\ &\equiv 1 \mod q \end{aligned}$$

This means that we can write $(m^{\varphi(n)})^t$ as

$$(m^{\varphi(n)})^t = 1 + sq$$

This means that we can write

$$\begin{aligned}
m^{de \mod \varphi(n)} &\equiv m \cdot (m^{\varphi(n)})^t \mod n \\
&\equiv m \cdot (1 + sq) \mod n \\
&\equiv m + msq \mod n \\
&\equiv m + msq \mod n && m = u \cdot p \\
&\equiv m + (us)(pq) \mod n \\
&\equiv m + (us)n \mod n && n = p \cdot q \\
&\equiv m \mod n
\end{aligned}$$

7.2.3 Implementation details

The core of the RSA cryptosystem is raising the message to a power. This means that we should try to optimise the algorithm for raising a number to a power. One of the fastest algorithms for computing the power of a number is the **square and multiply algorithm** (B.1). The complexity of this algorithm is

$$T(n)_{square\ and\ multiply} = \lceil \log_2 \varphi(n) \rceil + HW(e)$$

where $HW(e)$ is the Hamming Weight of the public exponent (i.e., the number of 1s in the binary representation of e). This means that the lower the number of 1s in the public exponent, the better it is. Typically, the exponent is chosen among between the values of the form $2^k + 1$ (checking that $e \in \mathbb{Z}^*_{\varphi(n)}$) which are the numbers that have 1s only in the most and least significant positions. Note that e belongs to the public key, which also contains the user-specific value n hence it's not a problem if e is a well-known value. Also note that, even if e is small (e.g., $e = 3$), the secret exponent d will have, with high probability, approximately the same bit-length of $\varphi(n)$ with a non-predictable Hamming weight.

Having a small Hamming weight allows us to reduce the number of multiplications we have to compute, however, we still have to compute $\log_2 \varphi(n)$ values.

Decryption function speed up

The decryption function can be sped up by using the Chinese remainder theorem (A.7.3). In particular, if we want to compute

$$m = c^{d \mod \varphi(n)} \mod n$$

we can calculate

$$m_p = c^{d \mod \varphi(p)} \mod p = c^{d \mod p-1} \mod p$$

and

$$m_q = c^{d \mod \varphi(q)} \mod q = c^{d \mod q-1} \mod q$$

Now we can use the Chinese remainder's theorem, in fact, we can write

$$m = c^{d \mod \varphi(n)} \mod n \iff \begin{cases} m = m_p \mod p \\ m = m_q \mod q \end{cases} \iff \begin{cases} m \equiv c^{d \mod \varphi(p)} \mod p \\ m \equiv c^{d \mod \varphi(q)} \mod q \end{cases}$$

Now we can apply the formula to swiftly compute m,

$$m \equiv \big(q(q^{-1} \mod p)m_p + p(p^{-1} \mod q)m_q\big) \mod n$$

$$T_{\mathbb{E}}(n) = \mathcal{O}\left(\frac{3}{2}(t-1)\cdot \text{mul}_t\right)$$

$$T_{\mathbb{D}}(n) = \mathcal{O}\left(2\frac{3}{2}\frac{t-1}{2}\cdot \text{mul}_{\frac{t}{2}}\right) = \mathcal{O}\left(\frac{3}{8}(t-1)\cdot \text{mul}_t\right)$$

7.2.4 Mathematical security

RSA is used in many contexts, hence it's important to check why it's secure. As we've already assessed, we can't achieve perfect security, however, we can still get computational security. The security of RSA is based on the integer factorisation problem in fact, given the public modulo n, the only way of breaking RSA is factoring n, namely finding p and q. In particular, the security of RSA is related to two problems:

- The **factoring problem**. Given an integer n obtained as the product of only two prime factors p and q, namely $n = p \cdot q$, the factoring problem consists of finding p and q.
- The **RSA problem**. Given a ciphertext $c = m^e \mod n \in \mathbb{Z}_n$ with $n = p \cdot q$ and $e \in \mathbb{Z}^*_{\varphi(n)}$, the RSA problem consists of finding $m \in \mathbb{Z}_n$.

If one can solve the factoring problem then he or she can also solve the RSA problem since he or she can factor n as $n = p \cdot q$ and then compute $\varphi(n) = (p-1)(q-1)$ and d. On the other hand, if one can solve the RSA problem, we don't know if he or she can solve the factoring problem. In other words, given an algorithm for solving the factoring problem, then it can also be used for solving the RSA problem. Contrary, if we have an algorithm for solving the RSA problem, we don't know if it's possible to use it for solving the factoring problem.

Factoring problem

Since solving the factoring problem allows us to solve the RSA problem, hence to break RSA, we should understand the complexity of the factoring problem. First, let us introduce the following function which will be used to compute the complexity of the factoring problem:

$$\mathcal{L}_n(\alpha, \beta) = \exp\left(\big(\beta + o(1)\big)(\log n)^{\alpha}\big(\log(\log n)\big)^{1-\alpha}\right)$$

with $a \in [0, 1]$. This is a sub-exponential function which grows slower than an exponential and faster than a linear function. In particular,

- For $\alpha = 0$ we have

  $$\mathcal{L}_n(0, \beta) = \exp\left(\big(\beta + o(1)\big)(\log n)^{0}\big(\log(\log n)\big)^{1-0}\right) = \exp\left(\big(\beta + o(1)\big)\big(\log(\log n)\big)\right)$$

  This means that an algorithm running in $\mathcal{O}(\mathcal{L}_n(0, \beta))$ has a polynomial complexity in the number of bits of n, i.e., in $\log(n)$.

- For $\alpha = 1$ we have

$$\mathcal{L}_n(1,\beta) = \exp\left(\big(\beta + o(1)\big)(\log n)^1\big(\log(\log n)\big)^{1-1}\right) = \exp\left(\big(\beta + o(1)\big)(\log n)\right)$$

  This means that an algorithm running in $\mathcal{O}(\mathcal{L}_n(1,\beta))$ has exponential complexity in the number of bits of n, namely $e^{\log n} = n$.

There exist different algorithms for factorising a number, each working for numbers with a certain number of bits. The most used algorithms are:

- **Trivial division**. This algorithm tries every prime number up to $\sqrt{n}$ and checks if it's a factor of n. Trivial division practically works for integers with up to 12 digits (i.e., numbers smaller than 10^{12}). The complexity of this algorithm is

$$T_{trivial\ division}(n) = \mathcal{L}_n(1,1)$$

  hence it is exponential.

- **Trivial division variation**. This algorithm generates a value $x = p_1 p_2 p_3 \dots p_r$ of r primes and computes $\gcd(n, x)$ for finding the largest prime factor in n.

- **Elliptic curve method**. The Elliptic curve method has a sub-exponential complexity of

$$T_{elliptic\ curve}(n) = \mathcal{L}_p\left(\frac{1}{2}, c\right)$$

  where p is the smallest factor of n. This algorithm is appropriate for numbers whose smallest factor is smaller than 2^{50} (i.e., when $p < 2^{50}$).

- **Quadratic sieve**. The quadratic sieve is the fastest method for factoring integers with sizes between 2^{400} and 2^{300} and has a complexity of

$$T_{quadratic\ sieve}(n) = \mathcal{L}_n\left(\frac{1}{2}, 1\right)$$

- **General Number Field Sieve** (GNFS). The General Number Field Sieve algorithm is used for numbers bigger than 2^{300} and has a complexity of

$$T_{GNFS}(n) = \mathcal{L}_n\left(\frac{1}{3}, 1.923\right)$$

Recommended key lengths

Since RSA is widely used, we have to understand the magnitude of the values used as a key to consider the algorithm secure. The security of RSA is based on the possibility of factorising a number. Currently, the largest numbers that have been factorised are 768-bit numbers, hence we must use a public modulo encoded with more than 768 bits. More precisely, it's recommended to use:

- At least 2048-bit numbers for n to ensure medium-term security.
- At least 4096-bit numbers for n to ensure long-term security.

It's also important to compare RSA with AES, the most used symmetric key encryption system, to highlight that the latter requires way fewer bits to ensure the same security margin as RSA. These results are shown in Table 7.1.

Validity period	Security margin	Equivalent symmetric cypher	RSA
2010	80 bits	Double DES	1024 bits
2010 to 2030	112 bits	Triple DES	2048 bits
after 2030	128 bits	AES-128	3072 bits
much after 2030	192 bits	AES-192	7680 bits
much much after 2030	256 bits	AES-256	15360 bits

Table 7.1: A table that shows, for each RSA key length, the period in which it should be considered secure, its security margin and the equivalent symmetric cypher required to achieve the same security margin.

Security lemmas

The security of RSA and the relationship between the RSA problem and the factorisation problem is assessed by a set of lemmas.

Lemma 7.1 (RSA Security 1)**.** *The RSA problem is no harder than the factoring problem.*

Proof. By assumption, we know how the factors of n, hence we can compute $\varphi(n)$ as

$$\varphi(n) = (p-1)(q-1)$$

Given $\varphi(n)$ it's easy to compute d, using the Fermat's theorem, as

$$d \equiv e^{-1} \equiv e^{\varphi(\varphi(n))-1} \mod \varphi(n)$$

As the last step, d can be used to raise $c = m^e \mod n$ to the power of d to obtain m and solve the RSA problem. Namely

$$c^d \equiv (m^e)^d \equiv m^{ed} \equiv m \mod n$$

□

We've shown that if we have a program to solve the factoring problem, we can use it to easily solve the RSA problem. It would be interesting to also understand how much easier it is to solve the RSA problem with respect to the factorisation problem. The following lemma states an important result in this sense.

Lemma 7.2 (RSA Security 2)**.** *Knowing the private exponent d of an RSA private key, corresponding to the public key (n, e), it's still efficiently possible to factor the public modulus n.*

Proof. Since we have chosen d such that

$$e \cdot d \equiv 1 \mod \varphi(n)$$

then we can write

$$ed - 1 = s \cdot \varphi(n) = s(p-1)(q-1)$$

because the division by $\varphi(n)$ has remainder 1. If we pick $x \in \mathbb{Z}_n^*$, then we know that

$$\begin{aligned} x^{ed-1 \mod \varphi(n)} &\equiv x^{s\cdot\varphi(n) \mod \varphi(n)} \mod n \\ &\equiv x^{0 \mod \varphi(n)} \mod n \\ &\equiv 1 \mod n \end{aligned}$$

If we remember that p and q are prime numbers, then we know that $p-1$ and $q-1$ are even numbers, hence $ed-1 = s(p-1)(q-1)$ is also an even number. This means that we can compute

$$y = \sqrt{x^{ed-1}} = x^{\frac{ed-1}{2}}$$

Since $y = x^{\frac{ed-1}{2}}$, then we can write

$$y^2 \equiv 1 \mod n \iff y^2 - 1 \equiv 0 \mod n$$

As a result, $y^2 - 1$ can be divided by n. We can also write

$$y^2 - 1 \equiv (y-1)(y+1) \mod n$$

hence

$$(y-1)(y+1) = s \cdot n$$

Then, a factor can be computed as $\gcd(y-1, n)$ or $\gcd(y+1, n)$. As a result:

- If $y \not\equiv \pm 1 \mod n$, then we can obtain a factor of n as $\gcd(y-1, n)$.
- If $y = -1 \mod n$, then we have to repeat the procedure from the beginning with a different x.
- If $y = +1 \mod n$, then we have to repeat the procedure from the square root of y.

□

Lemma 7.3 (RSA Security 3)**.** *Given the RSA public modulus n of a RSA public key and $\varphi(n)$ in the corresponding private key, one can efficiently factor $\varphi(n)$.*

Proof. We can start by writing

$$\begin{aligned} \varphi(n) &= (p-1)(q-1) \\ &= pq - p - q + 1 \\ &= n - p - q + 1 \\ &= n - (p+q) + 1 \end{aligned}$$

Now we can rearrange the terms to obtain

$$-(p+q) = \varphi(n) - n - 1$$

Now we can build a system with the equality $n = pq$, namely

$$\begin{cases} -(p+q) = \varphi(n) - n - 1 \\ pq = n \end{cases}$$

From the second equation we can write $p = \frac{n}{q}$ and replace p it in the first one to obtain:

$$\begin{aligned} -(p+q) &= \varphi(n) - n - 1 \\ -\frac{n}{q} - q &= \varphi(n) - n - 1 \\ -\frac{n}{q} &= q + \varphi(n) - n - 1 \\ -n &= q^2 + (\varphi(n) - n - 1)q \\ q^2 + (\varphi(n) - n - 1)q + n &= 0 \end{aligned}$$

Solving this second-grade equation gives us one of the two prime factors p and we can obtain q as $q = \frac{n}{p}$. □

Since $\varphi(n)$ is a secret and the RSA cryptosystem is computationally expensive (because modular arithmetic is computationally expensive), one could think of reusing the same value of $\varphi(n)$ (i.e., the same modulus n) and different values of d and e, one for each key pair. In fact, if we choose a fixed value of n we can build custom hardware modules specifically optimised for that value of n. This isn't however secure as stated by the following lemma.

Lemma 7.4 (RSA Security 4)**.** *Given two RSA ciphertexts $c_1 = \mathbb{E}_{n,e_1}(m)$ and $c_2 = \mathbb{E}_{n,e_2}(m)$ one can efficiently recover the original message m if gcd$(e_1, e_2) = 1$.*

Proof. Assuming that $\gcd(e_1, e_2) = 1$, we can write the Bézout identity

$$t_1 e_1 + t_2 e_2 = \gcd(e_1, e_2) = 1$$

Since e_1 and e_2 are public, we can use the Euclidean algorithm to compute t_1 and t_2. Knowing t_1 and t_2 we can write

$$\begin{aligned} c_1^{t_1} \cdot c_2^{t_2} &\equiv (m^{e_1})^{t_1} \cdot (m^{e_2})^{t_2} \mod n \\ &\equiv m^{e_1 t_1} \cdot m^{e_2 t_2} \mod n \\ &\equiv m^{e_1 t_1 + e_2 t_2} \mod n \\ &\equiv m^1 \mod n && t_1 e_1 + t_2 e_2 = 1 \\ &\equiv m \mod n \end{aligned}$$

□

As we can see, using the same value of n, even when choosing a different value of e, allows an attacker to efficiently recover the message m. We've also noted that public exponents e are always chosen among the values that have a one in the most and least significant positions. If the exponent is small, the computational cost of encrypting a message is smaller, however, it can't be too small as stated by the following lemma.

Lemma 7.5 (RSA Security 5)**.** *The use of a small public exponent is unsafe in the case of multicast communication.*

Proof. Let us prove the lemma for $e = 3$ (which is small and in the form $2^n + 1$). Let $(n_1, 3)$, $(n_2, 3)$ and $(n_3, 3)$ three RSA public keys and m a message encrypted using all the keys. A passive attacker collects

$$\begin{cases} c_1 = m^3 \mod n_1 \\ c_2 = m^3 \mod n_2 \\ c_3 = m^3 \mod n_3 \end{cases}$$

Thanks to the Chinese remainder theorem we can compute

$$\begin{aligned} x &\equiv m^3 \mod (n_1 \cdot n_2 \cdot n_3) \\ &= \sum_{i=1}^{3} c_i \cdot \frac{N}{n_i} \cdot \left(\left(\frac{N}{n_i} \right)^{-1} \mod n_i \right) \end{aligned}$$

Because 3 is small, then m^3 is smaller than $n_1 \cdot n_2 \cdot n_3$, hence it holds

$$x = m^3$$

and we don't have to compute the modulo operation when applying the Chinese remainder theorem. This means that we can compute the cubic root of x to compute the message m, which is much faster since we are not considering the modulo operation. □

To solve this problem we can append some bits of random noise to each message. This allows to encrypt a different message every time, even when we pass the same message m to the encryption function. Another important security issue regards the primes p and q used to generate the public exponent.

Lemma 7.6 (RSA Security 6)**.** *Generating two moduli n_1 and n_2 sharing a prime exposes both of them to trivial factoring.*

Proof. If $n_1 = p \cdot q_1$ and $n_2 = p \cdot q_2$, then n_1 and n_2 have one divisor in common, which is p and is the greatest common divisor, hence we can compute p as $\gcd(n_1, n_2)$. Once we've found p we can divide n_1 and n_2 by p to obtain q_1 and q_2. □

As a result of Lemmas 7.4, 7.5 and 7.6 we can say that a RSA cryptosystem should adhere to the following rules:

- Each message should contain, at its end, a random value so that the same message is encrypted to a different ciphertext even when using the same key pair.
- Very small public exponents e should be avoided. Usually, the value chosen is $e = 65537 = 2^{16} + 1$.
- The primes p and q used for generating n should be obtained from a sound and secure PRNG initialised with a sufficiently long seed (i.e., at least 128 bit long).

7.2.5 Weaknesses against chosen ciphertext attacks

The RSA cryptosystem is not secure against chosen ciphertext attacks (CCAs), namely against attacks where the attacker can use an oracle that uses the victim's key pair to decrypt a chosen ciphertext. In particular, an attacker can:

1. Intercept a ciphertext $c = m^e \mod n$ from which he or she wants to obtain the message m.
2. Generate a message $x \in \mathbb{Z}_n$.
3. Compute $c' = (c \cdot x^e) \mod n$.
4. Send c' to the oracle. Since the oracle decrypts the value sent by the attacker, the attacker obtains:

$$\begin{aligned} m' = \mathbb{D}(c') &= (c')^d \mod n \\ &= (c \cdot x^e)^d \mod n \\ &= (c^d \cdot x^{ed}) \mod n \\ &= (c^d \cdot x) \mod n \end{aligned}$$

   Since the attacker knows the multiplicative inverse of x (he or she has chosen x), he or she can compute $c^d \mod n$ by multiplying $(c^d \cdot x) \mod n$ by x^{-1}.

Note that an attacker is also able to **recover the hidden secret key**. We can secure RSA against chosen ciphertext attacks using a function ϕ which is applied to the message before encrypting it. In other words, we compute $\mathbb{E}(\phi(m))$ instead of $\mathbb{E}(m)$. The function ϕ must be bijective and is defined as

$$\phi : \{0,1\}^{N-l} \times \{0,1\}^l \to \{0,1\}^N$$

where

- $\{0,1\}^l$ is a random value.
- $\{0,1\}^{N-l}$ is the message to encrypt.
- $N = \lceil \log_w((n-1)+1) \rceil$

Moreover, the computation of ϕ^{-1} (i.e., ϕ's inverse should) be deterministic to allow the random value to be discarded. The function ϕ should also be designed such that

$$\phi(a \cdot b \mod n, r_{ab}) \not\equiv \phi(a, r_a) \cdot \phi(b, r_b) \mod n \tag{7.1}$$

Let us now understand why using the function ϕ before encrypting the message straightens the RSA cryptosystem. Say someone encrypts a message m to obtain

$$c = \big(\phi(m, r_m)\big)^e \mod n$$

Now an attacker can choose a random message x and compute

$$c' = \phi(c \cdot x^e) \mod n$$

The attacker can then send the ciphertext just obtained to the oracle which returns

$$\begin{aligned} m' = \mathbb{D}(c') &= \phi^{-1}\big((c \cdot x^e)^d\big) \mod n \\ &= \phi^{-1}(c^d \cdot x^{ed}) \mod n \\ &= \phi^{-1}(c^d \cdot x) \mod n \\ &= \phi^{-1}(\big(\phi(m, r_m)\big)^{ed} \cdot x) \mod n \\ &= \phi^{-1}(\phi(m, r_m) \cdot x) \mod n \end{aligned}$$

Because of (7.1) we have that

$$\phi^{-1}(\phi(m, r_m) \cdot x) \not\equiv \phi^{-1}(\phi(m, r_m)) \cdot \phi^{-1}(x) \mod n$$

hence we can't recover $\phi(m, r_m)$.

Optimised Asymmetric Encryption Padding

An example of function ϕ is the Optimised Asymmetric Encryption Padding (OAEP) which adds a random padding at the end of the message to encrypt. The OAEP can be used with whatever one-way trapdoor permutation (i.e., also to RSA). The OAEP scheme:

- **Adds randomness**. In particular, by adding some randomness, we ensure that the cryptoscheme is **semantically secure**, meaning that if we encrypt the same message m twice (i.e., we compute $c_1 = \mathbb{E}(m)$ and $c_2 = \mathbb{E}(m)$) we obtain two different ciphertext even when using the same key pair.
- **Prevents partial decryption**. Namely, an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation.

This means that OAEP solves both the problem of short public exponents and the vulnerability with respect to CCAs.

In practice, OAEP is used as follows:

$$\mathbb{E}(m) = f\Big(\{m.(0^{k_1} \oplus G(r))\}.\{r \oplus H(m.(0^{k_1} G(r)))\}\Big)$$

where:

- k_0 and k_1 are two numbers such that the computational effort of 2^{k_0} or 2^{k_1} is impossible (i.e., it's impossible to bruteforce).
- $G : \{0,1\}^{k_0} \to \{0,1\}^{N-k_0}$ and $H : \{0,1\}^{N-k_0} \to \{0,1\}^{k_0}$ are two hash functions.
- m is a message of $N - k_0 - k_1$ bits.
- r is a random bit-string of k_0 bits.
- 0^l is a sequence of l zeros.
- f is a N-bit trapdoor one-way permutation. An example could be RSA with $N = \lceil \log(n) \rceil$ with n public modulus.

An Optimised Asymmetric Encryption Padding can be implemented as a two-round Feistel network. Applying OAEP to RSA makes RSA secure with respect to chosen ciphertext attacks, as stated by the following theorem.

Theorem 7.1 (RSA-OAEP Security). *If we model the hash functions G and H as truly random functions then RSA-OAEP is proven to be secure against adaptive chosen ciphertext attacks and chosen plaintext attacks if the RSA assumption holds (i.e., the RSA problem and the factorisation problem are hard to solve).*

7.3 Discrete logarithm cryptosystems

Some cryptosystems are based, differently from RSA, on the discrete logarithm problem. Formally, we can define the problem as follows:

Definition 7.2 (Generalised Discrete Logarithm Problem). *Let $(G, \cdot)$ be a cyclic group with order $n = |G|$, where $g \in G$ is one of its generators (i.e., $G = \langle g \rangle$). The Generalised Discrete Logarithm Problem (GDLP) for the group G consists of, given $a \in \langle g \rangle$, finding the smallest positive integer $x \in \mathbb{Z}_n$ such that $g^x = a$. Such an integer x is the discrete logarithm of a to the base g and we write*

$$x = \log_g^D a$$

or

$$x = ind_g a$$

Note that, even if we are talking about discrete logarithms, the same properties of logarithms hold. In particular

$$\log_g^D(a \cdot b) = \log_g^D(a) + \log_g^D(b) \mod n$$

because if we take $x = \log_g^D(a) \mod n$ and $y = \log_g^D(b) \mod n$, namely such that $g^x = a$ and $g^y = b$, then we have

$$a \cdot b = g^x \cdot g^y = g^{x+y} \mod n$$

This means that

$$x + y = \log_g^D(a \cdot b) \mod n$$

hence we have

$$\log_g^D(a) + \log_g^D(b) = \log_g^D(a \cdot b) \mod n$$

Moreover, given $a \in G$ and a generator $g_1 \in G$ different from g we have

$$\log_g^D(a) \cdot \log_{g_1}^D(g) = \log_{g_1}^D(a) \mod n$$

7.3.1 Choice of the group

Not every group can be used for cryptosystems based on the discrete logarithm problem. In particular, a group is secure if the generalised discrete logarithm problem is hard to solve on that group. Let us consider an example to understand what groups shouldn't be used for GDLP-based cryptosystem. Take the additive group $(\mathbb{Z}_{19}, +)$. One of its generators is $g = 2$ (but we could take any of its elements but 0 because 19 is prime). Now take a random element $a = 15 \in G$. Finding the discrete logarithm means finding $x \in G$ such that $2^x = a$ but, since we are considering an additive group, 2^x means $2 \cdot x$ since we are applying x times the operation $+$. This means that we have

$$2 \cdot x = 15 \mod 19$$

hence we want to compute

$$x = 15 \cdot 2^{-1} \mod 19$$

As a result, computing x has the same complexity of computing the inverse of 2 in $\mathbb{Z}_{19}$. In general, since 2 and 19 are coprime, the multiplicative inverse can be found using the Euclidean algorithm, which has polynomial complexity, hence the discrete logarithm has a polynomial complexity. This means that using an additive group is not secure, hence we **should never use an additive cyclic group** $(\mathbb{Z}_p, +)$ **for a GDLP-based cryptosystem**. Instead, we should use:

- **Subgroups of a generic multiplicative finite field** $(\mathbb{F}_{p^m}^*, \cdot)$ where $m \geq 1$, p is prime, $p^m \geq 2^{1024}$ and $n = |G|$ is a prime number. Solving the discrete logarithm problem over this groups has a sub-exponential complexity, namely

  $$T_{GDLP, finite\ field}(|G|) = \mathcal{O}(L_{|G|}(\alpha, \beta))$$

  with $0 < \alpha < 1$.

- **Subgroups of the group of elliptic curve points defined over a finite field**. Solving the discrete logarithm problem over this groups has an exponential complexity, namely

  $$T_{GDLP}(|G|, elliptic\ curve) = \mathcal{O}(L_{|G|}(1, \beta))$$

Since solving the GDLP is harder when using elliptic curves, then, by using the latter groups, we can use smaller parameters with respect to the former groups, while ensuring the same level of security. This means that, if we want to ensure the same security margin we can use elliptic curves with smaller parameters or multiplicative groups with larger parameters.

7.3.2 Discrete logarithm problems

As for now we have only talked about the discrete logarithm problem, however we can use three different problems in cryptography:

- The **discrete logarithm problem** (DLP).
- The **Diffie-Hellman problem** (DHP). Given $a = g^x \in G$ and $b = g^y \in G$ with $x, y \in \mathbb{Z}_n$, solving the Diffie-Hellman problem means computing $c = g^{xy} \in G$.
- The **decisional Diffie-Hellman problem** (DDHP). Given $a = g^x \in G$, $b = g^y \in G$ and $c = g^z \in G$ with $x, y, z \in \mathbb{Z}_n$, solving the Diffie-Hellman problem means establish if $z \equiv xy \mod n$.

We can compare the three problems above, similarly to what we did when analysing the integer factorisation and the RSA problems. For starters,

Lemma 7.7 (Discrete logarithm problem and Diffie-Hellman problem)**.** *The Diffie-Hellman problem is no harder than the discrete logarithm problem.*

This means that we can use a solution for the discrete logarithm problem to efficiently solve the Diffie-Hellman problem, or equivalently that solving DLP is a sufficient requirement to solve DHP. In fact, if given g^z we can find z, then we can also find x and y given $a = g^x$ and $b = g^y$. Having x and y we can also compute g^{xy} and compute c.

On the other hand, we don't know if, by solving the Diffie-Hellman problem, we can efficiently solve the discrete logarithm problem. We only know that for Gap-groups the two problems are equivalent but this is not true in general.

We can also compare the Diffie-Hellman problem with the decisional Diffie-Hellman problem.

Lemma 7.8 (Diffie-Hellman problem and decisional Diffie-Hellman problem)**.** *The decisional Diffie-Hellman problem is no harder than the Diffie-Hellman problem.*

This means that we can use a solution to the Diffie-Hellman problem for solving the decisional Diffie-Hellman problem, or equivalently that solving DHP is a sufficient requirement for solving DDHP. In fact, if given $a = g^x \in G$, $b = g^y \in G$ and $c = g^z$ we can compute $\gamma = g^{xy}$, then we can check

$$\begin{aligned} \gamma &= c \\ g^{xy} &= g^z \\ xy &= z \end{aligned}$$

On the other hand, only for trivial groups we can show that DDHP is equivalent to DHP. In particular, there are groups where the DDHP is known to be easy (if you have a pairing map similar to the ones employed in identity based cryptosystems), but the DHP is conjectured to be hard. Moreover, in multiplicative groups of non trivial finite fields and in cyclic groups of elliptic curve points, the decisional Diffie-Hellman problem is assumed to be hard.

7.3.3 Diffie-Hellman key exchange

The Diffie-Hellman protocol allows two parties, say Alice and Bob, to agree on a shared key by exchanging messages on a public channel. The protocol works as follows:

1. Alice and Bob publicly agree on a group $(G, \cdot)$ with cardinality $n = |G|$ and on a generator g of G (i.e., $G = \langle g \rangle$).
2. Alice picks a value $k_{priv,A} \in \mathbb{Z}_n \setminus \{0, 1\}$.
3. Alice computes $k_{pub,A} = g^{k_{priv,A}} \in G$.
4. Bob picks a value $k_{priv,B} \in \mathbb{Z}_n \setminus \{0, 1\}$.
5. Bob computes $k_{pub,B} = g^{k_{priv,B}} \in G$.
6. Alice sends $k_{pub,A}$ to Bob.
7. Bob sends $k_{pub,B}$ to Alice.
8. Alice computes $k_{shared} = (k_{pub,B})^{k_{priv,A}} = (g^{k_{priv,B}})^{k_{priv,A}} = g^{k_{priv,B}k_{priv,A}} \in G$.
9. Bob computes $k_{shared} = (k_{pub,A})^{k_{priv,B}} = (g^{k_{priv,A}})^{k_{priv,B}} = g^{k_{priv,A}k_{priv,B}} \in G$.

Authentication problem

The Diffie-Hellman protocol is not secure against man-in-the-middle attacks, in fact a third party Chuck can establish a key with Bob, pretending to be Alice and a key with Alice, pretending to be Bob. Alice sends messages to Chuck thinking she's talking to Bob and Chuck reads and forwards the message to Bob which thinks that the message is coming from Alice. This happens because each party is not authenticated (i.e., there is no way to tell if a certain g^a belongs to a party). To solve this problem we can use long term RSA key pairs which allow each user to authenticate the other part. In particular, Alice can send its value g^a signed with her private key so that Bob can verify that g^a really belongs to Alice (and Bob does the same with g^b, i.e., he signs g^b with his private key). One might now wonder why don't we just use RSA to exchange a shared secret instead of using RSA as an overlay for the Diffie-Hellman key agreement protocol. The reason is that RSA alone does not ensure forward secrecy. A system ensures forward secrecy if, when broken, it's not possible to read all previous messages. If one breaks RSA and we had exchanged the shared secret directly using RSA, then all the messages encrypted with that shared secret would be revealed. On the other hand, if we use RSA only to authenticate the Diffie-Hellman protocol, then breaking RSA would not allow an attacker to obtain the key agreed using the Diffie-Hellman protocol.

Practical groups

The Diffie-Hellman protocol uses one of the subgroups of the multiplicative cyclic group

$$(\mathbb{Z}_p^*, \cdot)$$

where $\mathbb{Z}_p^*$ is the set of invertible numbers modulo p. Since p is prime, the number of elements in $\mathbb{Z}_p^*$ is $\varphi(p) = p-1$. But because p is prime, then $p-1$ is not prime, hence the $(\mathbb{Z}_p^*, \cdot)$ has many subgroups, which are the groups $(\mathbf{H}_{d_i}, \cdot)$ where d_i is one the divisors of $\varphi(p)$ and $\mathbf{H}_{d_i}$ has d_i elements. Among all subgroups, we should choose those with a large number of values, namely a subgroup for which $|\mathbf{H}_{d_i}| = d_i$ is large enough.

Usually, two prime numbers p_1 and p_2 are generated in such a way that

- $p_1 \geq 2^{160}$,
- $p = 2p_1p_2 + 1$ is prime,
- $p \geq 2^{1024}$.

Given a generator $\alpha \in \mathbb{Z}_p^*$, a generator g of a subgroup $\mathbf{H}_{p_1}$ of $\mathbb{Z}_p^*$ is generated as

$$g = \alpha^{\frac{p-1}{p_1}}$$

7.4 The ElGamal cryptosystem

The ElGamal cryptosystem is a discrete logarithm-based cryptosystem. Differently from the RSA cryptosystem, ElGamal has two different schemes,

- one for encryption and decryption, and
- one for signing and verifying a signature.

The RSA cryptosystem uses the same functions for both operations, in fact

- $\mathbb{E}$ is used with the receiver's public key for encryption and with the signer's private key for signing,
- $\mathbb{D}$ is used with the receiver's private key for decryption and with the signer's public key for signature verification.

For both operations, the ElGamal cryptosystem assumes that there exist a publicly known cyclic group $(G, \cdot)$ with $n = |G|$ possibly prime and $g \in G$ generator of the groups (i.e., $G = \langle g \rangle$). We've already seen that if the group is large enough, the discrete logarithm problem, the Diffie-Hellman problem and the decisional Diffie-Hellman problem are computationally hard. In particular, this problem is hard in,

- the multiplicative group and sub-groups of finite fields, and
- the additive groups and sub-groups of elliptic curves.

This cryptosystem is not as popular as RSA because of two reasons:

- The RSA cryptosystem doesn't increase the size of the message when it's encrypted. This is because the encryption function rises the message (which is modulo n) to the exponent e and then computes the modulo n, again. As a result, the encrypted message is again modulo n, hence can be written in the same number of bits of the message. On the other hand, ElGamal performs an expansion. In particular, the ciphertext is made of two numbers of the same size of the plaintext (hence the ciphertext has double the size of the plaintext).
- The RSA computational performance is better than ElGamal's. In particular, ElGamal is roughly two times slower than RSA because its output is twice as long.

7.4.1 Key pair

Let's take a cyclic multiplicative group $(G, \cdot)$ with order $n = |G|$ possibly prime and generator $g \in G$. The **private key** is a number s in $\mathbb{Z}_n$ (i.e., $s \in \{0, \dots, n-1\}$), namely

$$k_{priv} = s \in \mathbb{Z}_n$$

The **public key** is a tuple

$$k_{pub} = (n, g, g^s)$$

This means that the public key exhibits every information about the public group (i.e., n and g). The cardinality n and the generator g aren't strictly speaking part of the public key since they are known when choosing the group but they are added to the key for ease of use. The third component $g^s \in G$ is the actual public key. If the discrete logarithm problem is hard, then it's computationally hard to obtain s, which is the private key, from the public key, since it would require to solve the discrete logarithm problem.

7.4.2 Encryption and decryption

The encryption transformation works as follows:

1. Pick a message m and encode it such that $m \in G$. For instance, if $G = \mathbb{Z}_p$, then we can encode m in binary and then ensure that the number of bits is less than the number of bits of p.

2. Pick a random number
$$l \in \mathbb{Z}_n^*$$
namely l should be an invertible number modulo n. This number must change every time we encrypt something and should be obtained using a secure random number generator.

3. Compute
$$\gamma = g^l \in G$$
Note that this is a modular exponentiation since γ must be an element of the group G.

4. Compute
$$\delta = m \cdot (g^s)^l \in G$$
This is also a modular exponentiation since δ must be an element of the group G.

5. The encrypted message is the tuple
$$c = \mathbb{E}_{k_{pub}}(m) = (\gamma, \delta)$$

The decryption transformation works as follows:

1. Receive the ciphertext $c = (\gamma, \delta)$.

2. Compute
$$m = \mathbb{D}_{k_{priv}}(c) = \gamma^{n-s} \cdot \delta$$

Note that the decryption done in this way allows us to save some computational effort. In fact, we can compute m as

$$\begin{aligned} m &= \delta \cdot \gamma^{-s} \\ &= m \cdot (g^s)^l \cdot (g^l)^{-s} \\ &= m \cdot g^{sl} \cdot g^{-sl} \\ &= m \end{aligned}$$

However, computing γ^{-s} means computing a multiplicative inverse, which is computationally heavy. Luckily, thanks to Lagrange's theorem (A.1) we know that if we rise a generator $a \in G$, with $n = |G|$ to it's cardinality n we get the identity 1, namely

$$a^n \equiv 1 \mod n$$

This means that we can write

$$\begin{aligned} \gamma^{n-s} &\equiv \gamma^n \cdot \gamma^{-s} \mod n \\ &\equiv (g^l)^n \cdot \gamma^{-s} \mod n \\ &\equiv (g^n)^l \cdot \gamma^{-s} \mod n && \text{Lagrange's theorem} \\ &\equiv 1^l \cdot \gamma^{-s} \mod n \\ &\equiv 1 \cdot \gamma^{-s} \mod n \\ &\equiv \gamma^{-s} \mod n \end{aligned}$$

hence we can use either γ^{n-s} or γ^{-s}. Since $s < n$, the exponent $n - s$ is positive and m can be obtained as the product of two numbers and we don't have to compute the multiplicative inverse of γ^s.

7.4.3 Security of the ElGamal cryptosystem

When presenting the Diffie-Hellman cryptosystem we introduced the Diffie-Hellman problem and used it to evaluate the security of the Diffie-Hellman cryptosystem. We can do the same for the ElGamal cryptosystem and define the ElGamal problem. The following lemma compares the two problems.

Lemma 7.9 (Equivalence of the Diffie-Hellman and ElGamal problems)**.** *The Diffie-Hellman problem is equivalent to the ElGamal problem.*

Another important lemma is the following, which explains the security of the ElGamal cryptosystem against a chosen plaintext attack.

Lemma 7.10 (Security of ElGamal against chosen ciphertext attacks)**.** *Assuming that the Diffie-Hellman problem is hard, then ElGamal is secure under a chosen plaintext attack (CPA). Namely, it is hard for the adversary, given the ciphertext, to recover the whole of the plaintext.*

The same isn't true for adaptively chosen ciphertext attacks as stated by the following lemma.

Lemma 7.11 (Security of ElGamal against adaptively chosen ciphertext attacks)**.** *ElGamal is not secure against an adaptively chosen ciphertext attack.*

Proof. To explain why, let us consider that there is a decryption oracle that can be queried with as many messages as the attacker wants, except for the ciphertext that is of interest to the attacker. Namely, the attacker can't decrypt the message he or she wants to but he or she can decrypt any other ciphertext. Assume that the attacker sees a ciphertext

$$c = (\gamma, \delta) = (g^l, m(g^s)^l)$$

and wants to decrypt it without knowing s. The attacker can create another ciphertext

$$c' = (\gamma, 2\delta)$$

and ask the oracle to decrypt the ciphertext c'. The oracle will return

$$m' = 2\delta \cdot \gamma^{n-s} = 2(\delta \cdot \gamma^{n-s}) = 2m$$

This means that the attacker can now divide m' by 2 to obtain m. □

To solve this problem, any implementation of ElGamal is slightly modified to deal with this type of attack. In particular, instead of using directly l and m, we compute the hash of $m||l \in G$ (where $||$ represents a composition such that $m||l$ belongs to G and such that m can be obtained even without knowing l) and we use $H(m||l)$ every time we have to use l or m. The resulting encryption function becomes

$$c = \mathbb{E}_{k_{pub}}(m, l) = (g^{H(m||l)}, (m||l) \cdot (g^s)^{H(m||l)})$$

The decryption function has to be modified, too. We have to compute

$$m' = \mathbb{D}(c)$$

with the non-modified function and then check that

$$c = \mathbb{E}(m', H(m'))$$

If this equality holds, we can recover m knowing that $m' = m||l$.

This scheme is only marginally less efficient than non-modified ElGamal. The details of the implementation are contained in the paper How to Enhance the Security of Public-Key Encryption at Minimum Cost.

7.4.4 Signature and verification

Signature and verification require use the same key-pair used for encryption and decryption but the functions used for signing an object and verifying a signature are different than those used for encrypting and decrypting a message. We also need an hash function $h : \{0,1\}^* \to \mathbb{Z}_n$ that takes an arbitrary long string of bits and computes a digest in $\mathbb{Z}_n$, with $n = |G|$ cardinality of the group. The hash function must be publicly known and it's used to make the signing process faster by signing the hash of a message instead of the message itself.

The **signature** transformation $\mathbb{S}_{k_{priv}}$ takes as input a message m to sign and a secret k_{priv} and outputs a signature that can be computed only by knowing the secret k_{priv},

$$\mathbb{S}_{k_{priv}} : G \to G$$

The signature procedure works as follows:

1. Pick a message m and encode it such that $m \in G$.
2. Pick a random number
$$l \in \mathbb{Z}_n^*$$
3. Compute
$$\gamma = g^l \in G$$
This is also a modular exponentiation since γ must be an element of the group G.
4. Compute
$$\delta = l^{-1} \cdot \big(h(m) - s \cdot h(\gamma)\big) \mod n$$
Differently form the encryption function, δ isn't necessarily an element of the group G but it's simply a number.
5. The signature of the message is the tuple
$$S = \mathbb{S}_{k_{priv}}(m) = (\gamma, \delta)$$

The **validation** (or verification) transformation $\mathbb{V}_{k_{pub}}$ takes as input signature S, a message m and the public key k_{pub} corresponding to the secret key k_{priv} used to generate the signature S and verifies if S is the correct signature for the message m, computed by the holder of k_{priv}. The verification procedure works as follows:

1. Receive $S = (m, S) = \big(m, (\gamma, \delta)\big)$.
2. Compute $h(m)$ and $h(\gamma)$.

3. Accept the signature only when

$$\mathbb{V}_{k_{pub}}(m, S) = true \iff (g^s)^{h(\gamma)} \cdot \gamma^\delta = g^{h(m)}$$

Let us now understand why the validation function correctly checks if the signature has been computed using k_{priv} related to k_{pub} on m. Namely, we want to show that

$$\mathbb{V}_{k_{pub}}\big(\mathbb{S}_{k_{priv}}(m)\big) = m \quad \forall m \in G$$

We can write

$$\begin{aligned}
(g^s)^{h(\gamma)} \cdot \gamma^\delta &= g^{s \cdot h(g^l)} \cdot (g^l)^{l^{-1} \cdot \left(h(m) - s \cdot h(g^l)\right)} \mod n \\
&= g^{s \cdot h(g^l)} \cdot g^{l \cdot l^{-1} \cdot \left(h(m) - s \cdot h(g^l)\right)} \mod n \\
&= g^{s \cdot h(g^l)} \cdot g^{h(m) - s \cdot h(g^l)} \mod n \\
&= g^{s \cdot h(g^l) + h(m) - s \cdot h(g^l)} \mod n \\
&= g^{h(m)}
\end{aligned}$$

Note that the receiver (i.e., the one that verifies the signature) can compute $g^{h(m)}$ since he or she knows g and h, which are public, and has received m.

Security of the signature and verification schemes

Let us consider the point of view of an attacker. An attacker (if we assume that m is sent in clear) knows

- g because it's public,
- h because it's public,
- n because it's public,
- m because it's sent in public,
- g^s because it's public,
- γ because it's sent on a public channel, and
- δ because it's sent on a public channel.

If the discrete logarithm is hard, then the ElGamal problem is hard, hence the signature and verification procedure are secure. Better said, if the Diffie-Hellman problem is hard, then the ElGamal problem is also hard, hence the signature and verification procedure are secure.

The security margin of ElGamal, compared with RSA and some symmetric cryptosystems is shown in Table 7.2.

Note that the size of the private key is also the size of the group G since $s \in \mathbb{Z}_n$, hence $s \in \{1, \ldots, n-1\}$ and the maximum value of s is n and s must be encoded with the same number of bits of n.

Note that if n is encoded with 256 bits, then the group has $n = 2^{256}$ elements. Since it's difficult to define a multiplicative group of a finite field with exactly that number of elements, then we can use a sub-group of a multiplicative group of a finite field. For instance, if we consider $\mathbb{Z}_p$ with p

Date	Security margin	Symmetric cypher	public key size [bit]	private key size [bit]
2010	80	TDES with 2 keys	1024	160
2011-2030	112	TDES with 3 keys	2048	224
$>$ 2030	128	AES-128	3072	256
$\gg$ 2030	192	AES-192	7680	384
$\ggg$ 2030	256	AES-256	15360	512

Table 7.2: ElGamal security margin.

encoded with 1024 bit, we can take $\mathbb{Z}_p^*$, which is cyclic and has $p-1$ elements. This group has a composite order since p is prime (hence $p-1$ is not), and for the Lagrange theorem, the factors of $p-1$ tell us the cardinality of the subgroups of $\mathbb{Z}_p^*$. We can use the public key size as $p-1$ to find one group with the cardinality cardinally.

7.4.5 ElGamal for digital signatures

The ElGamal cryptosystem is used in many standard signature libraries. For instance, ElGamal is used in the Digital Signature Algorithm (DSA) standard as part of the Digital Signature Standard (DSS). This means that the DSS contains the DSA which is based on the ElGamal cryptoscheme. In fact, the DSA is a variant of the ElGamal signature and verification scheme.

The DSA uses groups of the shape

$$(\mathbb{Z}_p^*, \cdot)$$

with p prime. In particular, DSA works with a sub-group $\mathbf{H}_q$ of $\mathbb{Z}_p^*$ with size q such that q is also prime. As a result, q divides $p-1$ because the subgroups of $\mathbb{Z}_p^*$ are all those with cardinality that divides $p-1$. The generator of the sub-group $\mathbf{H}_q$ or order q is called g and has order q (i.e., $g^q \equiv 1 \mod p$). The **public key** is the tuple

$$k_{pub} = (p, q, g, g^s)$$

and the **private key** is a number

$$k_{priv} = s \in \mathbb{Z}_q^*$$

The standard also defines

- $L = \log_2(p)$ which is the number of bits to encode p, and
- $N = \log_2(q)$ which is the number of bits to encode q.

These values are also standardised and the recommended values for L and N, respectively are

- 1024 and 160,
- 2048 and 224,
- 2048 and 256,
- 3072 and 256.

DSA also mandates that either SHA-224, SHA-256, SHA-384 or SHA-512 is used, depending on the size selected for the key-pair. Note that the difference between these functions is the size of the digest (i.e., of the output computed by the hash function). In particular,

- If we choose $(L, N) = (1024, 160)$ or $(L, N) = (2048, 224)$, we should choose at least SHA-224.
- If we choose $(L, N) = (3072, 256)$, we should choose at least SHA-256.

The standard also defines how to obtain the generator g of the group $\mathbf{G}$. The generator must be an element of $\mathbb{Z}_p^*$ with order q. Let us understand how can we find an element of a specific order. We know how to find a generator of a group, say h. If we raise this generator to $\frac{p-1}{q}$, we will obtain a number which has order equal to q. As a result, to find a generator we have to compute

$$g = h^{\frac{p-1}{q}}$$

We can take for instance $h = 2 \in \mathbb{Z}_p^*$. If the generator is equal to 1 we can try to compute a new generator using $h + 1$ (and keep incrementing h) until we find $g \neq 1$.

The actual signature transformation works as follows:

1. Pick a message m and hash it using H to obtain a value in $\mathbb{Z}_q \setminus \{1\}$,
$$H(m) \in \{2, \ldots, q-1\}$$
2. Pick a random value l in $\mathbb{Z}_q^*$,
$$l \in \mathbb{Z}_q^* = \{1, \ldots, q-1\}$$
3. Compute
$$\gamma = \left(g^l \mod p\right) \mod q$$
γ is an integer number (differently from the standard ElGamal scheme).
4. If $\gamma = 0$, go to step 2. This is required since by adding a $\mod q$, we might obtain $\gamma = 0$.
5. Compute
$$\delta = l^{-1} \cdot \left(H(m) + s \cdot \gamma\right) \mod q$$
δ is an integer number.
6. If $\delta = 0$, go to step 2.
7. Obtain the signature as the couple γ and δ,
$$S = \mathbb{S}_{k_{priv}}(m) = (\gamma, \delta)$$
8. Send
$$(m, S)$$

The validation check works as follows:

1. Receive
$$(m, S)$$
2. If $\gamma \notin \{1, \ldots, q-1\}$ or $\delta \notin \{1, \ldots, q-1\}$ reject the signature.
3. Compute $H(m)$.
4. Accept the signature if and only if
$$\mathbb{V}_{k_{pub}}(m, S) = true \iff \left(g^{u_1}(g^s)^{u_2}\right) \mod p \mod q = \gamma$$
where
   - $u_1 = H(m) \cdot \delta^{-1} \mod q$, and
   - $u_2 = \gamma \cdot \delta^{-1} \mod q$.

7.5 Elliptic curve cryptography

Elliptic curve cryptography is a cryptoscheme based on the discrete logarithm problem in the groups of the points on an elliptic curve. Elliptic curve cryptography is better than cryptoschemes based on the discrete logarithm problem in $\mathbb{F}_{p^m}$ because:

- It requires shorter keys to ensure the same level of security.
- Solving the discrete logarithm problem in the group of elliptic curve points is harder than solving it in $\mathbb{F}_{p^m}$. In fact, the best solution we know has an exponential complexity whereas the best solutions in other groups have a sub-exponential complexity.

Since elliptic curve cryptography is based on elliptic curves, we should start by defining them.

Definition 7.3 (Elliptic curve)**.** *Let $\mathbb{K}$ be a field of real numbers. Let us call*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

a Weierstrass equation with $x, y, a_i \in \mathbb{K}$. An elliptic curve over $\mathbb{K}$ is the set of solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ of a Weierstrass equation:

$$\mathbb{E}(\mathbb{K}) = \left\{(x, y) \in \mathbb{K} \times \mathbb{K} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, a_i \in \mathbb{K}\right\}$$

An elliptic curve is defined over a generic field $\mathbb{K}$, however we will consider only the fields that can be used for cryptography applications, which are the finite fields $\mathbb{F}_{p^m}$ with $p \geq 3$ or $\mathbb{F}_{2^m}$. Depending on the coefficients a_i of the Weierstrass equation, an elliptic curve can be:

- **Singular**. A singular admits at least one tangent in every point, but there exist point that have multiple tangents. Figure 7.1 shows an example of eliptic curve with a singularity in $(0, 0)$.
- **Non-singular**. In a non-singular curve, the tangent is well defined on every point of the curve. Geometrically, this means that the graph has no cusps, self-intersections, or isolated points.

Elliptic curve cryptosystems have to use non-singular curves for reasons that will be clear later.

7.5.1 Singularities

Field with characteristic greater than 3

Let us now consider elliptic curves over a finite field $\mathbb{F}_{p^m}$ with characteristic p greater than 3. In this setup, we can apply the substitution

$$\xi = x - \frac{a_2}{3}$$
$$\upsilon = y - \frac{a_1x + a_3}{2}$$

and call

$$a = \frac{1}{9}a_1^2 + a_4 \in \mathbb{F}_p$$
$$b = \frac{2}{27}a_2^3 - \frac{1}{3}a_2a_4a_6 \in \mathbb{F}_p$$

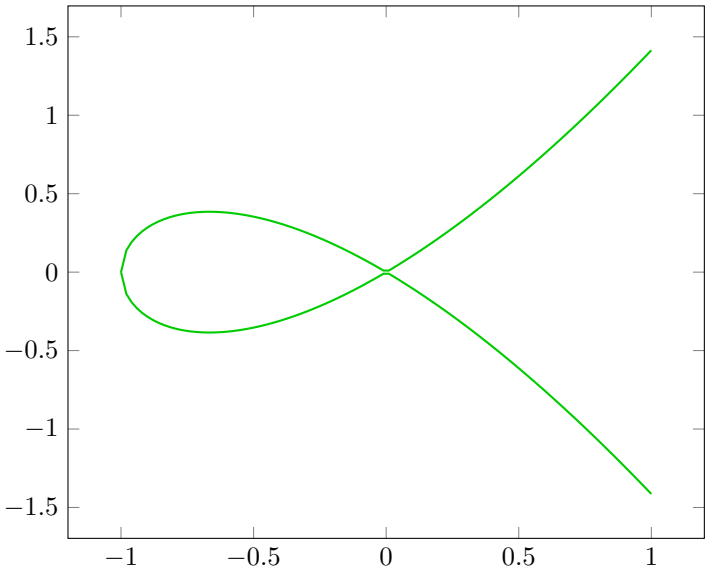


Figure 7.1: A singular elliptic curve.

to obtain a shorter version of the Weierstrass equation

$$\mathbb{E}(\mathbb{F}) : Y^2 = X^3 + aX + b \tag{7.2}$$

As we can see, the right-hand side is a function of only X, hence we can write

$$Y^2 = f(X)$$

Let us now derive both sides for X:

$$\frac{dY^2}{dX} = \frac{df(X)}{dX}$$

The right-hand side can be simply be written as $f'(X)$ to relax the notation. On the left-hand side, Y can be written as a function of X, in fact, from Equation 7.2 we can obtain $Y = \pm\sqrt{X^3 + aX + b}$, which is a function of X. As a result, we can write

$$\frac{dY(X)^2}{dX} = f'(X)$$

Let us focus on the derivative of $Y(X)^2$. This derivative can be computed as

$$\frac{dY(X)^2}{dX} = Y(X)\frac{dY(X)}{dX} + \frac{dY(X)}{dX}Y(X) = 2Y(X)\frac{dY(X)}{dX}$$

Replacing this result in the Weierstrass equation allows us to obtain

$$2Y(X)\frac{dY(X)}{dX} = f'(X)$$

If we now divide both sides of Equation 7.5.1 by $2Y$ we get

$$\frac{dY(X)}{dX} = \frac{f'(X)}{2Y(X)}$$

This equation shows that when $Y = f'(x) = f(x) = 0$, the derivative $\frac{dY(X)}{dX}$ isn't well defined. As a result, we have to pick functions such that

$$Y = f'(x) = f(x) \neq 0$$

to have a non-singular elliptic curve. Equivalently, we can say that $f(X) = X^3 + aX + b$ must not have multiple roots, namely it must hold

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

Fields with characteristic 2

The same reasoning can be applied to the fields $\mathbb{F}_{2^m}$. Assuming $a_1 \neq 0$, we can apply the substitutions

$$\mathbf{x} = a_1^2 x - \frac{a_3}{a_1}$$
$$\mathbf{y} = a_1^3 y - \frac{a_1^2 a_4 + a_3^2}{a_1^3}$$

and call

$$a = \frac{1}{9}a_1^2 + a_4 \in \mathbb{F}_{2^m}$$
$$b = \frac{2}{27}a_2^3 - \frac{1}{3}a_2 a_4 a_6 \in \mathbb{F}_{2^m}$$

to obtain

$$\mathbf{y}^2 + \mathbf{xy} = \mathbf{x}^3 + a\mathbf{x}^2 + b$$

In this case we only have to ensure that

$$b \neq 0$$

to have a non-singular curve.

7.5.2 Elliptic curve group

We want to give to the set of points on an elliptic curve the structure of a group. This means that we have to define

- an internal associative operation, which we will denote with the symbol $+$, and
- a neutral element, which we will denote with the symbol $\mathcal{O}$.

As a result, the group we are considering is

$$(\mathbb{E}(\mathbb{K}), +, \mathcal{O})$$

Neutral element

Let's start by defining the neutral element $\mathcal{O}$. The neutral element $\mathcal{O}$ is a point at infinity along the vertical direction. For this reason, the neutral element is also called **point at infinity**. In the Cartesian plane the point at infinity can be interpreted as a direction or a slope (e.g., all the vertical lines). The real nature of a point at infinity will be clear later on when we will represent the group in a space different from the Cartesian plane.

Composition law

The operation + is called **chord and tangent rule**.

Definition 7.4 (Chord and tangent rule)**.** *Let $P \in \mathbb{E}(\mathbb{K})$ and $Q \in \mathbb{E}(\mathbb{K})$ be two points on the same elliptic curve. Let $r = \overline{PQ}$ be the line that passes through P and Q, or the tangent to the curve passing through P if $P = Q$. The line r will intersect the elliptic curve in at most three points (two of which are P and Q) because $\mathbf{y}^2 = \mathbf{x}^3 + a\mathbf{x} + b$ is a polynomial of degree 3 and if we put it in a system with $\mathbf{y} = m\mathbf{x} + q$ (to find the intersections with a line) we get at most three distinct points. Note that two points might coincide. Let $R \in \mathbb{E}(\mathbb{K})$ be the third point on the curve that intersects r. Let $r' = \overline{R\mathcal{O}}$ be the vertical line that passes through R and the point at infinity $\mathcal{O}$. The line r' passes, for the same reasons seen before, for three points, two of which are R and $\mathcal{O}$. Let $S \in \mathbb{E}(\mathbb{K})$ be the third intersection point of r' on the curve. Then S is the composition of P and Q computed through the chord and tangent rule and we write*

$$S = P + Q$$

In practice, to compute the sum between two points P and Q on the curve we have to:

1. Draw the line r passing between P and Q or the tangent to the curve passing through P if $P = Q$.
2. Take a point R, different from P and Q, on the curve that intersects r.
3. Draw the line r' passing through $\mathcal{O}$ and R.
4. Take the point S, different from R on the curve that intersects r'.

The same process is represented in Figure 7.2. This graphical representation holds for a non-finite field $\mathbb{K}$. For finite fields $\mathbb{F}_{p^m}$, the curve points continue to have the same algebraic properties, although it is not possible to show a graphic representation of them.

Group with the chord and tangent rule

The group

$$(\mathbb{E}(\mathbb{K}), +, \mathcal{O})$$

with the chord and tangent rule is an abelian group. This means that the following properties hold:

- $\mathcal{O}$ is the neutral element.
$$\forall P \in \mathbb{E}(\mathbb{K}), P + \mathcal{O} = P$$
- Each element of the group has an inverse which is unique.
$$\forall P \in \mathbb{E}(\mathbb{K}), \exists! Q = (-P) \in \mathbb{E}(\mathbb{K}) : P + Q = \mathcal{O}$$
- The sum is associative.
$$\forall P, Q, R \in \mathbb{E}(\mathbb{K}), P + (Q + R) = (P + Q) + R$$
- The sum is commutative.
$$\forall P, Q \in \mathbb{E}(\mathbb{K}), P + Q = Q + P$$

Moreover, we have the following theorems.

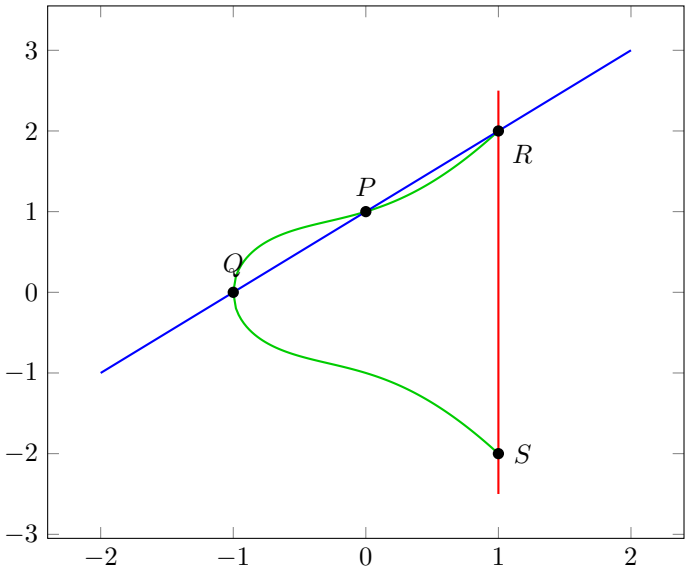


Figure 7.2: The chord and tangent rule.

Theorem 7.2 (M. Tsfasman; F. Voloch; H.-G. Ruck)**.** *Working over a finite field, the group of points $\mathbb{E}(\mathbb{F}_q)$ is always either a cyclic group or the direct product ($\times$) of two cyclic groups. More precisely,*

$$(\mathbb{E}(\mathbb{F}_q), +) \simeq (\mathbb{Z}_{n_1}, \cdot) \times (\mathbb{Z}_{n_2}, \cdot)$$

where

$$n_2 = gcd(n_1, q-1)$$

with only few fully described exceptions.

As a corollary we have

Theorem 7.3. *if $n = |\mathbb{E}(\mathbb{F}_q)|$ is equal to the product of distinct primes, then $(\mathbb{E}(\mathbb{F}_q), +)$ is cyclic.*

Proof. If $n = p_1 p_2$ then $p_1 p_2 = n_1 n_2$, but as $n2|n1$, then $n_2 = 1$. Being $\mathbb{Z}_{n_1}$ cyclic (see the statement of the Theorem) then also $\mathbb{E}(\mathbb{F}_q)$ is cyclic. □

Computation of the chord and tangent law

We've seen the theory behind the chord and tangent law but we haven't mathematically explained how the sum of two points should be computed. Let us consider the field $\mathbb{F}_{p^m}$ for $p > 3$. Let

$$P_1 = (x_1, y_1) \in \mathbb{F}_{p^m} \times \mathbb{F}_{p^m}$$

and

$$P_2 = (x_2, y_2) \in \mathbb{F}_{p^m} \times \mathbb{F}_{p^m}$$

Table 7.3 shows the formulas to directly compute the sum between P_1 and P_2.

	x_3	y_3
$P_1 + P_2 = (x_3, y_3)$	$\left(\frac{y_1-y_2}{x_1-x_2}\right)^2 - x_1 - x_2$	$\frac{y_1-y_2}{x_1-x_2}(x_1 - x_3) - y_1$
$[2]P_1 = (x_3, y_3)$	$\left(\frac{3x_1^2+a}{2y_1}\right)^2 - 2x_1$	$-y_1 + \left(\frac{3x_1^2+a}{2y_1}\right)(x_1 - x_4)$
$-P_1 = (x_3, y_3)$	x_1	$-y_1$

Table 7.3: The formulas to compute the chord and tangent law.

Let us now consider the field $\mathbb{F}_{2^m}$. Let

$$P_1 = (x_1, y_1) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$$

and

$$P_2 = (x_2, y_2) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$$

be two points on the curve $\mathbb{E}(\mathbb{F}_{2^m})$. Table 7.4 shows the formulas to directly compute the sum between P_1 and P_2.

	x_3	y_3
$P_1 + P_2 = (x_3, y_3)$	$\left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a$	$\left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + x_3 + y_1$
$[2]P_1 = (x_3, y_3)$	$x_1^2 + \frac{b}{x_1^2}$	$x_1^2 + \left(x_1 + \frac{y_1}{x_1}\right)x_3 + x_3$
$-P_1 = (x_3, y_3)$	x_1	$x_1 + y_1$

Table 7.4: The formulas to compute the chord and tangent law.

Let us now understand how to obtain the results in Table 7.3. Let's start by considering two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the curve $\mathbb{E}(\mathbb{F}_{p^m})$. From Definition 7.4, the result of the sum between two points is the intersection between the line passing between the points P_1 and P_2 and the elliptic curve. The line passing through P_1 and P_2 can be written as

$$y = \lambda(x - x_1) + y_1$$

with $\lambda = \frac{y_1-y_2}{x_1-x_2}$, which is the slope of the line passing through P_1 and P_2. Thanks to this equation we can write the system

$$\begin{cases} y^2 = x^3 + ax + b \\ y = \lambda(x - x_1) + y_1 \end{cases}$$

Replacing the second equation in the first one we obtain:

$$\begin{aligned}
\big(\lambda(x-x_1)+y_1\big)^2 &= x^3+ax+b\\
\big(\lambda(x-x_1)+y_1\big)^2 &= x^3+ax+b\\
\big(\lambda(x-x_1)\big)^2+y_1^2+2\lambda(x-x_1)y_1 &= x^3+ax+b\\
\lambda^2(x-x_1)^2+y_1^2+2\lambda(x-x_1)y_1 &= x^3+ax+b\\
\lambda^2(x^2+x_1^2-2xx_1)+y_1^2+2\lambda(x-x_1)y_1 &= x^3+ax+b\\
\lambda^2x^2+\lambda^2x_1^2-\lambda^2 2xx_1+y_1^2+2\lambda(x-x_1)y_1 &= x^3+ax+b\\
x^3-\lambda^2x^2-\lambda^2x_1^2+\lambda^2 2xx_1-y_1^2-2\lambda(x-x_1)y_1+ax+b &= 0\\
x^3-\lambda^2x^2+(\lambda^2 2x_1-2\lambda y_1+a)x-\lambda^2x_1^2-y_1^2+2\lambda x_1y_1+b &= 0
\end{aligned}$$

The equation on the left of the equality is a third-degree polynomials, hence it has three roots, two of which are known and are x_1 and x_2 (because the line passes through P_1 and P_2). The equality can therefore be written as

$$(x-x_1)(x-x_2)(x-x_3)=0$$

If we expand the term on the left we get

$$\begin{aligned}
&(x-x_1)(x-x_2)(x-x_3)=0\\
&(x^2-xx_1-xx_2+x_1x_2)(x-x_3)=0\\
&x^3-x^2x_1-x^2x_2+xx_1x_2-x^2x_3+xx_1x_3+xx_2x_3-x_1x_2x_3=0\\
&x^3-(x_1+x_2+x_3)x^2+(x_1x_2+x_1x_3+x_2x_3)-x_1x_2x_3=0
\end{aligned}$$

By looking at the coefficients of the unknown with degree 2, we can write

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

We can now isolate x_3 and obtain the formula for computing it as

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

And since $\lambda=\frac{y_1-y_2}{x_1-x_2}$, we get

$$x_3=\left(\frac{y_1-y_2}{x_1-x_2}\right)^2-x_1-x_2$$

Replacing this value (i.e., $x=x_3$) in the equation $y=\lambda(x-x_1)+y_1$ gives us the formula for computing y_3, namely

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

The same procedure can be used for computing the formula for $[2]P_1$. In this case, we have to remember that the slope of the tangent to a function f passing through a point P is the derivative

of f evaluated in P. The derivative of the elliptic curve can be computed as

$$\begin{aligned}\frac{d}{dx}y(x) &= \frac{d}{dx}(x^3 + ax + b)\\ 2y(x)\frac{d}{dx}y(x) &= 3x^2 + a\\ \frac{d}{dx}y(x) &= \frac{3x^2 + a}{2y(x)}\\ \frac{d}{dx}y &= \frac{3x^2 + a}{2y}\end{aligned}$$

When evaluated in x_1, the slope of the tangent in P_1 becomes

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

Finally, the coordinate of the opposite of a given point are obtained through intersecting the curve with a vertical line $x = x_1$.

In the case of binary finite fields, the method to derive the addition and doubling formulas is exactly the same method seen for curves over prime fields.

7.5.3 Points of an elliptic curve over a finite field

To use elliptic curves for cryptography we need elliptic curves defined over finite fields $\mathbb{F}_{p^m}$. This means that we need a way to compute the point belonging to the curve. If the group is small enough, we can use a bruteforce approach. This means that we can enumerate every possible point and check if it belongs to the curve by replacing it in the Weierstrass equation. This approach works only when we have a small group. Let us then focus on a more general approach. For starters, we can try and understand if it's possible to know how many points belong to the curve. In general, the number of points belonging to an elliptic curve $n = |\mathbb{E}(\mathbb{F}_{q=p^m})|$ can be upper bounded by $q + 1$, namely we have

$$n \leq q + 1$$

We can also write

$$n = q + 1 - t$$

where t is an unknown parameter that we can try and approximate. The following theorem approximates the number of points belonging to the curve, namely to approximate t.

Theorem 7.4 (Hasse)**.** *The number of points of an elliptic curve $n = |\mathbb{E}(\mathbb{F}_q)| = q + 1 + t$ lies between*

$$q + 1 - 2\sqrt{q} \leq n \leq q + 1 + 2\sqrt{q}$$

Equivalently,

$$|t| \leq 2\sqrt{q}$$

The best known algorithm for computing $n = |\mathbb{E}(\mathbb{F}_{p^m})|$ is the **Schoof–Elkies–Atkin algorithm** and has a time complexity of

$$\mathcal{O}(\log_2^5 q)$$

This means that computing the order of a finite elliptic curve group is computationally easy. In practice, only curves with *nearly prime* cardinality are considered, namely

$$|\mathbb{E}(\mathbb{F}_q)| = c \cdot p$$

where $p \geq 2160$. To properly work in the subgroup with prime order p, we need to compute one of its generators it is sufficient to consider a generic point $Q(x_Q, y_Q)$ over the curve and test whether $[c]Q \neq \mathcal{O}$ or not. If the test is passed, then $P = [c]Q$ is a generator of the subgroup with prime order.

7.5.4 Elliptic curve discrete logarithm problem

To evaluate the security margin of elliptic curve cryptosystems we have to understand what problem an attacker should solve. As we did for cryptosystems based on $(\mathbb{F}_{p^m}, \cdot)$, we can define the elliptic curve discrete logarithm problem.

Definition 7.5 (Elliptic curve discrete logarithm problem)**.** *Let $(\mathbb{E}(\mathbb{F}_q), +)$ be a prime group as a sub-group of points on an elliptic curve and $P, Q \in \mathbb{E}(\mathbb{F}_q)$ be two points on the curve. The elliptic curve discrete logarithm problem requires finding the smallest value $k \in \{0, \ldots, |\mathbb{E}(\mathbb{F}_q)|\}$ such that*

$$[k]P = Q$$

The multiplication $[k]P$ can be computed with an algorithm similar to the square-and-multiply algorithm which is called **double and add**.

The complexity of the elliptic curve discrete logarithm problem defines the security margin of elliptic curve cryptosystems. The best-known algorithm for solving the elliptic curve discrete logarithm problem has a complexity of

$$\mathcal{O}(\log^{\frac{1}{2}} |\mathbb{E}(\mathbb{F}_{p^m})|))$$

As a result, the elliptic curve problem is much harder than the discrete logarithm problem, hence we can use shorter keys. Elliptic curve groups G should have a prime order n (or an order whose biggest factor is big enough) because, to solve the elliptic curve DLP, we can use an algorithm similar to the Pohlig-Hellman algorithm seen for solving the classical discrete logarithm problem. The Pohlig-Hellman algorithm reduces the problem of solving the DLP in a group with cardinality $n = p_1 p_2 \cdots p_k$ to solving the DLP in a group whose cardinality is the biggest order among the factors p_i. As a result, if the order of G is prime, then the complexity of solving the DLP is the complexity of solving the DLP in a group of order n. If $n = p_1 \cdots p_k$ and every p_i is small, then solving the DLP boils down to solving the DLP in a group with small cardinality. Table 7.5 compares the key length required by elliptic curve and classical cryptosystems to obtain the same level of security.

The elliptic curves used for cryptography are standardised by the NIST putting care in obtaining particularly fast finite field arithmetic.

7.5.5 Projective plane

The formulas for computing the sum between two points on an elliptic curve requires us to compute some divisions. Dividing two numbers is expensive (for sure more expensive than summing or subtracting them), hence it would be good to find a way to compute the sum without the division.

Symmetric key size [bit]	Size of ECC prime order group [bit]	RSA/DH/DSA (key lengths-group size) [bit]
80	160	1024
112	224	2048
128	256	3072
192	384	7689
256	512	15360

Table 7.5: Key lengths required by elliptic curve based and normal cryptosystems.

Solving this problem requires us to introduce a new set of coordinates on which computing the sum between points doesn't require to compute a division. This new set of coordinates is called **projective coordinates**, which allows to define a **projective plane** on which we can draw **projective points**. For starters, let us define a projective plane.

Definition 7.6 (Projective plane)**.** *Let $\mathbb{K}$ be a field. The projective plane $\mathbb{P}^2(\mathbb{K})$ over $\mathbb{K}$ is defined as the set of triples (X, Y, Z) where $X, Y, Z \in \mathbb{K}$ are not all simultaneously zero. The triple $(X, Y, Z) \in \mathbb{K}^3$ is called projective point. On these triples is defined an equivalence relation*

$$(X, Y, Z) \equiv (\lambda X, \lambda Y, \lambda Z)$$

with $\lambda \in \mathbb{K} \setminus \{0\}$.

In practice, a point on a projective plane is a class of equivalence containing all the points $(\lambda X, \lambda Y, \lambda Z)$. For instance, in $\mathbb{P}^2(\mathbb{F}_{11})$ the points

$$(4, 3, 1)$$

and

$$(1, 9, 3)$$

are equivalent (i.e., the same point in a projective plane) because we can write

$$(1, 9, 3) \equiv (3 \cdot 4, 3 \cdot 3, 3 \cdot 1) \mod 11$$

Correspondence between projective plane and Cartesian plane

Points on a projective plane can be mapped to points on the Cartesian plane. If $Z \neq 0$ we can find a bijection between a projective point (X, Y, Z) and a point in the Cartesian plane (x, y). In particular, there exists a unique tuple in its equivalence class $(X, Y, 1)$ where

$$x = \frac{X}{Z}$$

and

$$y = \frac{Y}{Z}$$

that can be put in one-to-one correspondence with the affine point (x, y).

If instead we consider $Z = 0$, the point is called point to infinity and it's the neutral element $\mathcal{O}$ of the group $(\mathbb{E}(\mathbb{K}), +)$. Since a point to infinity has $Z = 0$, then we can write it as $(X, Y, 0)$. This point is equivalent to a point in which every coordinate is scaled by $\frac{1}{X}$, hence we obtain

$$(X, Y, 0) \equiv \left(1, \frac{Y}{X}, 0\right)$$

If we take a Cartesian plane, $\frac{y}{x}$ is the slope of a line passing through the origin (since $y = \lambda x$ is the equation of the line). This means that points to infinity can be put in direct correspondence with the direction of the straight lines having slope $\frac{Y}{X}$.

Elliptic curves in the projective plane

Having introduced a new plane, we have to define elliptic curves for this new plane. We want to rewrite the Weierstrass equation using projective points.

Definition 7.7 (Elliptic curve in the projective plane)**.** *Let $\mathbb{K}$ be a field. An elliptic curve is defined as the set of solutions in the projective plane of a non-singular homogeneous Weierstrass equation of the form*

$$\mathbb{E}(\mathbb{K}) : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

The points at infinite of an elliptic curve can be found by fixing $Z = 0$, hence by solving the following system:

$$\begin{cases} Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \\ Z = 0 \end{cases}$$

If we replace the second equation into the first we obtain

$$0 = X^3$$

which yields $X = 0$. This means that the points at infinite are of the form

$$\mathcal{O} = (0, \lambda, 0) \ , \lambda \in \mathbb{K} \setminus \{0\}$$

If we consider a finite field $\mathbb{F}_{p^m}$ $(p > 3)$, the Weierstrass equation can be rewritten as

$$\mathbb{E}(\mathbb{F}_{p^m}) : Y^2Z = X^3 + aXZ^2 + bXZ^2 + bZ^3$$

with

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

to ensure that the curve is non-singular.

Addition formula in the projective plane

The formula for adding two points in the projective plane on an elliptic curve can be obtained similarly to the Cartesian case. Let us consider two points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$.

The coordinates of the sum $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$ can be computed as

$$\begin{cases} X_3 = p_1(t - q_2) \\ 2Y_3 = r(3q_2 - 2t) - p_3(s_2 + s_1) \\ Z_3 = wp_3 \end{cases}$$

where

- $u_1 = X_1 Z_2$
- $u_2 = X_2 Z_1$
- $s_1 = Y_1 Z_2$
- $s_2 = Y_2 Z_1$
- $w = Z_1 Z_2$
- $p_1 = u_1 - u_2$
- $p_2 = p_1^2$
- $p_3 = p_1 p_2$
- $q_1 = u_1 + u_2$
- $q_2 = p_2 q_1$
- $r = s_2 - s_1$
- $t = wr^2$

Doubling formula in the projective plane

The formula for doubling a point in the projective plane on an elliptic curve can be obtained similarly to the Cartesian case. Let us consider a point $P_1 = (X_1, Y_1, Z_1)$. The coordinates of $P_3 = [2]P_1 = (X_3, Y_3, Z_3)$ can be computed as

$$\begin{cases} X_3 = \lambda_1(\lambda_4 - 4\lambda_5) \\ Y_3 = \lambda_2(6\lambda_5 - \lambda_4) - 2Y_1^2\lambda_6 \\ Z_3 = \lambda_1\lambda_6 \end{cases}$$

where

- $\lambda_1 = 2Z_1 Y_1$
- $\lambda_2 = 3X_1^2 + aZ_1^2$
- $\lambda_3 = X_1 Y_1$
- $\lambda_4 = \lambda_2^2$
- $\lambda_5 = \lambda_1\lambda_3$
- $\lambda_6 = \lambda_1^2$

Types of projective planes

There exist different projective planes, which are variants of the projective coordinates we have described. Different practical implementations of elliptic curves choose different projective coordinates depending on many factors like

- the specific finite field,
- the average number and the relative efficiency of each arithmetic operation (multiplications, additions, doublings),
- how many point additions or how many point doubling a specific primitive is supposed to execute.

Each coordinate system offers different trade-offs between the number of coordinates and the computational cost. Some examples are:

- The **Jacobian system**. In the Jacobian system (also adopted for the NIST standard curves) a point (x, y) is also represented with three coordinates (X, Y, Z), but $x = \frac{X}{Z^2}$, $y = \frac{Y}{Z^3}$.
- The **López–Dahab system**. In the López-Dahab system the relation between Cartesian and projective coordinates is $x = \frac{X}{Z}$ and $y = \frac{Y}{Z^2}$.
- The **modified Jacobian system**. In the modified Jacobian system the same relations are used but four coordinates are stored and used for calculations, i.e., (X, Y, Z, aZ^4).
- The **Chudnovsky-Jacobian system**. In the Chudnovsky-Jacobian five coordinates are used, i.e., (X, Y, Z, Z^2, Z^3).

7.5.6 Elliptic curve Diffie-Hellman

Now that we know how elliptic curve cryptography works, we can apply it to some asymmetric protocols we've already studied. Let's start from the Diffie-Hellman protocol. The Elliptic Curve Diffie-Hellman (ECDH) protocol is an asymmetric key agreement protocol that allows two parties, say Alice and Bob, to agree on a shared secret key by communicating on an insecure channel. The protocol works as follows:

1. Alice and Bob publicly agree on a group $(\mathbb{E}(\mathbb{F}_{p^m}), +)$ with order $n = |\mathbb{E}(\mathbb{F}_{p^m})|$ and generator P.
2. Alice chooses a private key $k_{priv,A} \in \{2, \ldots, n-1\}$.
3. Bob chooses a private key $k_{priv,B} \in \{2, \ldots, n-1\}$.
4. Alice computes $k_{pub,A} = [k_{priv,A}]P$.
5. Bob computes $k_{pub,B} = [k_{priv,B}]P$.
6. Alice sends $k_{pub,A}$ to Bob.
7. Bob sends $k_{pub,B}$ to Alice.
8. Alice computes $k_{shared} = [k_{priv,A}]([k_{priv,B}]P)$.
9. Bob computes $k_{shared} = [k_{priv,B}]([k_{priv,A}]P)$.

7.5.7 Elliptic curve ElGamal encryption

Elliptic curve ElGamal encryption works like normal ElGamal encryption but multiplications are replaced with sums and powers are replaced with multiplications. Let us consider a group $(\mathbb{E}(\mathbb{F}_{p^m}), +)$ with order $n = |\mathbb{E}(\mathbb{F}_{p^m})|$ and generator $P = (x_P, y_P)$. The protocol works as follows:

1. Assume Alice is willing to send an encrypted message to Bob, then she maps the message into a curve point $M \in \mathbb{E}$.
2. Bob is equipped with:
   - a secret key $0 < k_{priv,B} < n$, and
   - a public key $(n, P, k_{pub,B} = [k_{priv,B}]P)$.
3. to encrypt M, Alice
   (a) picks at random $r = \{1, \ldots, n-1\}$,
   (b) computes $\gamma = [r]P$ and $\delta = M + [r]k_{pub,B}$, and
   (c) sends the ciphertext (γ, δ).
4. After receiving the ciphertext (γ, δ), Bob
   (a) computes $[k_{priv,B}]\gamma = [k_{priv,B}]([r]P) = [r]k_{pub,B}$,
   (b) decrypts the message as $\delta - [k_{priv,B}] = (M + [r]k_{pub,B}) - [r]k_{pub,B}$.

The main issue of this protocol is that a message M should be a point on the elliptic curve, however, it's not always possible to map a message into a point on the curve. To solve this problem we have to modify some operations of the protocol. The updated protocol works as follows:

1. Alice wants to send an encrypted message m encoded in binary with the same bit-size of x_P to Bob.
2. Bob is equipped with:
   - a secret key $0 < k_{priv,B} < n$, and
   - a public key $(n, P, k_{pub,B} = [k_{priv,B}]P)$.
3. to encrypt M, Alice
   (a) picks at random $r = \{1, \ldots, n-1\}$,
   (b) computes $\gamma = [r]P$ and $\delta = m \oplus + xcoord([r]k_{pub,B})$ (where the function $xcoord(\cdot)$ extracts the x coordinate from its argument), and
   (c) sends the ciphertext (γ, δ).
4. After receiving the ciphertext (γ, δ), Bob
   (a) computes $[k_priv, B]\gamma = [k_{priv,B}]([r]P) = [r]k_{pub,B}$,
   (b) decrypts the message as $\delta \oplus xcoord([k_{priv,B}])$.

7.5.8 Elliptic curve signatures

Elliptic curves can be applied to signatures, too. When Alice wants to sign a message $m = \{0,1\}^*$, she

1. picks $r = \{1, ..., n-1\}$ such that $gcd(r, n) = 1$,
2. computes $[r]P = (x_1, y_1)$ and $k = x_1 \mod n$
3. if $k = 0$ then goes to step 1, else
4. computes $z = r^{-1}(SHA1(m) + sk) \mod n$.
5. if $z = 0$ then goes to step 1.

The signature of m is (k, z). Bob, to verify a signature (k, z),

1. checks if $k, z \in \{1, ..., n-1\}$
2. computes $u_1 = z^{-1} \cdot SHA1(m) \mod n$ and $u_2 = z^{-1} \cdot k \mod n$,
3. computes $X = u_1 P + u_2 k_{pub,A} = (x_1, y_1)$,
4. if $X = \mathcal{O}$ then rejects the message is rejected, otherwise
5. accepts the signature if and only if $x_1 \mod n = k$.

When the signature (k, z) is correct, we have that $z = r^{-1}(SHA1(m) + sk) \mod n$, thus:

$$\begin{aligned} u_1 P + u_2 k_{pub,A} &= SHA1(m) z^{-1} P + k z^{-1}(sP) \\ &= (z^{-1}(SHA1(m) + sk))P \\ &= rP = (x_1, y_1) \Rightarrow k = x_1 \mod n \end{aligned}$$

Chapter 8

Public key authentication

8.1 Hybrid cryptoschemes

One important problem we want to solve is communication security, that is obtaining confidential (i.e., only the sender can read the message) and authenticated (the sender knows that the receiver is who he or she claims to be) communication between two parties (say Alice and Bob).

8.1.1 Purely symmetric encryption

If we could only use symmetric encryption, this problem would have had a trivial solution which is to meet in person the other party we want to communicate with, exchange a key k and use that key k for communicating. This mechanism provides confidentiality because the key is shared only by the two people that are communicating and authentication because of the fact that we can read messages written with the key we've exchanged. This solution isn't however applicable to real-world usage since we need some mechanism that doesn't require two parties to meet in person.

Even if we could assume that there existed a way to exchange a shared key k, purely shared encryption would still be a bad idea since it doesn't scale well. In fact, in a group we want to have a shared key for each pair of users, hence we have to generate $n(n-1)$ keys since each user has to generate n keys (and keys are symmetric, that's why it's not n^2).

To sum things up, purely symmetric encryption has two main problems:

- **It requires the parties to meet in person** to share a secret key.
- **It doesn't scale well**.

8.1.2 Hybrid cryptosystems

We can use digital signatures to solve the problems related to purely symmetric encryption. If Alice and Bob want to communicate, each generates an asymmetric key pair, namely $(k_{pub,A}, k_{pri,A})$ and $(k_{pub,B}, k_{pri,B})$. These pairs are then used to encrypt and sign a shared key which will then be used to encrypt all future messages. In practice, when Alice starts communicating with Bob:

1. Alice chooses a shared key k. This key is usually called session key since it's used for a conversation (i.e., a limited exchange of messages that ends when one of the parties disconnects from the other).

2. Alice signs the shared key with her private key $k_{pri,A}$, obtaining $\mathbb{S}_{k_{pri,A}}(k)$. This means that anyone can verify that the message k has been sent by Alice (i.e., k has been signed by her).
3. Alice encrypts $\mathbb{S}_{k_{pri,A}}(k)$ with Bob's public key, obtaining $\mathbb{E}_{k_{pub,B}}(\mathbb{S}_{k_{pri,A}}(k))$. This means that only Bob can read $\mathbb{S}_{k_{pri,A}}(k)$.
4. Alice sends $\mathbb{E}_{k_{pub,B}}(\mathbb{S}_{k_{pri,A}}(k))$ to Bob.
5. Bob decrypts the message to obtain $\mathbb{S}_{k_{pri,A}}(k)$.
6. Bob checks the signature $\mathbb{S}_{k_{pri,A}}(k)$ and obtains the key k which is used to communicate with Alice from now on.

This solution allows to exploit the speed of symmetric encryption (otherwise using asymmetric encryption to encrypt the whole message would be too expensive and slow) without the need of meeting in person thanks to digital signatures and public key encryption.

General hybrid cryptosystem

In general, in a hybrid cryptosystem, when Alice wants to talk to Bob:

1. Alice picks a random shared key k_{msg} which is used for a single message.
2. Alice encrypts a message m with the shared key k_{msg}.
3. Alice encrypts the shared key with Bob's public key $p_{pub,B}$ to obtain $\mathbb{E}_{k_{pub,B}}(k_{msg})$.
4. Alice computes the hash of the message $h(m)$. She could even use the message itself if it's really short.
5. Alice signs the hashed message with her private key $k_{pri,A}$ to obtain $\mathbb{S}_{k_{pri,A}}(h(m))$. Note that we sign the hash since public key encryption (and signing) is long, hence it's better to sign a hash, which is usually smaller than the original message.
6. Alice sends the encrypted message, the encrypted key and the signature to Bob.
7. Bob decrypts the shared key using his private key to obtain $k_{msg} = \mathbb{D}_{k_{pri,B}}(\mathbb{E}_{k_{pub,B}}(k_{msg}))$.
8. Bob uses the shared key k_{msg} to decrypt the message m.
9. Bob checks the signature $\mathbb{S}_{k_{pri,A}}(h(m))$ using Alice's public key.

This version is more general, in fact, we can use k_{msg} for more messages and obtain the first version of the cryptosystem.

Chapter 9

Digital certificates

9.1 Digital certificates

One problem remains still open in hybrid cryptosystems. In fact, when we receive a signature, we can verify it, but we can't tell if the public key associated with that signature (i.e., the public key used to verify that signature) actually belongs to the person we should receive the message from. The best solution we have to solve this problem is digital certificates. A digital certificate is a signed document containing

1. a public key, and
2. the identity (i.e., name, surname and mail for a person or the domain name for a server) of the person that owns that public key.

The document (or better, its hash) is signed by an authority that grants that a public key belongs to a person. The authorities that release and sign certificates are usually organised in a hierarchy where each node signs and grants for the nodes below. The root of the hierarchy is made of a set of nodes that grant for themselves and that we should trust. The infrastructure used to handle the association between a public key and a person is called Public Key Infrastructure (PKI).

When dealing with certificates we have to take care of three important aspects:

- **Certificate issuing**. We have to understand how certificates are built, signed and distributed.
- **Certificate use**. We have to understand how certificates are used to ensure authentication.
- **Certificate revocation and expiration**. We have to understand when a certificate should expire or be revoked. Digital certificate revocation is not easy since a certificate has multiple copies stored all over the Internet or maybe on computers not connected to the Internet.

A certificate is signed by a trusted entity which can be centralised or distributed. In particular, we employ two solutions:

- The **Public Key Infrastructure**. The PKI is a centralised entity organised in a hierarchy.
- The **Web of Trust**. The Web of Trust is a distributed trusted entity.

Each solution has its advantages and disadvantages but the final choice depends on the scale of the system. Let us now analyse each solution more in-depth.

9.2 Public Key Infrastructure

9.2.1 Certificate creation

The document containing the public key-user association is written using a grammar written in the X.509 standard. This grammar is based on the Abstract Syntax Notation One grammar.

Format

When building a certificate we have to define a format with which the certificate is written. Namely, we have to define, for each letter of the alphabet to which string of byte it's translated. In other words, we have to encode the alphabet.

One of the most used encoding mechanisms is Distinguished Encoding Rules (DER), defined in ITU X.690. This encoding scheme is very space efficient but it's used only by the machine since it contains non-printable characters, too. Microsoft uses PEM instead, which is basically Base64 encoding, hence it's printable (hence it can be copied and pasted easily) and more readable.

Certificates can also be bundled to generate a PKCS#12 digital envelope which contains multiple public keys and, optionally, some private keys.

Parsing certificates is hard since the grammar is not context-free and it's ambiguous. This is intrinsic in the fact that a sequence of bytes can be either seen as one long number or two smaller ones. The ambiguity of the grammar has been exploited to craft malicious certificates.

Contents

The X.509 standard mandates the certificate to contain:

- **Version and serial number**. Identify the certificate and specify allowed extensions (some of which are critical, hence to be decoded, other are not). The version is stuck to 3 since a long ago. The serial number is a 64-bit string and it's unique per issuer.
- **Subject**. The person, or entity to which the key belongs.
- **Issuer**. The entity that verified the information and issued (i.e., signed) the certificate.
- **Valid-From/Valid-To**. The validity period of the certificate.
- **Key-Usage**. Purpose of the public key (e.g. encryption, signature verification on data, signature verification on certificates). This information is very important since using the same key for multiple purposes may lead to some attacks, hence it's necessary to specify what a key is used for. For instance, if we use the same key for encrypting and signing in RSA, someone could ask to verify a message and get the decryption of something else as a result (this is because decryption and signing are the same in RSA). Note that specifying signature verification on data or on certificates is very important since data can contain random bytes whereas certificates have a specific structure.
- **Public key**. The public key to be bound to the subject.
- **Signature Algorithm**. The algorithm used to sign the hash of the certificate and the hash algorithm.
- **Signature**. The actual signature of the hash of the certificate.

Signing a certificate

Signing a certificate isn't as easy as simply hashing the document and signing it. Signatures are in fact created using the RFC 5280 standard which says that we have to sign, in this sequence:

1. Version and serial number.
2. Subject, issuer and validity dates.
3. Public key to be certified.
4. The critical extensions.

These fields are

1. encoded in DER,
2. hashed using the OS2IP (i.e., a custom sequence of additions) algorithm or a hashing function from the PKCS1 standard (the latter solution should be preferred),
3. signed with the issuer's private key $k_{pri,\ issuer}$.

A certificate is reliable only if the hash is collision resistant and the hashing algorithm is not broken. In the case of certificates, collisions are even more problematic. Consider for instance MD5. The algorithm for finding MD5 collisions takes two messages m_1 and m_2 and generates two pads p_1 and p_2 that generate a collision when appended to m_1 and m_2 respectively. Namely, we get $h(m_1||p_1) = h(m_2||p_2)$. This means that we can

1. take a valid certificate m_1,
2. forge another certificate m_2 with whatever we want on it,
3. we give m_1 and m_2 to the MD5 collision generator that will return two pads p_1 and p_2,
4. append the pad p_1 to the first certificate as a non-critical comment, hence it won't be considered,
5. append the pad p_2 to the second certificate, which will appear as a critical comment.

The complete attack is way more complex than this and it's available here (https://marc-stevens.nl/research/papers/CR09-SSALMOdW.pdf).

9.2.2 Entities

The Public Key Infrastructure is based on three types of entities:

- **Certification Authority**. A certification authority is a technical firm that deals with the technical part of signing a certificate. This means also securing the private keys and distributing the certificates.
- **Registration Authority**. A registration authority deals with the registration of a user. A registration authority is required since the certificate authority might not meet the legal requirements or might not want to deal with the burden of dealing with the user's identity. For instance, the firm that handles the users has to comply with the GDPR. The registration authority verifies that a person is who he or she claims to be and passes an opaque identifier to the certification authority, which uses the identifier for the signature.
- **User**. A user can either be a person or machine that asks to sign his or her certificate or employs the CA to verify the authenticity of the certificates for another user.

9.2.3 Actions

Bootstrapping a certificate authority

To sign other certificates, a root CA has to create a self-signed certificate. Practically, the CA:

1. Generates a public-private key pair for itself.
2. Generates a certificate for itself, which is signed with its own key. This certificate is used to generate the other certificates.
3. Publishes the certificate. The certificate should be visible to as many people as possible since it's required to eventually verify a signature.

Issuing a certificate

The process of issuing a certificate to a certificate authority works as follows:

1. The user generates a key pair $(k_{pri,\ user}, k_{pub,\ user})$.
2. The user generates a Certificate Signing Request (CSR), which contains the user's identity and public key. The CSR is then signed with the user's private key and sent to the CA. Note that signing the CSR with the private key has no security advantage since someone could use the user's public key (which is in fact known by everyone) to obtain the original CSR, modify the message with its own public key (i.e., replace the user's public key in the CSR with the attacker's) and send it to the CA. This mechanism is used only to prevent denial of service attacks since an attacker should have to check, modify and sign the CRS before re-issuing it, hence losing time (remember that verification is faster than signature).
3. The CA checks the integrity of the CSR by checking the signature.
4. The CA checks if the identity specified in the CSR matches the identity of the user issuing the CSR (either via a registration authority or by verifying directly the identity).
5. The CA generates the certificate signs it with its private key and sends it to the user.

Since we've mentioned CSR, let us describe its structure. The CSR contains, in this order:

1. The **version**.
2. The **subject identity**.
3. The **public key algorithm**.
4. The **user's public key**.
5. The **attributes**, namely critical extensions.
6. The **signature algorithm**.
7. The **signature**.

The first 5 fields are encoded in DER, hashed using a secure hash algorithm and signed using the algorithm specified in field 6 (which is not included in the signature). The result is then stored in field 7.

OpenVPN has a handful set of scripts on GitHub for creating and setting-up a certificate authority.

Checking the authenticity of an object

When a client retrieves an object from a server:

1. The client obtains the encrypted object and the certificate signed with the CA's private key.
2. The client checks that the CA is among the ones he or she trusts (otherwise a recursive call to another CA is required).
3. The client checks the signature done by the CA on the certificate.
4. The client checks the signature done by the server on the retrieved object.

Compromised certificates

A certificate might be compromised (i.e., the public key on the certificate might be wrong) for many reasons:

- The certificate is **expired**. If the certificate is expired, it contains a key no longer used. This is not a problem since we can verify the expiration date on the certificate (which is signed, hence its integrity is ensured) and simply discard the certificate.
- The certificate is **not expired**. If the certificate is not expired we need to revoke it. This means that we need an infrastructure to tell the clients that a certificate received from a server is not valid because it's compromised. Certificate revocation is handled using:
  - **Certificate Revocation Lists** (CRLs).
  - **Online Certificate Status Protocol** (OCSP).

Certificate Revocation Lists Certificate Revocation Lists are blacklists, namely lists of certificates (certificate ids) that are not valid anymore since they've been compromised. A user can download these lists and check if the list contains a certificate. Each node of the list contains:

- The id of the revoked certificate.
- The date of revoking.
- Some optional extensions.

A revocation list is managed by the same certificate authority that handles the certificates in the list since we can only trust the CA to know if a certificate has been compromised. The list of revoked certificates is specified in a critical extension as a URL pointing to where the CRL can be retrieved. Since the list is in a critical extension, it's signed by the CA that manages the list. This means that, even if the private key of the user has been compromised, we can still use the information in this list because the list is signed with the CA's private key, not the user's.

Online Certificate Status Protocol The Online Certificate Status Protocol allows to query a certificate authority to understand if a certificate with a certain id (which is unique with respect to a single certificate authority) is valid or not. This means that every time a user wants to check if a certificate is revoked, it has to query the CA's servers. The idea is similar to CRLs, however in this case the check is moved to the server which can highly optimise the search (e.g., using a database instead of a list). However, since the CA server is queried many times, we often use local caches on

the client. Local caches can increase the query speed but it can create security issues. In fact, if the local cache is updated rarely, it might contain old data, hence a compromised certificate could be considered valid.

An OCSP request contains:

1. The **version** of the protocol.
2. The **name of the requester**, which is optional.
3. A **list of requests**. Each request contains:
   (a) The **identifier of the hash algorithm** used for the certificate we want to verify.
   (b) The **hash of the issuer's name**.
   (c) The **hash of the issuer's public key**.
   (d) The **serial number of the certificate**.
4. Some **extensions**, which are optional.
5. The **signature algorithm** used to sign the OCSP request. This field is optional.
6. The **signature** of the OCSP request (i.e., of the first four fields). This field is optional in fact signing the request is used just for error checking.

Since version 3 of the OCSP protocol, nonces are included in the request to avoid reply attacks.

9.2.4 Hierarchical structure

The public key infrastructure is organised in a hierarchical structure. In this way, we can have a small set of root CAs which everyone trusts. The root CAs sign the certificates for their children (i.e., a root CA signs the association between a child and its public key). The children sign the certificates of their children and so on until we reach the end users. Each level of the tree warrants the authenticity of the level above, with the exception of the root, which provides warranties for itself.

Issuing and revoking certificates in a hierarchical PKI isn't that different from a flat PKI (i.e., with root CAs only). In particular:

- certificates are **issued to users only by leaf CAs**,
- certificates are **revoked by the CA that has emitted them (independently from the level)**.

Things get more complicated when users want to verify a certificate in fact we have to climb the hierarchy until we reach a root CA verifying the certificate of each node in between using its parent node. In practice:

1. The user checks if the certificate he or she wants to authenticate is signed by one of the CAs trusted by him or her (he or she has a copy of their certs).
2. If so (base case), the authenticity is checked with the public key of the trusted CA. If the certificate is authentic, he goes back to what he was checking before.

3. If not (recursion step), the user obtains the certificate of the CA which signed the certificate currently under exam and tries to authenticate it. Goes to 1. When the user knows the certificate is authentic, he can check the one currently under exam and go back to what it was checking before.

Since we have to check many intermediate certificates, verifying a certificate is a lengthy process. To speed things up we can do multiple checks in a single shot. This allows to reduce the length of the chain to 2 or 3 nodes.

Disadvantages

The main disadvantages of the hierarchical structure are:

- The root CAs are the single point of failure of the infrastructure since we trust them and if one of them is compromised the entire infrastructure gets compromised.
- The scheme still needs to distribute the certificates of the root CAs in a safe way and promptly remove root CA compromised keys.
- Since the chain of trust is indicated in the extensions field of the certificate, all the extensions should be properly parsed.
- The signature keys employed should match or exceed the advised key length by NIST.
- CAs should work at an equal- or higher- security level than the users. Namely, the signature keys used by the CAs should be larger or equal to the ones of the user since we would expect the CA to have a security level which is at least as good as the user's, otherwise an attacker can forge a certificate instead of owning the user's private key.

9.2.5 ACME protocol

The ACME protocol allows building a CA without ensuring extremely high-security guarantees. The company that invented this protocol gives out certificates for free using a fully mechanised protocol for identity testing, which they invented. When we want to obtain a certificate for a website we have to own the website. The protocol gives a randomly generated web page (i.e., a page with a random name and random content) to the user that has to put it on the website in a limited amount of time. If someone is able to put the randomly generated page on the website in a limited time, then we can safely assume that the user is the real owner of the website, hence the user can be given a certificate.

This protocol is vulnerable to man-in-the-middle attacks, however, the attacker should intercept the whole traffic. However, if an attacker is able to intercept the whole traffic, the owner is vulnerable even without having obtained the certificate.

The main advantages of the ACME protocol are:

- It's free.
- It has moderately acceptable security guarantees.
- It has a very tight validity (90 days) so revocation is minimised.

9.3 Web of Trust

The Web of Trust is the distributed alternative to the PKI. The Web of Trust was more popular in the past since the Internet is becoming a less cooperative game, but, since it's a distributed protocol, we need the cooperation of many users.

The main difference with respect to PKI is that an entity is both an end user (i.e., someone that requires a certificate) and a certificate authority. Moreover, the Web of Trust is based on the trust of someone we know. Namely, a user trusts someone he or she knows and might decide to trust someone known by a trusted entity. In some sense, trust is transitive (up until a level decided by the user), hence if we trust A and A trusts B then we might trust B. This schema works if we consider a small world (i.e., not the entire Internet). For instance, this approach is used for Linux distributions since the developers know each other hence they can sign each other certificates, which, in turn, increases trust.

9.3.1 Actors

The Web of Trust doesn't distinguish between issuers and subjects (i.e., CAs and users), hence we don't have a central authority that handles the certificates. This means that we have to define new actors who handle the certificates. The Web of Trust defines two different entities:

- **Users**. A user is both a CA and a end-user, hence he or she is able to encrypt, sign and verify a digital object. Since a user acts as a certificate authority he or she can sign an id-key association. The couple id-key is then kept in a local storage. When acting as a CA, a user arbitrarily chooses to sign an id-key pair depending on the knowledge of the user to grant for.
- **Keyservers**. Keyservers provide a globally synchronised, distributed storage which is used to distribute the certificates to the end-users. The storage is append-only, hence we can't remove an old id-key association. This means that revoking a certificate is not trivial. The distributed storage can be queried through the key or the user identifier. Everyone can run a keyserver.

9.3.2 Certificate format

Since the Web of Trust has a different approach to certificates, certificates have a different format than those used in PKI. That is, certificates used in the Web of Trust do not comply with the X.509 standard. An example of a certificate format is the OpenPGP format. In general, a certificate is composed of an id-key association and a list of signatures of other users that act as CAs. More precisely, each certificate is made of blocks called **pockets**. Each pocket contains:

- The **identifier** of the content of the pocket (1 byte).
- The **length** of the content (1 to 4 bytes).
- The **content** of the pocket.

Pockets are context-sensitive, hence they can be easily parsed (differently from PKI certificates). Pockets are encoded in binary for efficiency reasons, hence they are not printable. There also exists a printable format, called Radix64 (a variant of Base64 encompassing a mandatory, per line, CRC to allow the end user to check the integrity of the message), which is however not decodable, hence it's used only for representational uses only. Pockets are appended one after to other to build a certificate. In particular, initially, a certificate contains:

1. a pocket with the **subject's primary public key**, namely the key used by the requester (i.e., the subject) to sign digital objects (and certificates, too), and
2. a pocket with the **subject's identifier**.

These pockets are self-signed by the subject and

3. a pocket with the **signature**

is added to the certificate. Since using the same key for signing and for encrypting is not secure, a user should also publish his or her keys for encryption (or in general for other purposes). These keys can be appended to the certificate, in particular, for each key, we add:

4. a pocket with a **secondary public key**, and
5. a pocket with the binding between the primary key and the secondary key achieved through the **self-signature of the secondary key** (verifiable by the owner or subject) and the **signature of the secondary key verifiable with the subject's public key**.

Finally, each user acting as a CA can append to the certificate:

6. a pocket with a **signature of the certificate**.

9.3.3 Certificate signature

When a user acting as CA wants to sign a certificate:

1. The issuer (i.e., the user acting as CA) meets in person with the subject (i.e., the user with the certificate) to verify the subject's identity.
2. The issuer signs the subject's certificate by appending the signature to the certificate.
3. The certificate is published to the keyservers.

9.3.4 Certificate verification

Since we don't have a central authority and a certificate is signed by multiple CAs, a user has to use some sort of majority voting to verify a certificate. In particular, each user assigns to each other user one of the following trust levels and, verifies a certificate depending on the trust level:

- **Ultimate**. The ultimate trust level is assigned only to the subject itself. One signature by a CA with the ultimate trust level is enough to verify a certificate. Namely if one of the CAs that have signed the certificate has an ultimate trust level then a certificate is considered valid.
- **Complete**. The complete trust level is assigned to someone we deeply trust (i.e., like the ultimate level but for other users). One signature by a CA with a complete trust level is enough to verify a certificate. Namely if one of the CAs that have signed a certificate has a complete trust level then the certificate is considered valid.
- **Marginal**. Two signatures by CAs with marginal trust levels are enough to verify a certificate.
- **Untrusted**. Untrusted CAs aren't considered when verifying a certificate.

Note that the last three trust levels can be tuned by the user which can change the required number of CAs that have signed the certificate. For instance, we can say that we require three marginal CAs to verify a certificate. What we have described is the default behaviour.

As we can see, differently from PKI, verification is not recursive in fact the signatures are on the same file, hence they can be verified in one shot.

9.3.5 Certificate revocation

Since everyone can act as a CA, then everyone can revoke a certificate. When a user different from the subject of the certificate (i.e., the one associated with the public key on the certificate) wants to revoke the certificate he or she has to:

1. get the certificate,
2. add a pocket to revoke the certificate,
3. sign the pocket just added.

Note that this is the reason why we can only append pockets to a certificate and not remove it. In fact, if we could remove some pockets, we could also remove revocation pockets, hence making a revoked certificate valid again.

If instead, the subject itself wants to revoke his or her own certificate, then he or she has to:

1. add a pocket to revoke the certificate,
2. sign the pocket just added with its public key, so that he or she can verify it,
3. sign the pocket just added with its private key, so that the others can verify it.

As for signatures, a certificate can to be revoked by multiple CAs. The Web of Trust uses the same mechanism used for signatures to define how many revocations are required to actually revoke a certificate. In this case, a revocation by the issuer immediately revokes the certificate, independently from the trust level.

9.3.6 Disadvantages

The main disadvantages of the Web of Trust are:

- In-person verification is required.
- Prompt revocation is hard, especially for certificates that are not updated for a long time (i.e., that have a long expiration time).
- The identifier of the certificate is too short (only 32 bits), hence it's possible to have collisions among certificates.

9.4 The PGP/GPG encrypted mail protocol

Pretty Good Privacy (PGP) is the first attempt at providing strong cryptosystems to anyone. Gnu Privacy Guard (GPG) was developed as a free alternative to PGP. The idea behind the protocol is the same as PGP but GPG uses a standardised message and key format to provide interoperability. This protocol allows encrypting and signing any file but the most typical use is to encrypt and sign e-mails. GPG is CLI only, but GUIs are available for all the most common mail clients. Some examples are:

- GPGSuite for Apple Mail, and
- GPGOl for Outlook.

While relatively old, it is the only end-to-end e-mail encryption approach with wide trust.

9.4.1 Message preparation

PGP/GPG messages are encrypted and signed with a hybrid scheme. The signature and encryption actions are disjoint and the user may select whether to perform both of them or only one. The current implementation support the following asymmetric cyphers:

- RSA (which is the default asymmetric cypher),
- ElGamal/DSA.

The supported symmetric cyphers are:

- AES (which is the default symmetric cypher),
- Camellia,
- 3DES,
- CAST5,
- Twofish, and
- Blowfish.

Finally, the protocol supports the following hashing algorithms:

- SHA-1,
- SHA-2 (which is the default hashing algorithm),
- RIPEMD-160.

The procedure for preparing a message goes as follows:

1. The message is compressed (the `zip` and `bz2` algorithms are supported).
2. A random symmetric session key is generated for each recipient.
3. A copy of the message is encrypted for each symmetric key.
4. Each symmetric key is encrypted with the recipient public key.
5. The messages are bundled together and the hash of the bundle is signed.

PGP/GPG messages are expressed in the same packet-based format as the Web-of-Trust certificates (RFC4880). Once the signed message bundle is ready, its encoding is ready to be embedded in a common e-mail (all characters are 7-bit ASCII) message. The first way to embed the message bundle in an e-mail is to enclose it between two text markers, which look like

```
-----BEGIN PGP SIGNED MESSAGE-----
```

The PGP/GPG mail client will match it to understand where the message starts and ends. As this tends to clutter the message, especially in the signature-only case, it is possible to employ an `application/pgp-encrypted` MIME section followed by an `application/octet-stream` one for encrypted messages and an `application/pgp-signature` to delimit the signature.

Chapter 10

Algorithms

10.1 Primality test

Many protocols and cryptosystems require at least one prime number. This means that we have to find an efficient way to generate prime numbers. This can be done by generating a number and checking if it's prime. As a result, we have to find an efficient way to test if a number is prime. To understand if this approach is valid (i.e., picking a number at random and then checking if it's prime) we have to understand what is the probability of picking a prime number. The following theorem helps us in this sense.

Theorem 10.1 (Prime number theorem)**.** *Let $\pi(x)$ be the prime-counting function that gives the number of primes smaller than or equal to x, for any real number x. The limit of the quotient of the two functions $\pi(x)$ and $\frac{x}{\ln(x)}$ as x approaches infinity is 1:*

$$\pi(x) \sim \frac{x}{\ln(x)}, \quad x \to \infty$$

namely, the two functions are comparable. Equivalently, the value of the n-th prime number p_n is approximately equal to $n \cdot \ln(n)$, with the approximation error approaching 0 as n approaches infinity.

If we compute the probability of a number of being prime as the number of prime values $\pi(x)$ over all the possible values x, then we can write

$$Pr(x = p \text{ is prime}) = \frac{\pi(x)}{x}$$

From Theorem 10.1 we know that $\pi(x) \sim \frac{x}{\ln(x)}$, hence if we divide both sides by x we get $\frac{\pi(x)}{x} \sim \frac{1}{\ln(x)}$, hence we can say that the probability that a number $x = p$ is prime is about

$$Pr(x = p \text{ is prime}) = \frac{\pi(x)}{x} = \frac{1}{\ln(x)}$$

If we consider numbers encoded with 512 bits (which is a realistic number for current standards)

we get that the probability of picking a prime number is

$$Pr(p < 2^{512} \text{ is prime}) = \frac{1}{\ln(2^{512})} = \frac{1}{355}$$

This means that if we take $\frac{355}{2} = 177$ random odd numbers (we don't want to pick even numbers since they are surely not prime), one of them is likely prime. Since 177 isn't that big of a number we can say that, assuming that an efficient primality test is used, we can select a random prime number by picking numbers at random and testing their primality.

Having established that this method works we have to understand how to efficiently test if a number is prime.

10.1.1 Fermat test

The first algorithm we are going to analyse is based on Fermat's little theorem (A.14) which states that

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

with p prime and a invertible. Note that this is simply an implication, hence we write

$$p \text{ prime } \Rightarrow a^{p-1} \equiv 1 \mod p$$

This means that

- we can certainly say that p is not prime (i.e., it's composite) if we find that $a^{p-1} \not\equiv 1 \mod p$ (because if p is prime it immediately follows $a^{p-1} \equiv 1$), and
- we can say with some probability that p is prime if we find that $a^{p-1} \equiv 1 \mod p$.

Since we can't be sure that a number is not prime, we should try and understand what is the probability that p is not prime when using Fermat's little theorem. Luckily, the following theorem gives us the result we need.

Theorem 10.2 (Composability criterion)**.** *It can be shown that if n is composite then*

$$Pr(a^{n-1} \not\equiv 1 \mod n) \geq \frac{1}{2}$$

with $1 < a < n$. The base a is called Fermat witness for the compositeness of n.

Composite prime numbers that pass the Fermat test, i.e., that are marked as prime because $a^{n-1} \equiv 1 \mod n$ even if they are not, are called **Fermat pseudoprimes to base** a and a is called **Fermat liar**. If $gdc(a, n) = 1$, a Fermat pseudoprime is called a Carmichael pseudoprime.

Given an a, we know that the Fermat primality test successfully finds a composite number with a probability greater than $\frac{1}{2}$. This means that given a value of a, a number n is a Fermat pseudoprime with a probability smaller than $\frac{1}{2}$. We can therefore repeat the experiment k times using a different value of a for each experiment. If all experiments are passed, we know that n is a Fermat pseudoprime with a probability smaller than $\left(\frac{1}{2}\right)^k$. As a result, the number n is prime with probability $1 - \left(\frac{1}{2}\right)^k$. Algorithm 3 reports the Fermat primality test we have just described. Using the best algorithms for modular exponentiation, the time complexity of the Fermat primality test is

$$T_{Fermat\ primality} = \mathcal{O}(k \log_2^3 n)$$

bit operations, with $k \in \{1, \ldots, n-1\}$.

Algorithm 3 Fermat primality test

Input: integer to be tested n
Input: probability parameter k
for $i \in \{0, \dots, k-1\}$ **do**
  $a \leftarrow \text{RANDOM}(\{2, \dots, n-1\})$ *s.t.* $gcd(a, n) = 1$
  if $a^{n-1} \not\equiv 1 \mod n$ **then**
    return Composite
  end if
end for
return Prime ▷ with probability $\geq 1 - \frac{1}{2^k}$

10.1.2 Miller-Rabin test

There are infinite Carmichael numbers and, even if they are rarer than prime numbers, we can try and find a better algorithm for testing the primality of a number. An example is the Miller-Rabin test. The Miller-Rabin test is based on two facts.

Lemma 10.1 (Miller-Rabin 1)**.** *Let p be a prime number. Then the equation*

$$x^2 \equiv 1 \mod p \iff (x+1)(x-1) = 0 \mod p$$

has only two solutions $x \equiv \pm 1 \mod p$.

Lemma 10.2 (Miller-Rabin 2)**.** *Given a prime $p > 2$, is always possible to write*

$$p - 1 = d2^s$$

with d odd, $s \geq 1$. Therefore, for each $a \in \mathbb{Z}_p^*$ such that

$$a^{p-1} \equiv a^{d2^s} \equiv 1 \mod p$$

we have either

$$a^d \equiv \pm 1 \mod p$$

or

$$a^{d2^r} \equiv -1 \mod p$$

for at least one value of $1 \leq r \leq s-1$.

As for the Fermat primality test, to test if n is prime we can iteratively pick values of a and,

- if we find a value such that

$$\begin{cases} a^d \not\equiv \pm 1 \mod n \\ a^{d2^r} \not\equiv -1 \mod n \quad \forall 1 \leq r \leq s-1 \end{cases}$$

  then n is for sure composite,
- otherwise we can say that n is prime with some probability.

If we try more values of a the test is more precise since the probability of finding a false positive (i.e., a composite which is marked as prime) decreases. The probability that a composite number doesn't pass the Miller-Rabin test (i.e., that a composite number is correctly marked as composite) is stated by the following theorem.

Theorem 10.3. *For any odd composite n, at least $\frac{3}{4}$ of the bases $2 < a < n$ are witnesses for the composedness of n.*

Hence the probability of finding a liar is $\frac{1}{4}$ after 1 iteration and $\left(\frac{1}{4}\right)^k$ after k iterations. As a result, following the same reasoning done for the Fermat test, we can say that after k iterations, the algorithms correctly classifies a number as prime with probability

$$Pr(p \text{ prime}) \geq 1 - \left(\frac{1}{4}\right)^k$$

If we consider $k = 30$, the probability of getting a false prime is around 10^{-18} which is far smaller than the probability of an hardware failure in any digital circuit, hence we can be sure that the Miller-Rabin test mostly returns correct results.

Algorithm 4 shows an implementation of the Miller-Rabin primality test we have just described.

Algorithm 4 Miller-Rabin primality test

Input: odd integer to be tested n
Input: probability parameter $1 \leq k < n$
Consider $n - 1 = d2^s$, d odd, $s \geq 1$
for $i \in \{0, \ldots, k-1\}$ **do**
  $a \leftarrow$ RANDOM($\{2, \ldots, n-1\}$)
  $x \leftarrow a^d \mod n$
  if $x \not\equiv \pm 1 \mod n$ **then**
    $r \leftarrow 1$
    while $r < s$ **and** $x \not\equiv -1 \mod n$ **do** ▷ We have $a^d \not\equiv \pm 1$, this checks $a^{d2^s} \not\equiv -1$
      $x \leftarrow x^2 \mod n$
      $r \leftarrow r + 1$
    end while
    if $r = s$ **then**
      return Composite
    end if
  end if
end for
return Prime ▷ with probability $\geq 1 - \frac{1}{4^k}$

Using the best algorithms for modular exponentiation, the time complexity of the Miller-Rabin primality test is

$$T_{Miller-Rabin\ primality} = \mathcal{O}(k \log_2^3 n)$$

bit operations.

10.1.3 AKS algorithms

The Fermat and Miller-Rabin tests are probabilistic since they classify a number as prime with a certain probability. In some fields this can't be accepted (e.g., in the military or national defence fields) hence we also need a deterministic algorithm for testing the primality of a number. The best deterministic algorithm is called **AKS** algorithm and takes a polynomial time in the number of bits used to encode the number n to test. Even if this algorithm is has a polynomial complexity, the exponent of the polynomial is quite large, in fact the time complexity of AKS is

$$T_{AKS}(n) = \mathcal{O}\big((\log_2 n)^{7.5}\big)$$

As a result, the AKS algorithm takes way more time than its probabilistic counterparts and for this reason is used only when it's vital to obtain a number that is prime with probability 1. The details of the implementation of AKS are presented in the paper PRIMES in P.

10.2 Factoring algorithms

Many cryptosystems are based on the problem of factoring a large prime number. One famous example is RSA. We said that factoring a large number is computationally hard, however we haven't analysed any algorithm for solving this problem. It is now time to show some algorithm for factoring a number. Considering their complexity, which can be evaluated using the usual formula

$$\mathcal{L}_n(\alpha, \beta) = \exp\left(\big(\beta + o(1)\big)(\log n)^{\alpha}\big(\log(\log n)\big)^{1-\alpha}\right)$$

factoring algorithms can be divided into:

- **Elementary algorithms**. Elementary algorithms are easy to understand and implement however they have an exponential complexity (i.e., $a = 1$). Some examples of elementary algorithms are:
  - **Trivial division**.
  - **Fermat's factorisation algorithm**.
  - **Pollard's $P - 1$ method**.
  - **Pollard's rho method**.
- **Complex algorithms**. Complex algorithms are more complex than elementary algorithms however they have a sub-exponential complexity (i.e., $0 < a < 1$). The most famous factoring algorithm is the **General Number Field Sieve** (GNFS), which has a time complexity of

  $$T_{GNFS}(n) = \mathcal{O}\left(\mathcal{L}_n\left(\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right)\right)$$

  The GNFS is based on the **linear sieve** which has a worse complexity with respect to the GNFS (but still sub-exponential) but is easier to analyse.

10.2.1 Fermat's factorisation algorithm

Fermat's factorisation algorithm is based on the following theorem.

Theorem 10.4. *Let n be an odd integer number. Then n can be written as the difference between two squares,*

$$n = x^2 - y^2 = (x+y)(x-y)$$

If neither factor $(x+y)$ and $(x-y)$ is equal to 1, then $(x+y)(x-y)$ is a proper factorisation of n.

Now say $n = c \cdot d$. Then we can write

$$\begin{aligned} n &= \left(\frac{1}{2}(c+d)\right)^2 - \left(\frac{1}{2}(c-d)\right)^2 \\ &= \frac{1}{4}(c+d)^2 - \frac{1}{4}(c-d)^2 \\ &= \frac{1}{4}(c^2+d^2+2cd) - \frac{1}{4}(c^2+d^2-2cd) \\ &= \frac{1}{4}c^2 + \frac{1}{4}d^2 + \frac{1}{4}2cd - \frac{1}{4}c^2 - \frac{1}{4}d^2 + \frac{1}{4}2cd \\ &= cd \end{aligned}$$

Since n is odd, then also c and d must be odd. If we sum or subtract two odd numbers we obtain an even number, hence dividing $(c+d)$ and $(c-d)$ by 2 yields an integer value.

The idea behind Fermat's factorisation algorithm is to invert the equation of Theorem 10.4 to write

$$y^2 = x^2 - n$$

The algorithm tries different values of x until $x^2 - n$ is a perfect square. When we've found a y^2 which is a perfect square we can compute y by extracting the square root. The complexity of this algorithm can be influenced by the complexity of checking if a number is a square root. Luckily, there exist algorithms that check if a number is a perfect square in constant time. In particular, a perfect square has only 22 possible values for the last two hexadecimal digits, hence we can check if the last two hexadecimal digits of a number belong to this list. The list of possible values is

$$00, e1, e4, 25, o6, e9$$

where e stands for an even number and o for an odd number.

Algorithm 5 shows an implementation of Fermat's factorisation algorithm we have described. Note that the algorithm takes $\lceil\sqrt{n}\rceil$ as the first value of x since using a value smaller than this would yield a negative value for y^2.

Fermat's factorisation algorithm has a complexity of

$$T_{Fermat\ factorisation}(n) = \mathcal{O}(2^{\frac{\log_2(n)}{2}})$$

and works slowly when the factors are not near $\sqrt{n}$.

10.2.2 Pollard's rho method

Pollard's rho method is based on two observations:

1. Say λ is a factor of n. If we find two values a and b such that $a \equiv b \mod \lambda$ (i.e., λ is a factor of $a-b$), then λ is the greatest common divisor of $a-b$ and n.

Algorithm 5 Fermat's factorisation algorithm.

Input: a composite integer n
$x \leftarrow \lceil \sqrt{n} \rceil$
$\hat{y} \leftarrow x^2 - n$
while $\hat{y}$ not a perfect square **do**
  $x \leftarrow x + 1$
  $\hat{y} \leftarrow \hat{y} + 2x - 1$ ▷ $\hat{y} \leftarrow x^2 - n$
end while
return $(x - \sqrt{\hat{y}}, x + \sqrt{\hat{y}})$

2. Two random numbers a, b are equivalent modulo λ with probability $\frac{1}{2}$ after around $\sqrt{\lambda}$ have been generated. This comes from the birthday paradox.

From the first observation, we know that $\lambda = gcd(a - b, n)$ means that we can write $n = \lambda \cdot k$, hence λ is a factor of n. As a result, we can use the following strategy to find a factor of a number n:

1. Pick two values $a, b \in \{1, \ldots, n-1\}$ at random.
2. Check if $gcd(a - b, n) \neq 1$.
3. Go to step 1 if $gcd(a - b, n) = 1$.
4. Use $\lambda = gcd(a - b, n)$ as factor of n.

We can immediately understand that this process is over-complicated, in fact picking two values a, b completely at random is useless. We can optimise this process and pick numbers such that we always have $a \equiv b \mod \lambda$ and we don't have to recompute $gcd(a - b, n)$ for values that have already been computed.

Computing random values for the Pollard's rho method

Let us consider a function

$$f : \{1, \ldots, n-1\} \to \{1, \ldots, n-1\}$$

An example of such a function is $x^2 + 1 \mod n$. This function can be used to generate a pseudo-random sequence of points $x_1, x_2, \ldots$. In particular, the point i is generated by passing the previous point x_{i-1} to the function,

$$x_i = f(x_{i-1})$$

The values generated by this procedure are used to test if $gcd(x_i, x_j) \neq 1$. Since the function f returns, by definition, points in $\{1, \ldots, n-1\}$, then at some point we will generate a number which has already been generated and the procedure will eventually generate the same sequence of numbers. Figure 10.1 shows a pattern in a sequence of numbers. By looking at this representation we understand that this method is called rho because the shape of the pattern is similar to the Greek letter ρ.

We can exploit this property to understand when we can stop computing a *gcd*s since we have already computed it for a couple (x_i, x_{i+1}). In particular, we should compute the periodicity of the repetition and recognise the pattern of numbers generated by the procedure. We can use Floyd's cycle-finding algorithm, also called the tortoise and hare algorithm, to determine if a sequence of numbers is cycling, and its periodicity.

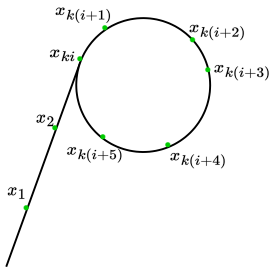


Figure 10.1: A pattern in a sequence of numbers generated by the Pollard's rho method.

Algorithm 6 Pollard's rho algorithm.

Input: a composite integer n
$x \leftarrow 2$ ▷ $x_1 \leftarrow 2$
$y \leftarrow 2$
$\lambda \leftarrow 1$
while $\lambda = 1$ **do**
  $x \leftarrow x^2 + 1 \mod n$ ▷ $x_{i+1} \leftarrow f(x_i)$
  $y \leftarrow (y^2 + 1 \mod n)^2 + 1 \mod n$ ▷ $y_{i+1} \leftarrow f\big(f(x_i)\big)$ for the tortoise-hare
  $\lambda \leftarrow \mathrm{GCD}(|x - y|, n)$
  if $\lambda = n$ **then**
    return Failure ▷ restart with a different x and y
  end if
end while
return λ

Algorithm 6 shows the Pollard's rho method as we have described it.
Pollard's rho method has a time complexity of

$$\mathcal{O}(\sqrt[4]{n}\log_2^2 n)$$

bit operations, thus it's asymptotically better than both Fermat's and trivial division methods.

10.2.3 Pollard's P-1 method

Pollard's $P - 1$ method is particularly fast to find the small prime factors of a composite number. This method is based on the idea of smooth and power-smooth numbers.

Definition 10.1 (Smooth number)**.** *Let B and n be integer numbers. The number n is called B-smooth if every prime factor of n is smaller than B. Formally, a number*

$$n = \prod_{i=0}^{N} b_i^{e_i}$$

is $min(\{b_1, \ldots, b_N\})$-smooth.

For instance, $n = 2^{78} \cdot 3^{89} \cdot 11^3$ is 12-smooth.

Definition 10.2 (Power smooth number)**.** *Let B and n be integer numbers. The number n is called B-power-smooth if every prime power of n is smaller than B. Formally, a number*

$$n = \prod_{i=0}^{N} b_i^{e_i}$$

is $min(\{b_1^{e_1}, \ldots, b_N^{e_N}\})$-smooth.

For instance, $n = 2^5 \cdot 3^3 \cdot 11 = 32 \cdot 27 \cdot 11$ is 33-power-smooth.

Let us now understand how the smoothness of a number can be used to factorise it. Pollard's $P-1$ method works as follows:

1. Take $n = p \cdot q$ with p and q prime.
2. Guess B such that $p-1$ is B-power-smooth but $q-1$ is not. A possible choice for B would be $B = \log_2^c(n)$ for some c (e.g., $0 < c \leq 6$).
3. $p-1$ divides $B!$ because $B!$ is the product of all values from 1 to B, hence it will also contain the prime powers of $p-1$ as factors (since these powers are smaller than B by definition of p as B-power-smooth). Namely, we can write $B!$ as

   $$B! = B \cdot (B-1) \cdot \ \cdots \ \cdot b_1^{e_1} \cdot \ \cdots \ \cdot (B-k) \cdot \ \cdots \ \cdot b_2^{e_2} \cdot \cdots$$

   As a result, if we take $a \equiv 2^{(B!)} \mod n$, then we have $a \equiv 1 \mod p$ because

   $$2^{\frac{B!}{p-1} \cdot (p-1)} = \left(2^{\frac{B!}{p-1}}\right)^{p-1} = 1 \mod p$$

   for Fermat's theorem (given that $2^{\frac{B!}{p-1}} \in \mathbb{Z}_p^*$), and $a \not\equiv 1 \mod q$. This means that p divides $a-1$ and q doesn't.
4. Recover the factor p as:

   $$p = gcd(a-1, n)$$

   if the result is

   $$1 < gcd < n$$

   otherwise, guess another value for B.

The computational complexity of Pollard's $P-1$ method is

$$T_{Pollard\ P-1}(n) = \mathcal{O}(B \log_2(B) log_2^2(n))$$

If the prime numbers p and q, Pollard's $P-1$ method can be used to obtain the factors of n. To make it difficult to use this method, we can pick primes numbers such that

$$p = 2p_1 + 1$$

and

$$q = 2q_1 + 1$$

with p_1 and q_1 also primes. Nowadays, this recommendation isn't required anymore since the prime p is so large that the Pollard's $P-1$ method can't be efficiently used to obtain p and q.

10.2.4 Linear sieve

10.3 Discrete logarithm extraction algorithms

Cryptosystems that don't exploit the factorisation problem are usually based on the discrete logarithm problem. Then, we should describe how we can solve the discrete logarithm problem. As for factoring algorithms, we can divide discrete logarithm extraction algorithms into:

- **Elementary algorithms**. Elementary algorithms work in generic cyclic groups G with order n, hence in elliptic curves, too. Some examples of elementary algorithms are:
  - **The Baby step giant step algorithm**.
  - **The Pollard's rho algorithm**.
  - **The Pohlig-Hellman algorithm**.
- **Complex algorithms**. Complex algorithms have a sub-exponential complexity (i.e., $0 < a < 1$) when used in finite fields but have an exponential complexity when used on elliptic curves. The algorithms are usually called index-calculus and are usually based on complex algorithms for solving the factoring problem, usually on the general number field sieve. The time complexity of complex algorithms for solving the DLP in finite fields is

  $$T_{DLP,complex}(n) = \mathcal{O}(L_p(\frac{1}{3}, c))$$

  where c depends on the field. In particular, we have $c = 1.578$ in $\mathbb{F}_{2^m}$ and $c = 1.923$ in $\mathbb{F}_{p^m}$ with $p > 2$.

10.3.1 Baby step, giant step algorithm

The baby step, giant step algorithm is a refinement of the bruteforce algorithm for computing the discrete logarithm and uses a man-in-the-middle approach.

Let us consider a generic cyclic group $(G, \cdot)$ with order $n = |G|$ and generator g. Say we take an element β of the group. Then β, since it belongs to G, can be written as g^x since g is a generator of the group. We don't know the value of x but we know that it belongs to $\{0, \dots, n-1\} = \mathbb{Z}_n$.

Since $x \in \{0, \dots, n-1\}$ we can write x as

$$x = i \cdot \lceil \sqrt{n} \rceil + j$$

with $i, j \in \{0, \dots, \lceil \sqrt{n} \rceil - 1\}$. In fact, the maximum value for x is obtained when $i = j = \lceil \sqrt{n} \rceil - 1$ for which we obtain

$$\begin{aligned} x &= i \cdot \lceil \sqrt{n} \rceil + j \\ &= (\lceil \sqrt{n} \rceil - 1) \cdot \lceil \sqrt{n} \rceil + (\lceil \sqrt{n} \rceil - 1) \\ &= (\lceil \sqrt{n} \rceil)^2 - \lceil \sqrt{n} \rceil + \lceil \sqrt{n} \rceil - 1 \\ &= (\lceil \sqrt{n} \rceil)^2 - 1 \end{aligned}$$

hence we consider every possible value of x (and even something more in some cases). If we replace this new writing for x in the equality $\beta = g^x$ we get

$$\begin{aligned}\beta &\equiv g^{i\cdot\lceil\sqrt{n}\rceil+j}\\ \beta &\equiv g^{i\cdot\lceil\sqrt{n}\rceil}g^j\\ g^j &\equiv \beta\cdot g^{-i\cdot\lceil\sqrt{n}\rceil}\\ g^j &\equiv \beta\cdot (g^{-\lceil\sqrt{n}\rceil})^i\end{aligned}$$

The left-hand side g^j is called **baby step** and the right-hand side is called **giant step**. The algorithm pre-computes the values of the baby step, which are $\sqrt{n}$ in total. Then, the algorithm tries different values of i of the giant step until the equivalence holds. Note that $g^{-\lceil\sqrt{n}\rceil}$ can be pre-computed, hence we only have to rise this value to the power of i and then multiply this result by β.

Algorithm 7 shows the baby step, giant step algorithm we have described. The algorithm still works even if $n \geq |G|$. The time complexity of the baby step, giant step algorithm is

$$T_{baby\ step\ giant\ step}(n) = \mathcal{O}(\sqrt{n})$$

and has the same space complexity.

Algorithm 7 The baby step, giant step algorithm.

Input: $G = \langle g\rangle$ of order n
Input: β
$m \leftarrow \lceil\sqrt{n}\rceil$
for $j \in \{0,\dots,m\}$ **do**
  $B_j \leftarrow g^j$ ▷ Compute the baby steps
end for
$y \leftarrow \beta$
for $i \in \{0,\dots,m\}$ **do**
  if $y \in B$ **then**
    return $(i\cdot m + j) \mod n$
  end if
  $y \leftarrow y\cdot g^{-m}$
end for

10.3.2 Pollard's rho algorithm

The Pollard's rho algorithm achieves the same time complexity of the baby step, giant step algorithm, however it requires constant space. Let us consider a generic cyclic group $(G,\cdot)$ with order $n = |G|$ and generator g. Let x be the discrete logarithm we want to find, namely

$$x \equiv \log_g^D h \mod n$$

hence

$$h \equiv g^x \mod n$$

We can now partition the group G into three sets S_1, S_2 and S_3, which are not necessarily subgroups of G. Namely, we get

$$G = S_1 \cup S_2 \cup S_3$$

Let us also assume that $1 \notin S_2$. These groups are used to define a random sequence of values in G as

$$x_{i+1} = f(x_i) = \begin{cases} h \cdot x_i & \text{if } x_i \in S_1 \\ x_i^2 & \text{if } x_i \in S_2 \\ g \cdot x_i & \text{if } x_i \in S_3 \end{cases}$$

starting from $x_0 = 1$. In practice, we assume that we can write x_i as

$$x_i = g^{a_1} \cdot h^{b_1}$$

and keep track of x_i, a_i and b_i, starting from $(1, 0, 0)$. The values of a_i and b_i can be computed as

$$a_{i+1} = f^{(a)}(a_i, x_i) = \begin{cases} a_i \mod n & \text{if } x_i \in S_1 \\ 2a_i \mod n & \text{if } x_i \in S_2 \\ a_i + 1 \mod n & \text{if } x_i \in S_3 \end{cases}$$

and

$$b_{i+1} = f^{(b)}(b_i, x_i) \begin{cases} b_i + 1 \mod n & \text{if } x_i \in S_1 \\ 2b_i \mod n & \text{if } x_i \in S_2 \\ b_i \mod n & \text{if } x_i \in S_3 \end{cases}$$

If we apply the logarithm base g to the equality $x_i = g^{a_1} \cdot h^{b_1}$ we obtain, for all i,

$$\begin{aligned} \log_g(x_i) &= \log_g(g^{a_i}) + \log_g(h^{b_i}) \\ &= a_i + \log_g(h^{b_i}) \\ &= a_i + \log_g(h^{b_i}) && \log_h(h^{b_i}) = \frac{\log_g(h^{b_i})}{\log_g(h)} \\ &= a_i + \log_h(h^{b_i}) \cdot \log_g(h) \\ &= a_i + b_i \cdot \log_g(h) \\ &= a_i + b_i \cdot x \end{aligned}$$

The functions f, $f^{(a)}$ and $f^{(b)}$ define three different random sequences of numbers. The Pollard's rho method for solving the discrete logarithm problem is based on the same algorithm for solving the factoring problem, hence we have to use recognise cycles, too. The algorithm uses the tortoise and hare method, hence we have to compute the double pointers, too. In particular, if we call x the tortoise and dx the hare, we get (thanks to the Floyd's cycle finding condition) that the tortoise and the hare meet at a certain step $s > 0$ (i.e., we have a loop) when,

$$g^{a_s} \cdot h^{b_s} = g^{a_{2s}} \cdot h^{b_{2s}}$$

namely,

$$x_s = dx_{2s} \iff g^{a_s} \cdot h^{b_s} = g^{a_{2s}} \cdot h^{b_{2s}}$$

If we remember that $h \equiv g^x$ we get

$$\begin{aligned} x_s = dx_{2s} &\iff g^{a_s} \cdot g^{xb_s} = g^{a_{2s}} \cdot g^{xb_{2s}} \\ &\iff g^{a_s + xb_s} = g^{a_{2s} + xb_{2s}} \end{aligned}$$

By moving to the exponents, we can write

$$\begin{aligned} a_s + xb_s &\equiv a_{2s} + xb_{2s} \mod n \\ a_s - a_{2s} &\equiv xb_{2s} - xb_s \mod n \\ -(a_{2s} - a_s) &\equiv x(b_{2s} - b_s) \mod n \end{aligned}$$

As a result, we can write x as

$$x \equiv -\frac{a_{2s} - a_s}{b_{2s} - b_s}$$

Note that if $b_{2s} = b_s$, this method fails. Luckily, for a large value of n the probability of such an event is negligible.

Algorithm 8 shows the Pollard's rho algorithm we have just described.

Algorithm 8 The Pollard's rho algorithm.

Input: $G = \langle g \rangle$ of order n
Input: $h \in G$
$a \leftarrow 0$
$b \leftarrow 0$
$x \leftarrow 1$
$da \leftarrow 0$
$db \leftarrow 0$
$dx \leftarrow 1$
$i \leftarrow 0$
while true do
  $a \leftarrow f^{(a)}(a, x)$
  $b \leftarrow f^{(b)}(b, x)$
  $x \leftarrow f(x)$
  $da \leftarrow f^{(a)}\big(f^{(a)}(da, dx), f(dx)\big)$
  $db \leftarrow f^{(b)}\big(f^{(b)}(db, dx), f(dx)\big)$
  $dx \leftarrow f\big(f(x)\big)$
  if $b = db$ **then return** Failure
  end if
  if $x = dx$ **then return** $\left(-\frac{da-a}{db-b}\right) \mod n$
  end if
  $i \leftarrow i + 1$
end while

10.3.3 The Pohlig-Hellman algorithm

The Pohlig-Hellman algorithm is based on the following lemma.

Lemma 10.3 (Pohlig-Hellman)**.** *Let G be a finite cyclic group with order $n = |G|$ and generator g (i.e., $G = \langle g \rangle$). Assume that n is B-smooth with $B \leq \log_2^c(n)$. Then we know*

the factorisation of n, hence we can write

$$n = \prod_i^s p_i^{e_i}$$

with $s \geq 1$, $e_i \geq 1$ and known p_i, e_i. Moreover, the computation of

$$x = \log_g \beta \qquad \beta \in G$$

will require $\mathcal{O}(\sqrt{B})$ operations.

Proof. We want to compute $x \equiv \log_g \beta \mod n$. Since we know the factorisation of n, because it's B-smooth, then we can write the following system:

$$\begin{cases} x \equiv x_1 \mod p_1^{e_1} \\ x \equiv x_2 \mod p_2^{e_2} \\ \vdots \\ x \equiv x_s \mod p_s^{e_s} \end{cases}$$

with x_i unknown. Thanks to the Chinese remainder theorem (A.7.3) we can solve this system and find x as

$$x \equiv \sum_{i=1}^{s} x_i \left(\frac{n}{p_i^{e_i}}\right)\left(\left(\frac{n}{p_i^{e_i}}\right)^{-1} \mod p_i^{e_i}\right) \mod n$$

This means that if we know the values of x_i we can solve the discrete logarithm problem by applying the formula above. We have reduced the discrete logarithm problem to the problem of finding the x_is. Note that finding the x_is is a sub-problem of the original problem but computed in a smaller space. Let us then focus on one equation. Since $x \equiv \log_g \beta$, we want to compute

$$x_i \equiv x \mod p_i^{e_i} \equiv \log_g \beta \mod p_i^{e_i}$$

Now we can write x_i in base (i.e., radix) p_i. This means that we can write x_i as

$$x_i = l_0 + l_1 p_i + l_2 p_i^2 + l_3 p_i^3 + \cdots + l_j p_i^l + \cdots + 0 p_i^{e_i} \qquad 0 \leq l_j < p_i$$

Thanks to this representation we can compute $l_0, l_1, l_2, \ldots$ iteratively, each time solving the discrete logarithm problem in a smaller group. Let's start by computing l_0. For starters, we have to compute

$$\eta = g^{\frac{n}{p_i}}$$

take $\gamma_0 = 1$ and consider

$$\delta_0 \equiv \left(\beta \gamma_0^{-1}\right)^{\frac{n}{p_i^1}} \mod p_i$$

Now we can write

$$\delta_0 \equiv \left(g^x\right)^{\frac{n}{p_i^1}} \mod p_i$$

Since we know that $x_i \equiv x \mod p_i^{e_i}$, we can write x as

$$x = q \cdot p_i^{e_i} + x_i$$

Replacing this value in the equation above we get

$$\begin{aligned}\delta_0 &\equiv \left(g^x\right)^{\frac{n}{p_i^1}} \mod p_i \\ &\equiv \left(g^{q\cdot p_i^{e_i}+x_i}\right)^{\frac{n}{p_i^1}} \mod p_i \\ &\equiv \left(g^{q\cdot p_i^{e_i}\cdot\frac{n}{p_i^1}}\right)\cdot\left(g^{x_i\frac{n}{p_i^1}}\right) \mod p_i\end{aligned}$$

Now we can notice that $g^{q\cdot p_i^{e_i}\cdot\frac{n}{p_i^1}} \equiv 1$, hence we can write

$$\begin{aligned}\delta_0 &= g^{x_1\frac{n}{p_i^1}} \\ &= g^{(l_0+l_1p_i+l_2p_i^2+\dots)\frac{n}{p_i^1}} \\ &= g^{l_0\frac{n}{p_i^1}} \\ &= \left(g^{\frac{n}{p_i^1}}\right)^{l_0}\end{aligned}$$

By definition of discrete logarithm, this means that

$$\begin{aligned}l_0 &\equiv \log_{g^{\frac{n}{p_i^1}}}\left(\delta_0\right) \mod p_i \\ &\equiv \log_\eta\left(\delta_0\right) \mod p_i\end{aligned}$$

This means that we can compute l_0 as $\log_\eta\left(\delta_0\right) \mod p_i$, which can be done using an elementary algorithm if p_i is small enough.

Now we can compute

$$\gamma_1 = \gamma_0 g^{l_0 p_i^0}$$

and consider

$$\delta_1 = \left(\beta\gamma_1^{-1}\right)^{\frac{n}{p_i^2}}$$

These values can be used to compute l_1 in a similar fashion to what we did for l_0. In particular, we can write

$$\begin{aligned}\delta_1 &= \left(g^{x-x_0}\right)^{\frac{n}{p_i^2}} \\ &= g^{(l_1p_i+l_2p_i^2+\dots)\frac{n}{p_i^2}} \\ &= g^{l_1\frac{n}{p_i}} \\ &= \left(g^{\frac{n}{p_i}}\right)^{l_1}\end{aligned}$$

As before, this means that we can compute

$$l_1 \equiv \log_\eta\left(\delta_1\right) \mod p_i$$

The same approach is repeated for l_2. We compute

$$\gamma_2 = \gamma_1 g^{l_1 p_i^1} = g^{l_0+l_1p_i}$$

and consider

$$\delta_2 = \left(\beta\gamma_2^{-1}\right)^{\frac{n}{p_i^3}}$$

These values can be used to compute l_2. In particular, we can write

$$\begin{aligned}\delta_2 &= \left(g^{x-l_0-l_1 p_i}\right)^{\frac{n}{p_i^2}} \\ &= g^{(l_2 p_i^2 + \dots)\frac{n}{p_i^3}} \\ &= g^{l_2 \frac{n}{p_i}} \\ &= \left(g^{\frac{n}{p_i}}\right)^{l_2}\end{aligned}$$

As before, this means that we can compute

$$l_2 \equiv \log_\eta \left(\delta_2\right) \mod p_i$$

□

Algorithm 9 shows the Pohlig-Hellman algorithm we have just described.

Algorithm 9 The Pohlig-Hellman algorithm.

Input: Group $G = \langle g \rangle$ with $n = |G|$ and B-smooth.
Input: $\beta \in G$
for $i \in \{0, \dots, s\}$ **do**
  $\gamma \leftarrow 1$
  $\eta \leftarrow g^{\frac{n}{p_i}}$
  for $j \in \{0, \dots, e_i - 1\}$ **do**
    $\gamma \leftarrow \gamma \cdot g^{l_{j-1} p_i^{j-1}}$
    $\delta \leftarrow \left(\beta \cdot \gamma^{-1}\right)^{\frac{n}{p_i^{j+1}}}$
    $l_j \leftarrow \log_\eta \delta$ ▷ Using elementary algorithm
  end for
  Reconstruct $x_i \leftarrow l_0 + l_1 p_i + l_2 p_i^2 + \dots + l_{e_i - 1} p_i^{e_i - 1}$
end for
$x \equiv \sum_{i=1}^{s} x_i \left(\frac{n}{p_i^{e_i}}\right)\left(\left(\frac{n}{p_i^{e_i}}\right) \mod p_i^{e_i}\right) \mod n$ ▷ Apply Chinese remainder theorem
return x

Since the intermediate steps in the Pohlig–Hellman algorithm are quite simple, the difficulty of solving a discrete logarithm problem is dominated by the time required to solve the DLP in the cyclic subgroups of prime order. If we use the big step, giant step algorithm, solving a DLP has a complexity of $\mathcal{O}(\sqrt{p_i})$. Given the order of the group $n = |G|$, if $n = \prod_i^s p_i^{e_i}$ has a B-smooth factorisation then the running time of the Pohlig-Hellman algorithm is given by:

$$\mathcal{O}\left(\sum_{i=1}^{s} e_i \cdot \left(\log_2 p_i + \sqrt{p_i}\right)\right) = \mathcal{O}\left(s \cdot \max e_i {}_{i=1}^{s} \cdot \left(\log_2 n + \sqrt{B}\right)\right)$$

bit operations.

Part II

Security protocols

Chapter 11

Secure communication

11.1 TLS

The Transport Layer Security (TLS) protocol is the standardised evolution of the Secure Sockets Layer (SSL) protocol, originally invented for securing HTTP over TCP. More precisely, IETF, given the popularity of SSL and the need for a standard protocol for secure communication, decided to fix some bugs of SSL v3 (i.e., the third version of SSL, even though SSL v1 has never been released) and publish it as a standard under the name of TLS (version 1).

Even though TLS is an evolution of SSL, the two protocols are not interoperable. However, it's possible to downgrade TLS v1 (until v1.2) to SSL v3 for back-compatibility purposes. This means that a machine that doesn't support TLS can ask another machine to use SSL instead of TLS for communicating. This is not great for security since SSL includes bugs solved in TLS, hence the former is less secure. In any case, we need to be able to downgrade to support communication with legacy services.

Since its release, TLS has been upgraded three times: from v1 to v1.1, from v1.1 to v1.2 and from v1.2 to v1.3. The first two updates fix important security issues but the last one is a massive code cleanup. The goal of the last update is to remove all those mechanisms that allowed admins to do damage by misconfiguring something. Namely, v1.3 doesn't solve massive security bugs but makes the protocol less vulnerable to misconfiguration errors that cause vulnerabilities. For instance, version 1.2 still allowed to use of DES as a symmetric cypher. This isn't a vulnerability on its own since it's enough to configure the server to accept only AES or other secure symmetric cyphers, however, an admin could misconfigure a server and allow DES to be used, which is a security threat.

11.1.1 Protocol overview

Since the basic structure of versions 2 and 3 is the same, we will focus on TLS v2. TLS uses a hybrid encryption scheme:

- **Asymmetric encryption** is used for
  - **authentication** using X.509 certificates (i.e., PKI), and
  - **key exchange** using the Diffie-Hellman protocol or encapsulation (i.e., sending the shared key encrypted with the public key).

- **Symmetric encryption** is used for **session encryption**, namely for encrypting every message after the handshake.

TLS (up to v1.2) also offers optional integrity via secret key authentication or MACs. This is important because we might be using a tamperable mode of operation (e.g., counter mode) without ensuring integrity. This means that we have to be extra careful about what symmetric cypher we choose in version 1.2 or lower of the TLS protocol. Integrity can however be ensured only after the handshake.

The lack of integrity has been abused, in combination with some padding schemes, to reveal the secret key (i.e., break confidentiality). An example of this kind of attack is BEAST. In practice, if we can tamper with the message and the padding of the last block is not defined as a recognisable pattern, then we can use the server as a decryption oracle. We send some messages and if the server doesn't complain it was a valid decrypted message. If the server complains, we get the information that it was not a valid decrypted message, hence we get a partial decryption oracle. As a result, if the encryption mode is not secure against active attacks, the protocol is not secure (use Galois-Counter mode to ensure authentication and integrity). This resulted in mandating authenticated encryption modes in TLS 1.3.

Suites

TLS supports different protocols for symmetric and asymmetric encryption, hashing and MAC. These protocols are called suites (or cypher-suites). A suite is specified by the string

```
TLS_A_WITH_B_C
```

where

- `A` is the signature algorithm used for authenticating the key exchange. Here both the signature and the key exchange algorithms are specified. An example is `DH_RSA` if we want to use RSA certificates to authenticate a Diffie-Hellman key agreement. Elliptic-curve Diffie-Hellman is also available (`ECDH_*`). Authentication can be skipped when using Diffie-Hellman by writing `DH_anon`.
- `B` is the stream cypher or block cypher algorithm with its mode of operation and padding strategy. For example, we can write `AES_GCM` if we want to use AES with Galois Counter Mode. Note that we can also write `NULL` to say that we don't want to encrypt the conversation (i.e., we don't want to ensure confidentiality).
- `C` is the hash algorithm. Hashing is used to derive a session key. In fact, if we use Diffie-Hellman to agree on a session key, the generated key is too long for a symmetric cypher, hence we can hash it to make it shorter. We can also specify the MAC algorithm used if we want to ensure integrity. Note that SHA1 is denoted with `SHA` whereas SHA2 is denoted with `SHA2`. Moreover, `MD2`, `MD4` and `MD5` are advised against since TLS v1.1 but not prohibited.

An example of suite is

```
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
```

11.1.2 Handshake with server-only authentication

First, we'll analyse the TLS handshake in which only the server authenticates (i.e., the client checks the server's identity). Since TLS is based on TCP, the first part of the TLS handshake is the TCP three-way handshake. The full handshake works as follows:

1. The client sends a `ClientHello` message containing, in clear,
   - the highest TLS version supported,
   - the supported cypher suites, and
   - a random number (to avoid replay attacks).
2. The server sends a `ServerHello` message containing, in clear,
   - the chosen TLS version and cypher-suite, and
   - a session nonce (added when moving from SSL to TLS).
3. The server sends a `ServerCertificate` message containing an RSA certificate (signed by a CA) containing the server's public key used for signing messages with RSA.
4. The server sends a `ServerKeyExchange` (only when using ECDH or DH) containing
   - the parameters for the Diffie-Hellman key exchange,
   - the public part of the Diffie-Hellman key exchange (i.e., g^s), and
   - a digital signature, which is the hash of the parameters signed with the server's private key.
5. The server sends a `ServerHelloDone` message.
6. The client:
   (a) verifies the certificate of the server using the PKI and the signature,
   (b) computes the pre-master key (pre-master secret) which is
      - g^{sc} if we are using Diffie-Hellman, or
      - a half-encrypted pre-master key sent by the server, or
      - a key picked by the client and encrypted with the server's public key if we are using RSA only.
   (c) computes the session key (or master secret) using the nonce sent by the server, the nonce picked by the client and the pre-master key (by hashing the pre-master key with the nonces).
7. The client sends a `ClientKeyExchange` message containing
   - g^c if we are using Diffie-Hellman, or
   - the pre-master key encrypted with the authenticated server's public key (using a different public key that the server gives us, with a certificate, to encrypt the session secrets).
8. The client sends a `ChangeCipherSpec` message to signal that it has everything it needs (since it has computed the master key) and it will start encrypting every message from now on.

9. The client sends a `Finished` message which contains a summary of the messages exchanged so far, encrypted with the symmetric cypher initially established.
10. The server:
    (a) computes the pre-master key (e.g., g^{sc} when using DH).
    (b) computes the session key using the nonces and the pre-master key,
11. The server sends a `ChangeCipherSpec` message to notify that the communication will be encrypted from now on.
12. The server sends a `Finished` message containing a MAC (i.e., a summary that can't be tampered with) of all the previous messages.

The handshake is 3 round-trips long considering the TCP handshake, otherwise, it's only 2 round-trips long. TLS 1.3 reduces this time to 1 round-trip for the TLS handshake (without TCP). Figure 11.1 shows the exchange of messages between client and server.

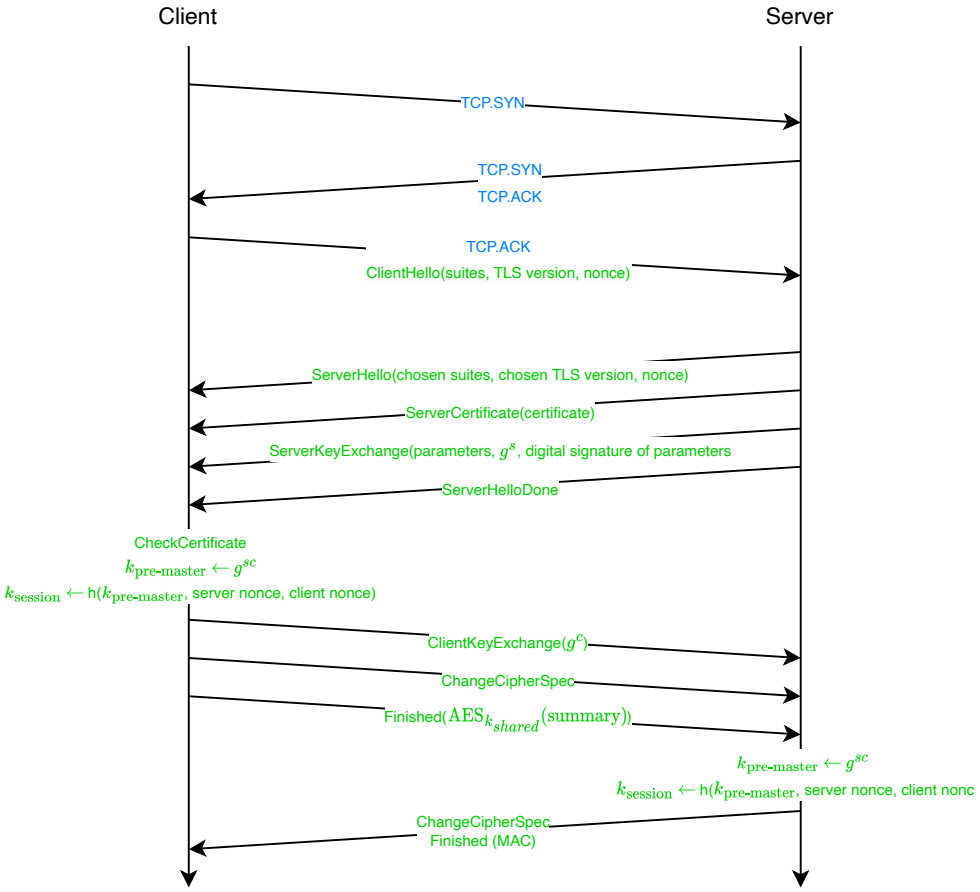


Figure 11.1: The messages exchanged between client and server in the TLS handshake.

Handshake with mutual authentication

In the protocol we have described above only the server authenticates. This means that the server doesn't verify the identity of the client (i.e., the client isn't authenticated). In some cases, it's useful to authenticate the client, too. For instance, authenticating a client allows users to log in to a website without credentials. This approach is however rarely used since it's hard for clients to set up their machine for mutual authentication since they have to generate a certificate which has to be signed by a CA and published. Anyways, TLS supports mutual authentication.

Mutual authentication is achieved through the client sending his own certificate before the `ClientKeyExchange` message. After sending the `ClientKeyExchange` message, the client sends a `CertificateVerify` message, which is a signature over all the previous messages, allowing the server to check that the one who initiated the communication was effectively the client. In this case, both the server and the client can close the connection if the identity of the other is not the correct one preventing impersonation attacks.

Figure 11.2 shows the messages exchanged during the TLS handshake with mutual authentication.

11.1.3 Issues

TLS is a key protocol since it secures communication over the Internet, however, it has its issues. The main are:

- **Public key cryptography is expensive and slow**. Since TLS uses asymmetric encryption for exchanging session keys (Diffie-Hellman or RSA encapsulation), if we use short sessions for security reasons, then the protocol becomes really expensive since we have to use asymmetric encryption at the beginning of every session. To solve this problem, TLS implements **fast session resume** which allows to resume a session. When a client disconnects, the server doesn't delete the session key, hence, upon reconnecting, the client skips the handshake and sends the client hello and the session id and starts encrypting. If the server recognises the session id, it can use the session key previously used hence the session is resumed. The authentication between the parties is implicitly handled by the fact that they both know the same session secret. Note that the session can't last too long, hence after some time the server should delete the session key and refuse to restart a session.

- **No forward secrecy is enforced**. An attacker who compromises a machine can read (i.e., decrypt) every message decrypted so far since the attacker can recover the session key. We can use the ephemeral Diffie-Hellman protocol to solve this problem. This variation of the classical protocol doesn't store the private number on the machine, i.e., Alice doesn't store a, which is deleted after computing $k_{session} \equiv g^{ab} \mod n$. As a result, an attacker that reads g^b on the public channel and compromises Alice's machine, can't decrypt and read the messages sent and received by Alice since a can't be found on Alice's machine and the attacker can't compute g^{ab}. Note that the same principle (i.e., deleting unused secrets) is not applied to RSA encapsulation, too. This isn't done for security reasons but for performance reasons. In fact, picking an RSA key is much more expensive (it might take seconds), hence we might want to reuse some keys. In general, it's always better to use Ephemeral Diffie-Hellman when available since it's the only one ensuring forward secrecy.

- **Weak cypher-suites** can be used. Weak cyphers like DES, which in version 1.2 are still available, if used allow attackers to break confidentiality, authentication or integrity.

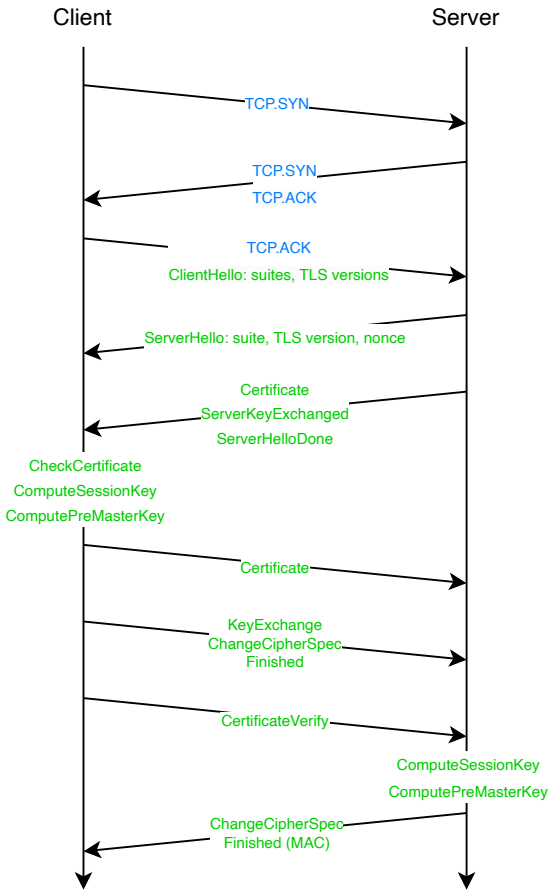


Figure 11.2: The messages exchanged between client and server in the TLS handshake with mutual authentication.

- **Cypher-suite downgrade** is possible. An attacker may force a downgrade of the cipher-suite altering the `ClientHello` message. The only solution to this problem is removing the vulnerable suites.
- **Cypher renegotiation** is possible. Cypher renegotiation allows an attacker to mount a reply attack, in fact, he or she can send a new cypher-suite using the correct messages he or she has sniffed over the network. Cypher renegotiation also allows an attacker to splice a request into a TLS setup. The solution to this problem, standardised in RFC5746 and implemented in the most popular libraries, is to force each entity to use only secure cyphers.
- Some versions of OpenSSL are affected by the **Heartbleed bug**. The Heartbleed bug is not a TLS weakness but simply a programming error (missing check on the size of a `memcpy`). In particular, the bug handled incorrectly the response to a keep-alive message. The TLS protocol mandates that the client sends a keep-alive message with a certain number of bytes, followed by a request of getting back the same number of bytes and the server should answer with the bytes received with the keep-alive message. Namely, the client sends 10 bytes, then asks the server to send back 10 bytes and the server sends back the 10 bytes it received with the first message. It turned out that an attacker could send a message asking back more bytes than those sent in the keep-alive message and the server would send them, anyways. This means that the server would copy the bytes from the buffer containing the bytes received with the keep-alive message and some other bytes, resulting in a leak of the stack. This bug has been fixed but we have to be sure to have an implementation without this bug.
- **TLS before version 1.3 doesn't force integrity checks**. To solve this problem we should try to use MAC or authenticated modes of operation as much as possible. This is done to prevent an attacker from using the server as an oracle by manipulating the padding.

11.1.4 Uses

TLS is used all over the Internet to secure non-secure protocols. As a rule of thumb, when we add an S to a protocol we mean the secure version of that protocol over TLS. For instance, HTTPS is HTTP over TLS and FTPS is FTP over TLS. Secure versions of SMTP, IMAP and POP3 have also been implemented. We usually have two options to secure protocols like SMTP:

- Starting an SMTP session and then requesting to secure the session using TLS.
- Starting a TLS connection and then running SMTP on top of TLS.

The latter option is way better since we can encrypt every SMTP message from the beginning. Moreover, using the former option allows an attacker to issue a command to start a TLS connection. TLS is also used by OpenVPN for point-to-point VPNs.

TLS wrappers

We are able to secure these protocols thanks to the Stunnel utility which provides a TLS transport wrapper for any protocol. Stunnel works like a forwarder in fact it opens a local port, sends unsecured traffic to the local port and then sends it encrypted over the network. Stunnel works on the local port and sends out the traffic it receives on a TLS connection.

11.1.5 TLS 1.3

As we've said, version 1.3 of TLS doesn't solve many security bug but focuses more on refactoring the code to make it less dangerous to use. In particular, the main changes with respect to version 1.2 are:

- The handshake requires only 2 round-trip-times since the client sends the session key and the supported cyphers in the `ClientHello` message.
- Key exchange has become mandatory. This provides perfect forward secrecy.
- The ChaCha20 (tweakable block cipher), Poly1305 (MAC), ED25519 (DSA) protocols have been added.
- It's not possible to transparently downgrade to previous versions of TLS. In particular, the downgrade (which should still be avoided) has been made evident by using the string `DOWNGRD` as the first bytes of the nonce.
- Weak primitives like DES, MD5 or SHA1 have been removed.
- Export-grade cyphers like RSA-512 have been removed.
- Data compression (which may leak information) and suite renegotiation have been removed. Data compression might in fact leak the entropy of the message.

11.2 SSH

Secure SHell (SSH) has been initially implemented for securing access to remote terminals. Since its creation, SSH has become a complete protocol for secure communication like TLS and it's even richer than TLS and has even more features. In fact, SSH is a complete suite, namely it has a wide set of tools for securing communication and application over a non-secure channel against eavesdroppers. For instance, SSH is used for securing FTP, hence obtaining SFTP (which is different from FTPS, which uses TLS).

11.2.1 Architecture

SSH uses a hybrid cryptoscheme, hence:

- an asymmetric cypher for authentication and key exchange or agreement and
- a symmetric cypher for session encryption.

SSH is divided into many sub-protocols, each handling a different part of the whole communication protocol. The main components of SSH are:

- The **SSH transport protocol**. The transport protocol works like TCP but inside an SSH connection. This protocol ensures server-side authentication, confidentiality and integrity for each SSH session.
- The **SSH client authentication protocol**. The client authentication protocol handles the client's authentication. Client authentication is handled separately from server-side authentication and in a custom protocol because client-side authentication might be different from server-side authentication and it might even not involve certificates or public keys.

- The **SSH connection protocol**. The connection protocol handles the connection on the SSH transportation layer. The connection protocol even allows fitting multiple connections (logical connections) in a single SSH transport tunnel. This means that we can authenticate once and use multiple applications in the same session.

Transport protocol

The transport protocol mainly focuses on:

- server-side authentication and
- confidentiality and integrity when transferring data.

Let's focus on each part separately.

Server-side authentication Server authentication guarantees to the client that it is talking to the intended server. The first time a client connects to a remote server, the user is prompted with a fingerprint, which is the server's public key. The user is asked to verify the fingerprint, hence the user has to do the CA's job. When the user confirms the server's identity, the key is added to the file

```
~/.ssh/known_hosts
```

This file contains a list of couples ip-public key of known hosts (one line per ip-key pair), namely of servers to which the user has already connected and which the user trusts (i.e., whose fingerprint has been verified by the user). From this point on the client can connect to that host without verifying the fingerprint. In short, SSH employs a trust-on-first-use mechanism for server authentication. Whenever a client wants to revoke a certificate, it simply has to remove the ip-key couple of the server whose certificate has to be revoked.

This is the default behaviour but it's still possible to defer server-side authentication to the PKI.

Confidentiality and integrity for transport The transport protocol uses protocol data units (PDUs) of 2^{17} bytes, at most (i.e., each PDU is at most 2^{17} bytes long). It's also possible to use LZ for lossless compression. The protocol describes how a tunnel should be initialised. A tunnel can then be used by multiple channels (i.e., connections between the client and an application). Differently from TLS, the SSH transport protocol mandates that both parties select a cypher-suite. In particular:

1. the client sends the supported cypher-suites to the server,
2. the server sends the supported cypher-suites to the client,
3. the client selects a cypher-suite and sends it to the server,
4. the server selects a cypher-suite and sends it to the client,
5. if client and server agree on a cypher-suite (i.e., the two chosen cypher-suites are the same) the tunnel is established, otherwise the connection terminates.

After establishing the cypher-suite, the client and the server agree on a session key using a one-side authenticated Diffie-Hellman key exchange (DH KEX, for short). This means that only the server authenticates itself during the key exchange. Namely, the server signs its part of the Diffie-Hellman key exchange (i.e., $g^s \mod n$), whereas the client does not (i.e., $g^c \mod n$ is not signed). The reason is that the client might not have an asymmetric key pair, hence it might not be able to generate a certificate. The Diffie-Hellman key exchange is used to obtain a session key K and an exchange hash H which is used as the session id.

Communication integrity is enforced using an HMAC computed on each packet.

Client authentication protocol

The client authentication protocol is used to authenticate the client, hence guaranteeing that the server is talking to the intended client. Client-side authentication can be performed in many different ways:

- **Password**.
- **Public key**.
- **Host based**.
- **Keyboard interaction**.
- **None**.

Let us analyse each method separately.

Password Password-based authentication uses a login-password mechanism for authenticating the user (i.e., the user is authenticated using something he or she knows). The username and the password are encrypted (which is possible because client authentication is performed after the key exchange). Note that man-in-the-middle attacks are prevented, at least from the user point of view, by the fact that the server has signed its part of the key exchange, hence the client is sure to be talking to the intended server. This means that, from the user's perspective, he or she must have received one part of the session key from the legitimate server, hence only the server can decrypt the password sent in this phase.

Public key Public key-based authentication works as server authentication. The first time the user wants to use public key authentication,

1. the client generates its key pair (e.g., using the command `ssh-keygen`),
2. the client stores its private key locally in a file without extension and its public key locally in a file with extension `.pub`,
3. the user deposits its public key on the server (after having accessed in through other authentication methods) in the file `~/.ssh/authorized_keys` which stores a list of public keys-host name pairs (separated by newlines).

Every time the user has to authenticate (i.e., when he or she connects to the server), challenge-based authentication is performed, namely:

1. the server generates a random number and asks the client to sign it (with the private key),

2. the user signs the random number and sends the signature to the server,
3. the server checks the signature using the public key stored in the file `~/.ssh/authorized_keys`. If the signature is valid, the client is authenticated.

If the signature is valid then the client must have signed with the private key connected to the public key stored on the server, hence the client must be the legitimate one.

The main advantages of this authentication method are that the user doesn't have to prompt anything (if not the first time) and no valuable information is sent over the communication channel since authentication is based only on the signature of a random nonce.

Host based Host-based authentication works like public key authentication but the user is authenticated by signing with the host-wide private key of the client and the check is done with the host public key. This authentication method is based on the assumption that if a user can access the local host, independently from the machine, then he or she is authenticated.

Keyboard interaction Keyboard interaction authentication is based on external smart cards or tokens that generate and store the private key. This means that authentication is delegated to the smart card and to an external protocol like GSSAPI or PAM. For instance, Pluggable Authentication Modules (PAM) allow SSH implementations to use smart cards, secure tokens or other delegates to perform authentication. Secure tokens require the server to have a corresponding PAM module requesting a signature from them.

None Even if we should always authenticate the user, SSH also supports no user authentication. This option is however strongly discouraged.

Connection protocol

The connection protocol is based on channels which are the equivalent of TLS sessions. Channels are used for user-interactive sessions (e.g., terminals or X11 services), machine-to-machine data transfer (secure copy `scp`), port forwarding (but remember that we are operating on top of ISO/OSI level 4) or SOCKS proxying. Each channel is used for a single user-interactive session or a single machine-to-machine data transfer. Multiple channels can be opened in a single SSH session (which is not a TLS session). The connection protocol defines how to open multiple channels in an SSH session.

11.2.2 Key management

SSH implementations support both GPG and X.509 standard certificates for key storage seamlessly. The keys are usually stored in the `.ssh` directory in the home directory. This directory usually contains:

- The `known_host` file where the client stores a new-line separated list of authorised server ip-public key pairs.
- The `autorized_keys` file where the server stores a new-line separated list of public key-client name pairs.

11.3 TOR

TOR is a secure communication protocol which ensures anonymity only. This means that it assumes that the protocols it runs on top of already ensure confidentiality and authentication. Moreover, TOR doesn't aim at fully anonymising a user (since some information might be extracted from meta-data) but simply at anonymising its source and destination IP.

The first attempt at obtaining such a result relied on a single point of failure and on a trustworthy proxy (that provides anonymity). TOR instead doesn't rely on any of these assumptions. Initially, TOR also provided total anonymity, namely even an attacker monitoring the whole network wouldn't be able to track the user. The problem with this approach is that it required adding delays to hide a request, hence the latency was very high. TOR trades-off anonymity for latency, namely it has less latency at the cost of offering less anonymity. To achieve lower latency and anonymity, TOR:

- Uses a small number of hops (at least 2 but usually not more than 5).
- Builds a tunnel through the network from the source to the destination.

Before analysing the protocol in detail, let us summarise what TOR guarantees:

- TOR guarantees **integrity** and **confidentiality** as a side-effect, in fact, it's run on top of TLS which guarantees integrity and confidentiality.
- TOR guarantees **anonymity** by design.
- TOR guarantees **forward secrecy** by design and not only as a side effect of running on top of TLS.

Note that TOR works just on top of the transport layer, hence it can't anonymise what's on top of it (e.g., what's on the application layer). This means that one can de-anonymise a person through an injected Javascript payload that collects information about the user. This doesn't mean that the protocol is broken but just that it's simply solving the problem of hiding the source and destination address of a request to the network. Every communication in TOR is run on top of TLS, hence every TOR message is contained as the payload of a TLS connection.

11.3.1 Actors

Let us start by describing TOR's network architecture. The TOR network is made of three classes of nodes:

- **Relays**. Relays are nodes that receive a packet and send it to another relay in the TOR network over a TLS connection which is established among relays.
- **Clients**. Clients are nodes that want to do a request. A client connects to a relay called **guard** with a TLS connection and sends packages to and receives packages from its guard. In other words, the guard is the only access point of a client to the TOR network.
- **Exit points** (which are a particular type of relay). Exit points are relays which agree on being the node that appears as the source of a request to a server, namely as the source of the traffic generated by the network. Being the exit node isn't always an advantage since it performs requests for people that want to be anonymous, hence visit websites that might be illegal.

For instance, one client that wants to get `example.com` sends a request to its guard. The request for `example.com` is forwarded from relay to relay (usually a couple of hops) and finally, it reaches the exit point. The exit points send the request to `example.com` and forwards-back the response through the same path used to reach the exit point.

11.3.2 Circuits

Now that we know the network architecture, we have to understand how messages are forwarded through it. TOR is based on the concept of a circuit. A circuit is a transport path for data across relays in the TOR network, similar to a physical wire. In other words, a circuit is a virtual private connection across the relays. A circuit can be built or torn down by a guard. The circuit is built on top of TLS (without client-side authentication, otherwise the protocol would reveal the identity of the client), hence sniffing isn't effective.

Circuits are built telescopically. This means that the client initially opens a TOR connection to its guard. The TOR session just created is used to create a TOR connection to a relay passing through the guard. This connection is then used to create yet another connection to another relay passing through the guard and the first relay. This process goes on until we reach the exit point. Namely, each TOR connection is wrapped in other TOR requests. In other words, we have built a circuit by adding one hop at a time. Note that the circuit should contain at least three relays (usually one guard, one normal relay and one exit point).

Cells

The basic transport units that travel through a TOR network are called cells. There exist two types of cells:

- **Control cells**. Control cells are used to set up and tear down a circuit.
- **Relay cells**. Relay cells are used to do everything else (usually send data).

Every cell has a two bytes id which identifies the circuit on which the cell should be travelling. The id acts similarly to ports because relays know where to forward cells thanks to the circuit id.

Circuit setup

The circuit is set up by the client, hence the client chooses what relays should be part of the circuit and what should be the exit point. We will now describe the procedure for building a basic circuit with 3 relays (which is the minimum required by the protocol). This procedure can then be easily extended to an arbitrary number of relays.

The protocol for building a circuit is based on the fact that the attacker is a malicious relay which can read TLS (which means that the attacker is stronger than usual).

The first thing we have to do is to build a connection from the guard to the first relay. This connection is a TLS connection but we have to add additional security mechanisms for protection against malicious relays. The connection is created as follows:

1. The guard sends a `Create` control cell which contains the circuit id c_1 and the guard's half of the Diffie-Hellman key g^x, encrypted using RSA. This is done to anonymise the guard's half of the Diffie-Hellman key.
2. The relay picks y, decrypts g^x and computes $k_1 = (g^x)^y$.

3. The relay sends a `Created` control cell containing:
    - the circuit id,
    - g^y and,
    - a hash of the session key k_1 concatenated to the string "handshake" so that the guard can test the correctness of the procedure.

At the end of this interaction, the guard and the first relay have a shared secret g^{xy}. This means that the guard can securely communicate with the first relay even against relays which are not the first relay.

Now we need to extend the circuit, hence to connect the relay with the exit point. This is done by wrapping a `Extend` cell, similar to the `Create` cell, for extending the circuit to the exit node into another cell, called `Relay` cell. The `Relay` cell is sent to the first relay which opens it and sends a `Create` cell to the exit point, hence the exit points receive the `Create` message from the relay, even if the actual source is the guard. This mechanism has two advantages:

- Anyone sniffing the communication between the guard and the relay will only see a `Relay` message, without knowing that it contains an `Extend` cell.
- The exit point only knows that the relay wants to create a connection, but doesn't know if the relay is a guard or a middle relay.

In practice, expanding the circuit works as follows:

1. The guard picks a secret x_2 and computes its half of the Diffie-Hellman key, namely g^{x_2}.
2. The guard builds an `Extend` control cell containing the identifier of the exit point (in general of the cell to which we want to extend the circuit) and g^{x_2} encrypted with the RSA public key of the exit point.
3. The guard wraps the `Extend` cell into a `Relay` control cell containing the identifier of the circuit c_1 and the `Extend` cell encrypted using AES (with the key shared between guard and relay).
4. The guard sends the `Relay` cell to the relay.
5. The relay, upon receiving the `Relay` cell, decrypts it and sends a `Create` cell to the exit node (as requested by the `Extend` cell). The `Create` cell contains the identifier of the circuit c_2 and the content of the `Extend` cell, namely g^{x_2} encrypted with the exit node's public key.
6. The exit node decrypts the content of the `Extend` cell using its private key (obtaining g^{x_2}), picks y_2 and computes $k_2 = (g^{x_2})^{y_2}$.
7. The exit node replies to the relay with a `Created` cell containing
    - the identifier of the channel c_2,
    - the exit's half of the Diffie-Hellman key g^{y_2}, and
    - the hash of the session key k_2 and the string "handshake" so that the guard can test the correctness of the procedure.
8. The relay builds a `Extended` control cell containing
    - the exit's half of the Diffie-Hellman key g^{y_2} (contained in the `Created` cell), and

- the hash of the session key k_2 and the string "handshake" so that the guard can test the correctness of the procedure.

9. The relay wraps the `Extended` cell in a `Relay` cell containing
   - the identifier of the circuit c_1, and
   - the `Extended` cell, encrypted with AES using the shared key k_1.
10. The guard unwraps the `Relay` cell and computes the session key with the exit point as $(g^{y_2})^{x_2}$.

Figure 11.3 shows the protocol we have just described with the messages exchanged by the relays.

Data relay

After having created a circuit, the client can issue a request to the network through its guard. When sending requests over the TOR network we want to ensure that:

- Only the exit node knows the destination of the request.
- The exit node doesn't know the source of the request.
- The middle relays only know that a cell should be relayed to another relay, without being able to read the request.

Two hops (i.e., three relays) are enough to ensure these properties. The protocol actually uses three hops to ensure these properties even when two relays are malicious.

The protocol for issuing requests works similarly to the one used for creating the circuit. The idea is to wrap a request into multiple `Relay` cells which are peeled away by the relays and forwarded until the request reaches the exit point. More precisely,

1. The client sends a request to the guard.
2. The guard builds a cell containing the request.
3. The guard wraps the cell in a `Relay` cell for the exit point containing
   - the identifier of the circuit c_2, and
   - the cell containing the request encrypted using AES with the key k_2 shared with the exit point.
4. The guard wraps the `Relay` cell in another `Relay` cell for the relay containing
   - the identifier of the circuit c_1, and
   - the `Relay` cell encrypted using AES with the key k_1 shared with the relay.
5. The guard sends the `Relay` cell to the relay.
6. The relay, upon receiving the `Relay` cell, decrypts its content using the shared key k_1.
7. The relay, since the content of the received message is another `Relay` message on circuit c_2, sends the `Relay` cell to the exit node through circuit c_2.
8. The exit point, upon receiving the `Relay` cell, decrypts its content using the shared key k_2.

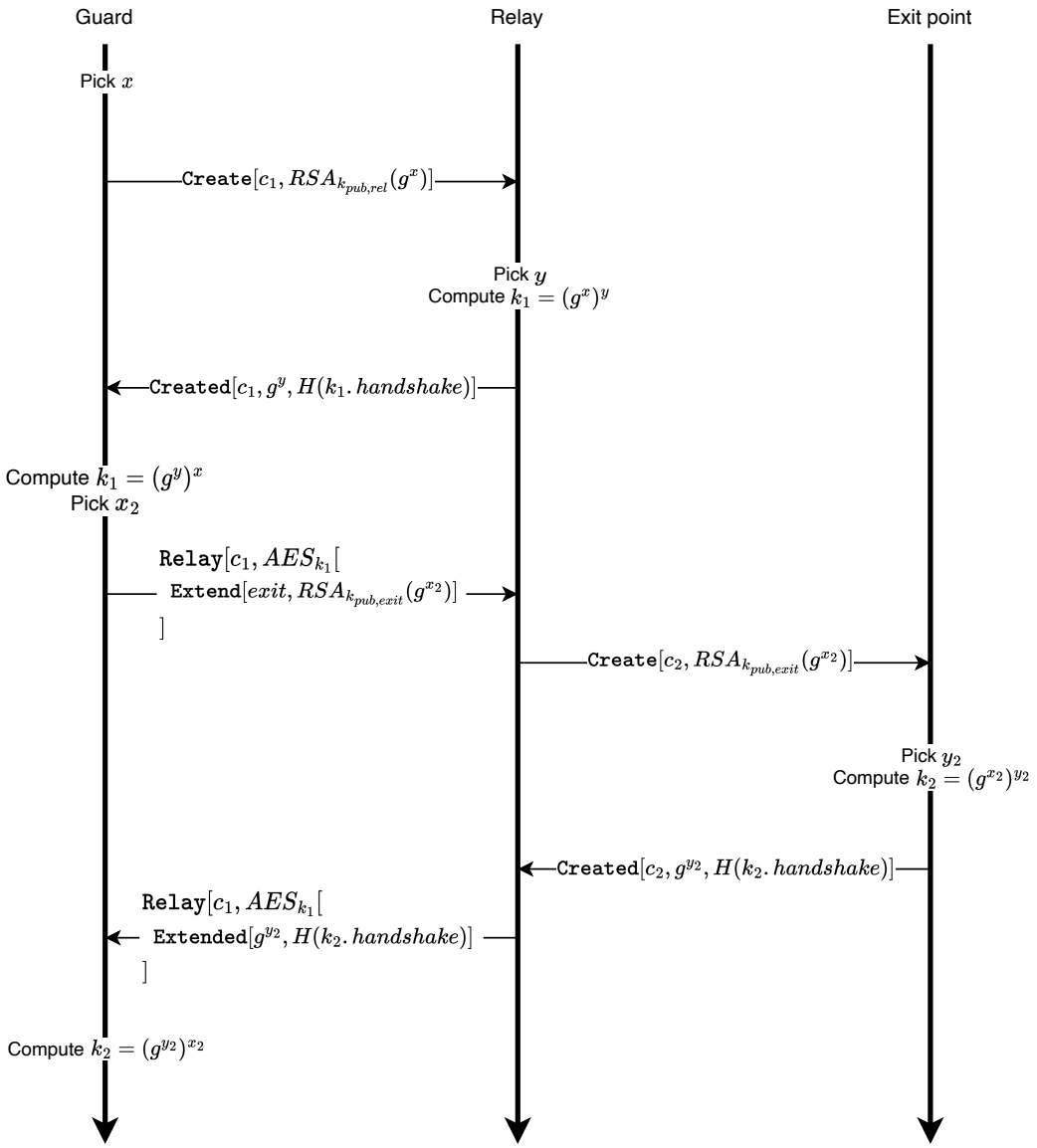


Figure 11.3: The messages exchanged to build a circuit.

9. The exit point, because the `Relay` cell contains a `Data` cell, executes the request in the `Data` cell.
10. The exit point, upon receiving the response, encrypts the response using AES with the key k_2 and wraps it in a `Relay` message containing
    - the identifier of the circuit c_2, and
    - the encrypted response.
11. The exit node sends the `Relay` cell to the relay.
12. The relay wraps the received `Relay` cell in another `Relay` cell containing
    - the identifier of the circuit c_1, and
    - the received `Relay` cell encrypted using AES with the key k_1.
13. The relay sends the `Relay` cell to the guard.
14. The guard decrypts both `Relay` cells and obtains the server's response.
15. The guard sends the server response to the client.

This protocol can be used for every type of communication. For instance, we can use it for

- Opening a TCP connection. The content is wrapped by the guard in a `Begin` cell and the response is wrapped by the exit node in a `Connected` cell.
- Sending GET (or whatever other method) requests. The request and the response are wrapped by the guard in a `Data` cell.

Figure 11.4 shows the protocol we have just described with the messages exchanged by the relays.

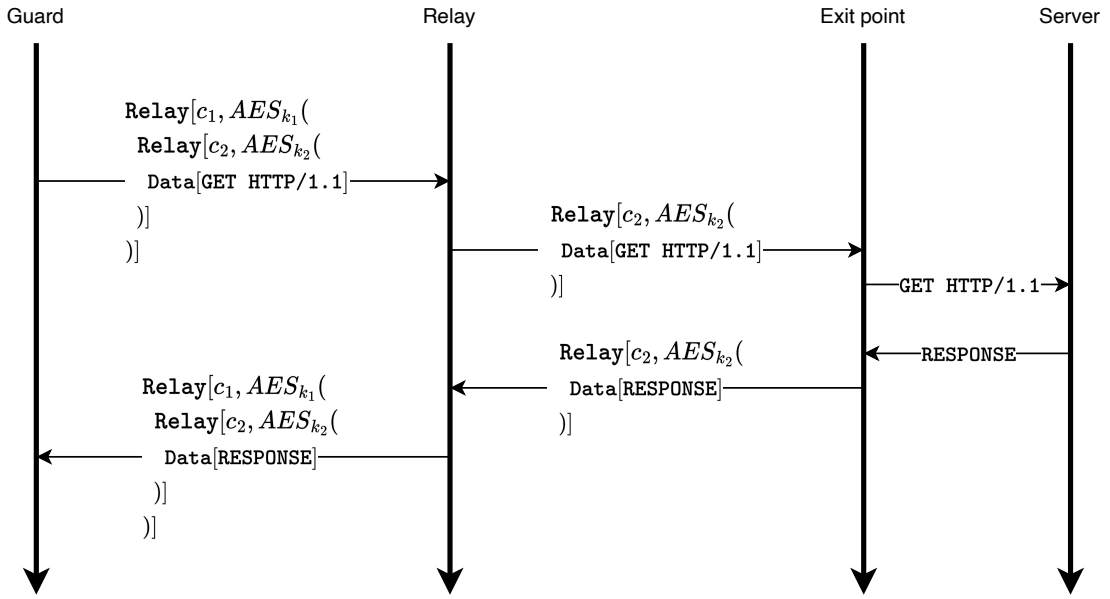


Figure 11.4: The messages exchanged to relay a GET request.

Chapter 12

Password storage and disk encryption

12.1 Password storage

Passwords are the most widespread authentication mechanism on the Internet. A server usually requires the user to provide a password and checks on its local database if the password corresponds to the one provided by the user when registering the first time. Since the users' passwords are stored on the server, we must describe how they can be securely stored. In particular, we want to be sure that, even in the case of a data breach (i.e., the company's database is leaked and the passwords are published on the Internet), no one can efficiently obtain the password of the users.

12.1.1 Hashed passwords

It's immediately evident that we can't store passwords in clear on the server since after a data breach everyone would be able to immediately associate a user with a password (since the database contains at least username-password tuples). The solution to this problem is using hashes because they are preimage resistant, hence given a hash, we can't compute the password corresponding to that hash if not by bruteforcing it. This means that a server must store tuples containing the username and the hashed password of a user. This is for sure a significant improvement over storing passwords as plaintext, however, it's still flawed. In particular, hashes are thought to be fast to compute since they might have to be performed even on large documents (we can even achieve speeds of about 1 MB/s). If the hash is fast to compute, an attacker might try and bruteforce the leaked dataset. In particular, an attacker can

1. take every possible password of a specific length,
2. compute the hash of every password chosen at step 1, and
3. check if any hash computed at step 2 is present in the dataset.

Since hashes can be computed swiftly, an attacker can recover a bunch of passwords with enough time. To slow this process down and make the attacker's life harder, we can hash a password multiple times (usually 100 times is enough). This means that, upon the registration of a new user, we store the username and $H(\ldots_{N\ times} H(H(H(pwd))))$. This process:

- Doesn't impact the user since computing one hash requires a little time, hence doing it 100 times doesn't change much. Hashing a password usually takes one microsecond, hence doing it 100 times requires less than 1 millisecond. A user won't notice the difference between microseconds and milliseconds, hence he or she doesn't notice that the password is hashed many times.
- Impacts the attacker since the process of bruteforcing passwords is slowed down by a factor of 100.

12.1.2 Rainbow tables

Even if passwords are hashed multiple times attackers can still recover some passwords from a leaked database. Attackers trade off memory for computation time to recover passwords from a database computing hashed passwords. Computing a hash is time-consuming however if we consider passwords of length at most N, we can pre-compute the hashes for each password of length at most N and then use directly the pre-computed hashed passwords to recover the passwords from the database. More precisely an attacker can

1. compute the hash for a set of passwords,
2. store the computed hash in a dictionary which associates a hash to a password,
3. access the dictionary to get the password which has a certain hash.

This attack clearly trades off memory for computation time in fact we can obtain the hash of a password in constant time (i.e., the time required to search a key in a hash table) but we have to store a huge amount of data. Even if this technique works in principle, it doesn't in practice. If we were to store the hashes of all passwords with exactly 7 alphanumeric characters, we would require around 58 terabytes of storage. To solve this problem we can trade off some memory for some computation time. Namely, we store less data but obtaining the hash of a password requires more time. This trade-off is achieved using **rainbow tables**. Note that rainbow tables apply to whatever unstructured function, hence we will use it on hashes. For instance, we can apply rainbow tables to symmetric cyphers to recover the secret key. In this case, the hash is replaced with the encryption function. The idea behind rainbow tables is that we don't care about the order in which passwords are computed when building the data structure that contains the password-hash tuples. This means that we can

1. initialise a counter ctr to 0,
2. take an input i_0,
3. compute its hash $d_0 = h(i_0)$,
4. store (i_0, d_0),
5. use a reduction function R to transform the digest d_0 into a valid input (i.e., make the digest a printable input, namely a valid password),
6. consider $i_1 = R(d_0)$, increment the counter ctr and repeat from step 2 until the counter is bigger than a certain value n.

This procedure creates a chain of hashes that starts with the hash d_0 and ends with the hash d_n. This procedure can be repeated as many times as we want (i.e., until we have generated every password or until we think we have generated a good number of hashes) but we don't have to store every hash (otherwise we wouldn't have a trade-off). Let us call d_0^a and d_n^a the first and last hashes of the a-th chain. We store only the first and last hash of each chain, namely d_0^a and d_n^a. We can store only these two hashes because we know how to go from d_0^a to d_n^a, hence we can compute the intermediate hashes of the a-th chain on the go. As a result, the length of the chain n is the parameter we can tune to trade off memory for computation time. In particular, the longer the chain, the less data we have to store but the more computational time we need to recover a password. A good choice, if we have enough memory is to take $n = \sqrt{len(pwd)}$. Figure 12.1 shows an example of a rainbow table.

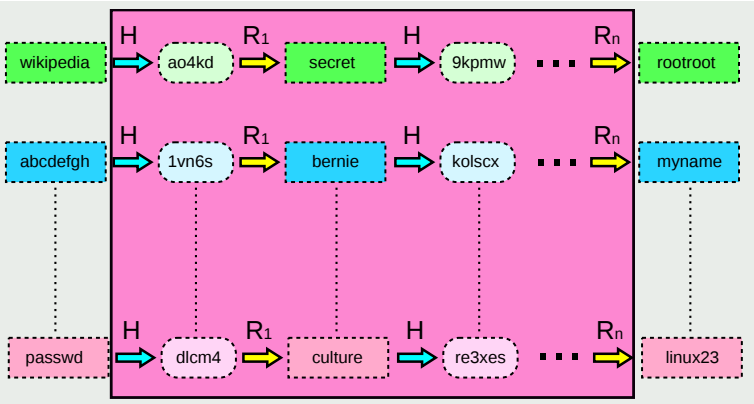


Figure 12.1: A rainbow table. The purple part is not saved.

Using rainbow tables

Now that we know what a rainbow table is and how it's computed, we need to understand how to use it. Say we want to find the password related to a hash d. What we have to do is:

1. Consider $t = d$.
2. Reduce the hash t using the same reduction function used for computing the rainbow table. We compute
$$i = R(d)$$
3. Look for i among hashes $\{d_n^0, \ldots, d_n^r\}$.
4. If we don't find i among the hashes $\{d_n^0, \ldots, d_n^r\}$ we compute go back to step 1 using $t = i$.
5. If we find i among the hashes $\{d_n^0, \ldots, d_n^r\}$, say $i = d_n^a$, then we take d_0^a and we start computing the chain until we obtain the hash d. When we find d we can take the input that generated d, namely if $d = h(i_l)$, we take i_l. The input i_l is the password with hash d.

This algorithm is based on the fact that a hash d is somewhere in a chain, hence we have to compute the chain starting from d until we reach the end of the chain. When we reach the end of the chain we know to which chain d belongs. This means that we can take the beginning of that chain and create the chain until we reach the input that generated d.

Parameters tuning

When building a rainbow table, we have to choose what reduction function R to use. In principle, it doesn't exist an ideal reduction function if the output of a hash function is really random. This means that any fast encoding function that transforms a hash into a valid password is a good choice.

Another thing we have to tune is how many chains to store in the rainbow table. This choice is a trade-off between password coverage and storage. If stop when most of the passwords are present in the chains we are more likely to find the hash of a password, however, we have to store a lot of data. On the other hand, we might miss some passwords but we don't need a lot of storage.

12.1.3 Salted passwords

Rainbow tables allow attackers to efficiently recover passwords from a leaked database. Luckily, we take some countermeasures to mitigate the effectiveness of rainbow tables. To understand how, we have to understand why rainbow tables work. A rainbow table is based on the fact that given a hash function H and a password p, the hash of p is the same on every server that uses H as a hash function. To make rainbow tables ineffective we have to find a way to make it so that the hash of a password p is different on each server and is different from the hash computed in the rainbow table. To achieve this goal we can append a value called salt to the password, hence we store

$$h = H(password||salt)$$

where $||$ represents the concatenation operation. A different salt is computed for each user and the salt is stored in plaintext and can be read by anyone. Note that the salt isn't stored encrypted because it's only used to make it impossible to use pre-computed hashes, but it doesn't provide any extra security, namely, the salt doesn't prevent the password to be cracked if it's easy (i.e., if the password is `1234567`, the salt doesn't make it more difficult to directly crack it). The hash is used because an attacker would have to compute a hash table for every possible salt to use pre-computed hashes. For this reason, the salt is usually 1024 bits long.

Attacking a salted database

When an attacker wants to attack a database containing salted passwords, he or she has two strategies:

- **Bruteforce a password using a specific salt**. This means that, if we want to reverse a specific hash h_i, we take the salt s_i related to h_i and try to hash every possible password, mixed with the salt (i.e., compute $H(pwd_t||s_i)$, for all pwd_t), until we get the hash we were trying to break.
- **Use a rainbow table for a specific salt format**. An attacker can look for a rainbow table build using all the possible combinations of passwords and salts. This means that the rainbow table is bigger than a rainbow table built without salt. Given a hash $H(pwd||salt)$ we can do a lookup on the salted rainbow table and obtain both the password and the salt (which must be compared with the salt stored in the database to be sure that the whole process worked correctly).

With both strategies, we break a password with one lookup but:

- In the first case, a lookup is an exhaustive key search on the password space, fixed a salt.
- In the second case, a lookup is a lookup in the salted rainbow table.

POSIX salted hash

The POSIX `crypt()` function is one of the most diffused salted hash primitives and is widely used to store passwords in a variety of Unices. The first edition employed a modified Enigma simulator as hashing function and was not salted (for this reason it's strongly deprecated). The second edition replaced it with 35 runs of the DES algorithm, employing the salt as the key and a combination of the password bytes as the plaintext. This version is deprecated, too. The third instance employed 1000 runs of MD5 and a salt mixing scheme which involved alternating salt and password bytes as the input of the MD5 primitive, without re-initialising the hash state. This version is deprecated as bruteforce speeds reach 600k keys per second on modern GPUs. A modern revision employs 3000 runs of SHA-256/512.

12.1.4 Memory hard algorithms

Hashing algorithms can be slowed down by repeatedly applying the hashing function, however, an attacker might design custom hardware to speed up the computation of hashes and make bruteforce achievable. To counteract this threat, we can design hashing algorithms which are purposely slow and memory inefficient. Such algorithms are called **memory hard algorithms**. Memory hard algorithms link the time complexity with the area required to implement them in hardware, hence they bind the time complexity with the spatial complexity. Formally, a memory hard algorithm on a RAM machine is defined as follows.

Definition 12.1 (Memory hard algorithm)**.** *A memory hard algorithm A on a RAM, with $T_A(n)$ time and $S_A(n)$ space requirements, is characterised by*

$$S_A(n) \in \Theta(T_A(n)^{1-\epsilon})$$

Long story short, a memory hard algorithm burns as much memory as time.

12.1.5 Sequential memory hard algorithms

Even if hash algorithms can be slowed down, an attacker can parallelise them. This means that we have to find an algorithm which can't be parallelised and it's slow even when using a parallel machine. Such algorithms are called sequential memory hard algorithms. Formally,

Definition 12.2 (Sequential memory hard algorithm)**.** *A sequential memory hard function is an algorithm A, that can be computed in $T_A^{RAM}(n)$ on a RAM, which can be computed on a PRAM in $T_A^{PRAM}(n)$ time and $S_A^{PRAM}(n)$ space no faster than*

$$\mathcal{O}\big(T_A^{RAM}(n)^2\big) = S_A^{PRAM}(n) \cdot T_A^{PRAM}(n)$$

This means that parallelising a sequential memory hard algorithm at most gives a square root improvement.

ROMix

ROMix is an example of sequential memory hard hashing function. Algorithm 10 shows the ROMix algorithm. Note that ROMix isn't a specific algorithm but more of a framework.

Algorithm 10 The ROMix hashing algorithm.

Input: a password p
Input: N
$t \leftarrow p$
for $i = \{0, \ldots, N-1\}$ **do**
  $v[i] \leftarrow t$
  $t \leftarrow H(t)$
end for
for $i = \{0, \ldots, N-1\}$ **do**
  $j \leftarrow t \mod N$
  $t \leftarrow H(t \oplus v[j])$
end for
return t

Argon2

Argon2 is another sequential memory hard hashing algorithm. Argon2 implements the idea of ROMix and also takes into account timing side-channel attack resistance. A public implementation of Argon2 is available at https://www.password-hashing.net.

12.2 Key derivation functions

A key derivation function is a function which generates a symmetric key from a password of arbitrary length. The current state-of-the-art for key derivation is the Password-Based Key Derivation Function 2 (PBKDF2) algorithm. The PBKDF2 derives a key k, of length l_k starting from an arbitrary length password p and a salt s, with a given work factor c. PBKDF2 uses an HMAC to generate a pseudo-random key. Note that using an HMAC makes sense because it depends on a key (hence we can use the password as a key so that the HMAC output depends on the password). A sequence number is used as input of the HMAC to generate the key.

More precisely, the core of PBKDF2 is a 2-parameters pseudo-random function H, for example, `HMAC(text, key)`, with output length l_h. The derived key is built as $i = \lceil \frac{l_k}{l_h} \rceil$ tiles t_i obtained as

$$t_i = \bigoplus_{j=0}^{c} s_j$$

where

$$s_0 = s||i$$

and

$$s_j = H(s_{j-1}, p) \quad \forall j \; 1 \leq j \leq c$$

12.2.1 ROMix as key derivation function

The ROMix function can be used to slow down a preexisting key derivation function and obtain a sequential memory hard key derivation function. In particular, starting from a password p and a salt s we:

1. Apply PBKDF2 to compute p as

$$p \leftarrow \text{PBKDF2}(password, salt, 1, l_h)$$

2. Use p as input to the ROMix function to obtain o as

$$o \leftarrow \text{ROMix}(p, N)$$

3. Apply PBKDF2 again, using o as salt,

$$return \leftarrow \text{PBKDF2}(password, o, 1, l_h)$$

The work factor N of this construction tunes both the computation and the memory required. This approach constitutes an alternative, which complies with the requirement of having PBKDF2 being the key derivation function in a procedure.

12.3 Disk encryption

Up until now, we have considered message encryption, namely how to ensure confidentiality, integrity and authentication of data sent from a source to a destination. It's now time to analyse how to provide confidentiality and (optionally) integrity of data at rest, namely data stored on a storage device (i.e., in mass memory). Data on a storage device can be encrypted using three different levels of granularity:

- **Single file level**. A single file is encrypted using a symmetric cypher. The key used for encryption can be obtained by applying a key derivation function on the user's password.
- **Filesystem overlay level**. The contents of all the files in the filesystem are encrypted, but no metadata is encrypted. In some cases, a new filesystem (i.e., an overlay) interposes itself between the filesystem and the user and transparently encrypts files in transit such that the user doesn't notice that the encryption is happening at the filesystem level. An example is Ubuntu's `ecryptfs` which provides a FUSE-based filesystem encryption overlay. A filesystem overlay doesn't alter the filesystem structure. The main advantages of a filesystem overlay are that it provides confidentiality but it's also flexible.
- **Block level**. The entire disk, or the entire logical volume, is encrypted, block by block. Note that the term block refers to a page of the disk, not a cypher block. This operation happens transparently, hence the filesystem doesn't notice anything. Some examples of block-level encryption programs are Bitlocker for Windows, `dm-crypt` for Linux, FileVault for MacOS, and Ciphershed (a Truecrypt fork) which is cross-platform.

When encrypting data at rest we want to provide:

- **Confidentiality**. Encrypted data should not be distinguishable from random data, even if vast amounts of ciphertext (even more than 2^{60} bytes) are available.
- **Integrity**. The encryption scheme should provide a way to spot alterations in the stored data.

We want to always provide confidentiality and optionally (but it's better if we can) integrity. To achieve this goal we use a symmetric encryption scheme employing a password-derived key. This means that we need

- a bruteforce-resilient password-based key derivation function, and
- a mode of operation tailored for efficient, tamper-resilient mass-data encryption.

AES-CTR (AES in counter mode) is suited for a large number of bytes.

12.3.1 XTS mode of operation

The state-of-the-art for encrypting data at rest is defined in the IEEE P1619 standard. This standard uses the XTS mode of operation, which is a combination of

- the **Xor-Encrypt-Xor** (XEX) mode of operation to have an efficient encryption and decryption scheme, and
- **CipherText Stealing** (CTS) to avoid padding at the end of the plaintext, as it may not be feasible (e.g., at the end of a disk).

Let us now analyse both technologies.

Xor-Encrypt-Xor

The Xor-Encrypt-Xor mode of operation improves the ECB mode of operation (which must not be used for any reason) performing plaintext and ciphertext whitening and using two symmetric encryption keys.

The two keys key_1 and key_2 are obtained by splitting the XEX key in half. This means that a XEX key must be twice as long as the key used for a single symmetric block cypher. For instance, if we use AES-128 as a symmetric block cypher we must use a 256 bits XEX key. The keys key_1 and key_2 are used for the same block cypher (e.g., AES-128) which is used for different purposes:

- The key key_2 is used to generate a pad for whitening the plaintext and ciphertext blocks. The whitening is performed by xoring the plaintext with the pad and the output of the cypher with the pad.
- The key key_1 is used to encrypt a whitened plaintext block.

The Xor-Encrypt-Xor mode of operation works as follows:

1. The index i of the physical block to encrypt is encrypted using key_2 to obtain pad_1.
2. The pad pad_1 is xored with the plaintext block to obtain the whitened plaintext block.
3. The whitened plaintext block is encrypted with key_1.
4. The output of the symmetric cypher is whitened using the same pad pad_1 used for the plaintext to generate the whitened ciphertext block.
5. The next pad p_i ($i \geq 2$) is obtained considering the bits of pad_{i-1} as coefficients of a polynomial over $\mathbb{Z}_{2^{128}}$ modulo $x^{128} + x^7 + x^2 + x + 1$ and multiplying them by x. This operation can be implemented as a left shift of pad_{i-1} by 1 bit and reduced as needed.
6. The pad pad_i ($i \geq 2$) is xored with the i-th plaintext block to obtain the whitened plaintext block.
7. The whitened plaintext block is encrypted with key_1.

8. The output of the symmetric cypher is whitened using the same pad pad_i used for the plaintext to generate the whitened ciphertext block.

9. Go to step 5.

Figure 12.2 shows the scheme of the Xor-Encrypt-Xor mode of operation.

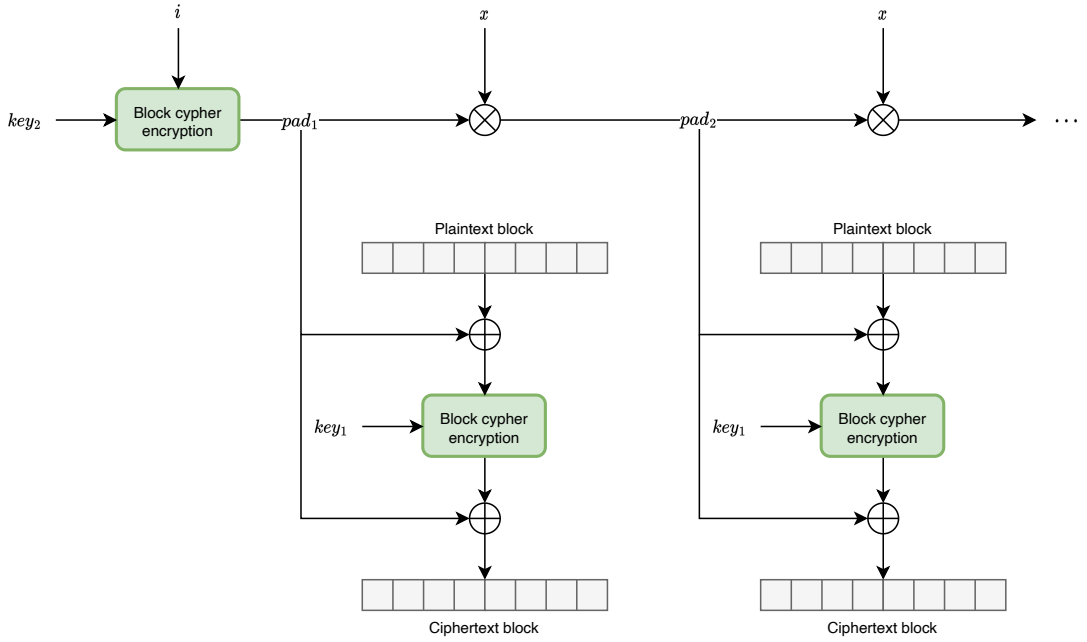


Figure 12.2: The Xor-Encrypt-Xor mode of operation.

Ciphertext stealing

Symmetric block cyphers (where a block is not a disk page but an actual cypher block) require the size of the input to be a multiple of the size of the block. Padding the input means obtaining a padded output. However, it's not possible to deal with a padded output because we might not have enough space to save the padded output. Say for instance we have encryption blocks of 1000 bytes and we want to encrypt a storage of 99500 bytes. If we pad the memory to a multiple of 1000 bytes we would have to store 100000 bytes we don't have enough space to store them. Ciphertext stealing allows encrypting data whose size is not a multiple of the cypher's size. Let us consider a block cypher of size B (i.e., the input size is B). Ciphertext stealing works as follows:

1. Encrypt the penultimate plaintext block p_{n-1}, obtaining c_{n-1}.

2. Pad the final plaintext block p_n with the last $B - m$ bytes of c_{n-1}, where m is the size of a the last block p_n. Let us call head the bits from 0 to $m - 1$ and tail the bits from m to B. Using this notation we can write the last plaintext block as $p_n||tail$.

3. Encrypt the padded final block and store it as the penultimate ciphertext block.

4. Store the first m bytes of c_{n-1} as the final, reduced-size ciphertext block.

Figure 12.3 shows the encryption process of the last two blocks.

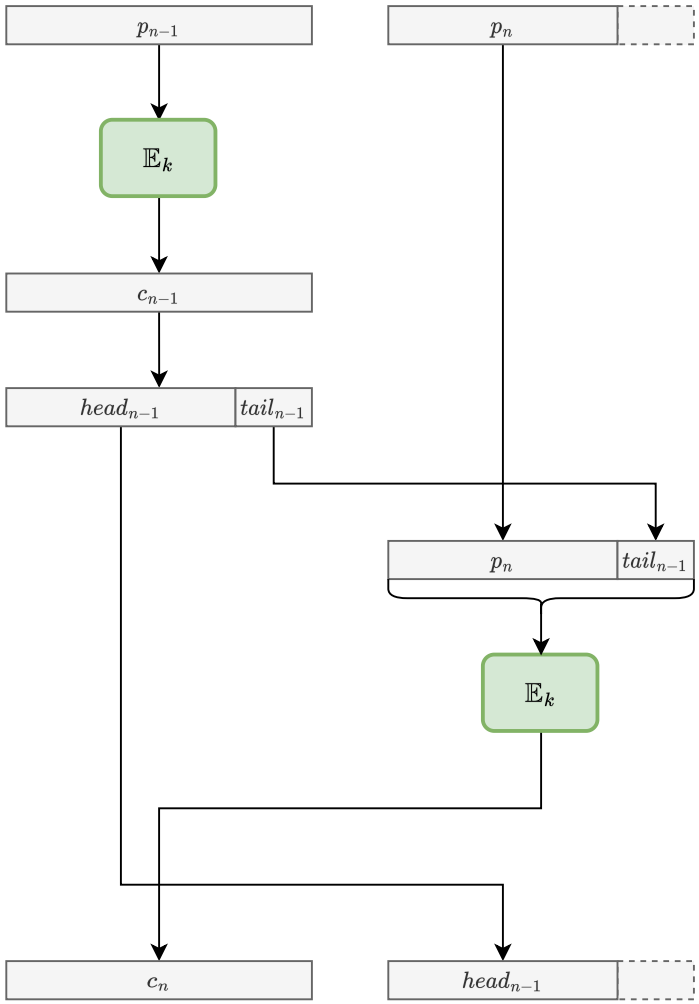


Figure 12.3: Ciphertext stealing for encryption.

Decryption when using ciphertext stealing works as follows:

1. Decrypt the penultimate block.
2. Compute how long the tail should be. This can be done by checking the length of the last block. If the length of this block is m, the tail of the last block should be $B - m$ bits long.
3. Take the last $B - m$ blocks of the decrypted penultimate block and append them to the last ciphertext.
4. Decrypt the last block.

Figure 12.4 shows the decryption process of the last two blocks.

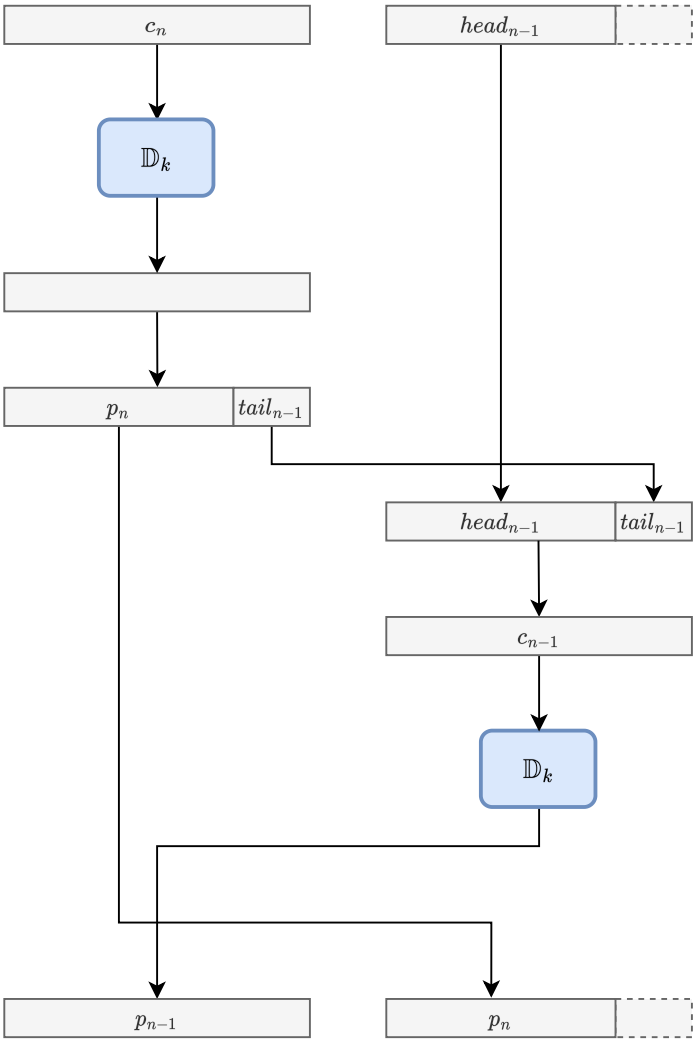


Figure 12.4: Ciphertext stealing for decryption.

12.3.2 Software for encrypting storage devices

Microsoft Bitlocker and Apple Filevault

Microsoft Bitlocker and Apple Filevault are the two alternatives offered by Microsoft and Apple to encrypt the disks on Windows and MacOs, respectively. Both solutions offer transparent, OS-integrated whole-volume encryption which allows booting from an encrypted volume. Both solutions integrate a key escrow mechanism if the password is forgotten. Microsoft also stores a copy of the password on the Cloud which, depending on the level of trust we have in Microsoft, might or might not be a desirable feature. Bitlocker and FileVault support AES-XTS as block cypher and mode of operation starting from Windows 8 and FileVault 2 onwards.

Encrypted volumes can be mounted from Linux, but cannot be used as a root filesystem.

LUKS and DMcrypt

LUKS allows encrypting whole physical disks or logical volumes in Linux and it's integrated within the system volume manager. LUKS uses:

- either PBKDF2 with HMAC-SHA-2-256 or Argon2 as a password-based key derivation function, and
- one among AES, Serpent e Twofish as a block cypher with the XTS mode of operation.

LUKS also supports AEGIS which is an authenticated mode of operation, providing active data integrity Can be read on Windows platforms (https://github.com/t-d-k/LibreCrypt).

A disk encrypted with LUKS can be read on Windows platforms using LibreCrypt (`https://github.com/t-d-k/LibreCrypt`).

Truecrypt

Truecrypt allows encrypting physical volumes and virtual volumes residing in a single file with AES-XTS. Truecrypt is still available for Windows, Linux and MacOS, despite being discontinued The codebase and cryptographic protocols of Truecrypt were scrutinised by independent third parties in a public code and protocol review, hence we can consider this protocol as secure. After discontinuation, two main forks emerged:

- **Veracrypt**. It has a larger variety of block cyphers with respect to Truecrypt.
- **Ciphershed**. It has the same features as Truecrypt and it's simply maintained by fixing bugs.

Part III

Appendix

Appendix A

Algebra

Here is a list of useful theorems and definitions.

A.1 Groups

Definition A.1 (Group)**.** *A group is a pair*

$$(G, *) \tag{A.1}$$

where

- *G is the support of the structure, and*
- *$*$ is a binary, internal operation on G,*

and for which the following properties hold for the operation $*$:

- *associative property: $(a * b) * c = a * (b * c) \forall a, b, c \in G$*
- *existence of a neutral element: $\exists e \in G : \forall a \in G,\ a * e = e * a = a$*
- *existence of the inverses $\forall a \in G,\ \exists b : a * b = e$.*

If the operation $*$ also exhibits the following,

- *commutative property $\forall a, b \in G a * b = b * a$*

the group is said to be abelian or commutative.

Definition A.2 (Cyclic group)**.** *Let $(G, \cdot)$ be a group and $g \in G$ one of its elements. If any other element of the group can be obtained as g^i with $i \in \{\dots, -2, -1, 0, 1, 2, \dots\}$ by repeatedly applying the group operation to g or its inverse, then $(G, \cdot)$ is said to be a cyclic group with generator g and is denoted also as*

$$(G, \cdot) = (\langle G \rangle, \cdot)$$

Definition A.3 (Order of an element)**.** *Let $(G, \cdot)$ be a group and $g \in G$ be one of its elements. We define the order of $g \in G$, denoting it with $|g|$, $o(g)$ or $ord(g)$, the smallest positive integer n (assuming it exists) such that $g^n = e$. If n exists, the element $g \in G$ is said to be periodic, or with a finite order. Conversely, if there is no positive integer n such that $g^n = e$, the element g is said to have order 1 (or zero order).*

Theorem A.1 (Lagrange's theorem)**.** *Let $(G, \cdot)$ be a finite group of order n. If H is a subgroup of G, then its order divides n, i.e., $|H||n$.*

A.1.1 Groups of the remainders modulo n

The group $(\mathbb{Z}_n, +)$ is the group of the remainders of the division by n. We can observe the following facts:

- The group $(\mathbb{Z}_n, +)$ contains n elements, i.e., $\{0, 1, \ldots, n-1\}$.
- The inverse of any element $a \in \mathbb{Z}_n$ is computed as its opposite $-a \equiv |\mathbb{Z}_n| - a \mod n$.
- The group $(\mathbb{Z}_n, +)$ is cyclic and has $\varphi(n)$ generators.

A.1.2 Groups of the remainders modulo n with multiplicative inverse

The group $(\mathbb{Z}_n^*, \cdot)$ is the group of the remainders of the division by n with multiplicative inverse. Namely,

$$\forall a \in \mathbb{Z}_n^* \ \exists b : a \cdot b = 1$$

We can also observe the following facts:

- The group $(\mathbb{Z}_n^*, \cdot)$ contains $\varphi(n)$ elements.
- If n is prime, the group $(\mathbb{Z}_n^*, \cdot)$ has $\varphi(\varphi(n))$ generators.
- There exist a subgroup of $(\mathbb{Z}_n^*, \cdot)$ for each divisor d_i of $|\mathbb{Z}_n^*|$ containing d_i elements.
- An element x belongs to $\mathbb{Z}_p^*$ if and only if $gcd(x, p) = 1$.

Theorem A.2. *The group $(\mathbb{Z}_n^*, \cdot)$ is cyclic if and only if*

- *$n = 1, 2, 4$, or*
- *$n = p^k$, or*
- *$n = 2p^k$*

where $k \geq 1$ and $p \geq 3$ is a prime integer.

Proposition A.1 (Numerical Finite fields)**.** *The finite group $(\mathbb{Z}_p^*, \cdot)$ is cyclic and also the finite group $(\mathbb{Z}_p, +)$ is cyclic therefore, the structure $(\mathbb{Z}_p, +\cdot)$ is a finite field. The field $(\mathbb{Z}_p, +, \cdot)$ is also denoted as $\mathbb{Z}/(p)$ or $\mathbb{Z}/p\mathbb{Z}$.*

Inverting numbers in $(\mathbb{Z}_n^*, \cdot)$ is fundamental in some cryptosystems like RSA. The inverse of a number $e \in \mathbb{Z}_n^*$ can be computed

- using the Euclid algorithm, or
- using the group's properties.

Since $(\mathbb{Z}_n^*, \cdot)$ is a finite group with order $n = |\mathbb{Z}_n^*|$, each one of its elements will have an order dividing $\varphi(n)$ since the subgroups of a group have an order that divides the order of the group (Lagrange Theorem A.1). By definition of order (A.3), if $x \in \mathbb{Z}_n^*$ has order r, then $x^r \equiv 1$. Since the order r divides $\varphi(n)$, we can write $\varphi(n) = k \cdot r$, for some k. We can use what we've obtained to write

$$x^{\varphi(n)} \equiv x^{k \cdot r} \equiv (x^r)^k \equiv 1^k \equiv 1 \mod n \tag{A.2}$$

As a consequence, it is true that

$$x^{\varphi(n)} \equiv 1 \mod n \quad \forall x \in \mathbb{Z}_n^* \tag{A.3}$$

Therefore

$$\begin{aligned} x^{-1} &\equiv 1 \cdot x^{-1} \mod n && \text{(A.4)} \\ &\equiv x^{\varphi(n)} \cdot x^{-1} \mod n && \text{(A.5)} \\ &\equiv x^{\varphi(n)-1} \mod n \quad \forall x \in \mathbb{Z}_n^* && \text{(A.6)} \end{aligned}$$

This result can be stated in the following theorem.

Theorem A.3 (Euler's theorem)**.** *Let $(\mathbb{Z}_n^*, \cdot)$ be a group with multiplicative inverses and $x \in \mathbb{Z}_n^*$ a value coprime with n. Then,*

$$x^{\varphi(n)} \equiv 1 \mod n \quad \forall x \in \mathbb{Z}_n^* \tag{A.7}$$

A.2 Rings

Definition A.4 (Ring)**.** *A ring is an algebraic structure*

$$(R, +, \cdot)$$

with two binary operations $+$ and $\cdot$ such that the following properties hold:

1. *$(R, +)$ is an abelian group, and is commonly called additive group of the ring.*
2. *$\cdot$ is an internal, associative composition law on R, i.e. $\forall a, b, c \in R : (a \cdot b) \cdot c = a \cdot (b \cdot c) \in R$.*
3. *$\cdot$ is distributive with respect to $+$, that is $\forall a, b, c \in R : c \cdot (a + b) = c \cdot a + c \cdot b$ and $\forall a, b, c \in R : (a + b) \cdot c = c \cdot a + c \cdot b$.*

Definition A.5 (Zero divisor)**.** *Let $(R, +, \cdot)$ be a ring and a, b two elements of the ring. If $a \neq 0$ and $b \neq 0$ but $a \cdot b = 0$ then a and b are called zero divisors in R.*

Definition A.6 (Integral domain)**.** *We define integral domain a commutative ring with unity and without any zero divisors.*

Definition A.7 (Division ring)**.** *Let $(R, +, \cdot)$ be a ring such that $(R \setminus \{0\}, \cdot)$ is a group. Then $(R, +, \cdot)$ is called a division ring or skew field.*

Definition A.8 (Galois field)**.** *Every finite division ring is a commutative field and is called a Galois field.*

A.3 Fields

Definition A.9 (Field)**.** *A commutative ring $(\mathbb{F}, +, \cdot)$ whose non-null elements are all invertible is a field.*

A.3.1 Polynomial rings

Definition A.10 (Polynomial ring)**.** *Given a field $(\mathbb{F}, +, \cdot)$, the set*

$$\mathbb{F}[x] = \{a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 | a_i \in \mathbb{F}, n \geq 0\}$$

of the polynomials with coefficients over $\mathbb{F}$ is an integral domain with respect to the usual sum and product of polynomials, called polynomial ring in the unknown x over the field $(\mathbb{F}, +, \cdot)$.

Definition A.11 (Root of a polynomial)**.** *Let $(\mathbb{F}, +, \cdot)$ be a field and $\mathbb{F}[x]$ a polynomial ring over $(\mathbb{F}, +, \cdot)$. Let*

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{F}[x]$$

The element $r \in \mathbb{F}$ such that

$$f(r) = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0 = 0$$

is known as root of $f(x)$ where $+$, $\cdot$ the operations of $(\mathbb{F}, +, \cdot)$.

Definition A.12 (Reducible polynomial)**.** *Let $(\mathbb{F}, +, \cdot)$ be a field and $\mathbb{F}[x]$ a polynomial ring over $(\mathbb{F}, +, \cdot)$. A polynomial $f(x) \in \mathbb{F}[2]$ of degree n is defined reducible in $\mathbb{F}$ if there exist two*

polynomials $g(x), h(x) \in \mathbb{F}[x]$ *with* $0 < deg(g(x)) < n$ *and* $0 < deg(h(x)) < n$ *such that*

$$f(x) = g(x)h(x)$$

In case such polynomials do not exist $f(x)$ *is defined as irreducible in* $\mathbb{F}$.

A.3.2 Polynomial fields

Definition A.13 (Class of equivalence of polynomials)**.** *Let* $\mathbb{F}$ *be a field, and let* $f(x) \in \mathbb{F}[x]$ *be a fixed polynomial over* $\mathbb{F}$. *If* $a(x), b(x) \in \mathbb{F}[x]$, *then we say that* $a(x)$ *and* $b(x)$ *are congruent modulo* $f(x)$, *and we write*

$$a(x) \equiv b(x) \big(\mod f(x) \big)$$

if $f(x)|(a(x) - b(x))$ *The set*

$$\{b(x) \in \mathbb{F}[x] : a(x) \equiv b(x) \mod f(x)\}$$

is called the congruence class of $a(x)$ *(denoted by* $[a(x)]$*) and contains all polynomials with degree* ≥ 0 *and smaller than* $deg(f(x))$. *The set of all congruence classes modulo* $f(x)$ *will be denoted by* $F[x]/(f(x))$.

Proposition A.2. *Let* $\mathbb{F}$ *be a field, and let* $f(x)$ *be a nonzero polynomial in* $\mathbb{F}[x]$. *For any* $a(x) \in \mathbb{F}[x]$, *the congruence class* $[a(x)]$ *has a multiplicative inverse in* $F[x]/(f(x))$ *if and only if* $gcd(f(x), a(x)) \in \mathbb{F}$.

Proposition A.3. *Let* $\mathbb{F}$ *be a field and* $f(x) = a_n x^n + \cdots + a_0 \in \mathbb{F}[x]$ *an irreducible polynomial with degree* n *over* $\mathbb{F}$. *The polynomial* $f(y)$ *is reducible when it is considered as a polynomial with coefficients in* $\mathbb{K} = \mathbb{F}[x]/(f(x))$. *Indeed, it has at least one root in* $\mathbb{K}$ *equal to the coset* $x + \langle f(x) \rangle$.

A.4 Field extensions

Definition A.14 (Prime field)**.** *A field containing no proper subfield is called a prime field.*

Definition A.15 (Extension field)**.** *Let* $\mathbb{K}$ *be a subfield of* $\mathbb{F}$, *and* M *a generic subset of* $\mathbb{F}$. *The writing*

$$L = \mathbb{K}(M) \tag{A.8}$$

is used to denote the field defined as the intersection of all subfields of $\mathbb{F}$ *containing both* $\mathbb{K}$ *and* M. L *is called extension field of* $\mathbb{K}$. *If* M *includes a single element, namely* $M = \{\alpha\} \in \mathbb{F}$, *then* $L = \mathbb{K}(\alpha)$ *is called a simple extension of* $\mathbb{K}$, *and* α *is the defining element of* L *over* $\mathbb{K}$.

Since it will be important later on, a simple extension $\mathbb{K}(\alpha)$, with $\alpha \in F$ is the intersections of the subfields $\mathbb{F}$ of $\mathbb{K}$ that contain α. The idea behind an extension of $\mathbb{K}$ is to build a field L that contains all the elements in $\mathbb{K}$ and an extra element α which is not in $\mathbb{K}$. This means that we can write $\alpha \in L \setminus \mathbb{K}$.

Definition A.16 (Minimal polynomial)**.** *Let $\mathbb{K}$ be a proper subfield of $\mathbb{F}$, and let α be a generic element of $\mathbb{F}$, (i.e., $\alpha \in \mathbb{F}$). If $g(x) \in \mathbb{K}[x]$ is the monic polynomial with the least degree such that $g(\alpha) = 0$, then α is called algebraic element over K, whereas $g(x)$ is called minimal polynomial of α over $\mathbb{K}$.*

To sum things us, if we take

- a field $\mathbb{F}$,
- one of its subfields $\mathbb{K}$,
- an element $\alpha \in \mathbb{F}$
- a polynomial $g(x) \in \mathbb{F}[x]$ such that $g(x)$ is monic,

$$g(\alpha) = 0,$$

  g has minimal degree (to satisfy the properties just stated),

then

- $\alpha \in \mathbb{F}$ is an **algebraic element**, and
- $g(x)$ is a **minimal polynomial** of α over $\mathbb{K}$.

Algebraic elements are very important since they allow defining some important properties of the minimal polynomial. The following theorem states such properties.

Theorem A.4. *Let $\alpha \in \mathbb{F}$ be an algebraic element over $\mathbb{K}$. Its minimal polynomial $g(x) \in \mathbb{F}[x]$ has the following properties:*

1. *$g(x)$ is irreducible in $\mathbb{K}[x]$.*
2. *For $f(x) \in \mathbb{K}[x]$ we have $f(\alpha) = 0$, if and only if $g(x)$ divides $f(x)$, i.e.,*

$$f(\alpha) = 0 \iff g(x) | f(x) \tag{A.9}$$

Again, let us rewrite the theorem above more schematically to make things more clear. If we take

- a field $\mathbb{F}$,
- a subfield $\mathbb{K}$ of $\mathbb{F}$,
- a polynomial $f(x) \in \mathbb{F}[x]$,
- an algebraic element $\alpha \in \mathbb{F}$,

- the minimal polynomial $g(x) \in \mathbb{K}(\alpha)$ (i.e., $g(x) \in \mathbb{F}[x]$ s.t. $g(\alpha) = 0$),

then

1. $g(x)$ is irreducible in $\mathbb{K}[x]$, and
2. $f(\alpha) = 0 \iff g(x) | f(x)$.

Theorem A.5. *Let $L = \mathbb{K}(\alpha)$ be a simple algebraic extension field, with $f(x) \in \mathbb{K}[x]$ the minimal polynomial of α over $\mathbb{K}$. Then, L is isomorphic to the field $\mathbb{K}[x]/\langle f(x)\rangle$, i.e.,*

$$L = \mathbb{K}(\alpha) \simeq \mathbb{K}[x]/\langle f(x)\rangle \tag{A.10}$$

Let us summarise in a schematic way the theorem we have just mentioned. If we take

- a field $\mathbb{F}$,
- a subfield $\mathbb{K}$ of $\mathbb{F}$,
- an algebraic element $\alpha \in \mathbb{F}$ (i.e., such that $\exists f(x) \in \mathbb{K}[x]$ such that $g(\alpha) = 0$),
- an extension $L = \mathbb{K}(\alpha)$ with $f(x) \in \mathbb{K}[x]$ minimal polynomial,

then

$$L \simeq \mathbb{K}[x]/\langle f(x)\rangle \tag{A.11}$$

where $\mathbb{K}[x]/\langle f(x)\rangle$ is the field containing the classes of equivalence of the polynomials with remainders modulo $f(x)$.

Definition A.17 (Splitting field)**.** *Let $f(x) \in \mathbb{K}[x]$ be a polynomial with degree greater than zero, and $\mathbb{F}$ an extension field of $\mathbb{K}$. Then $f(x)$ is said to split in $\mathbb{F}$, if*

1. *$f(x)$ can be written as the product of linear factors with roots in $\mathbb{F}$, i.e.,*

$$f(x) = a\prod_i (x - \alpha_i), \quad \alpha \in \mathbb{F}, a \in \mathbb{K} \tag{A.12}$$

2. *$\mathbb{F}$ is an algebraic extension of β_i such that*

$$f(\beta_i) = 0, \ \forall i : \mathbb{F} = \mathbb{K}(\beta_1, \dots, \beta_i, \dots) \tag{A.13}$$

In this case, F is said to be a splitting field of $f(x)$ over $\mathbb{K}$.

A.5 Finite fields and polynomials

Let us now consider a field $\mathbb{K}$ with order $q = |\mathbb{K}|$. Thanks to Theorem A.5 we can build an extension $\mathbb{F}$ of $\mathbb{K}$ as

$$\mathbb{F} = \mathbb{K}[x]/\langle f(x)\rangle \tag{A.14}$$

with $f(x) \in \mathbb{K}[x]$ irreducible, $\alpha \in \mathbb{F} \setminus \mathbb{K}$ and $f(\alpha) = 0$. The order of the field $\mathbb{F}$ is $|\mathbb{F}| = |\mathbb{K}|^n$. We can also state the following properties:

1. $\mathbb{F}_p = \mathbb{Z}_p = \mathbb{Z}/\langle p \rangle = \mathbb{Z}/p\mathbb{Z}$ (with p prime) is a **prime subfield**.
2. In every finite field (prime or extended/composite) if $\mathbb{F}_p$ is its prime subfield, then for any element $a \in \mathbb{F}$ is it true that:
$$pa = a + a + \cdots + a = 0 \tag{A.15}$$
The value p is called characteristic of the field $\mathbb{F}$.
3. For all $a \in \mathbb{K}$, with $q = |\mathbb{K}|$ it is always true that $a^q = a$.
4. Let p be the characteristic of the finite field $\mathbb{K}$, then $\forall a, b \in \mathbb{K}, m \geq 1$ we have
$$(a \pm b)^{p^m} = a^{p^m} \pm b^{p^m} \tag{A.16}$$

Proposition A.4. *Every finite field $\mathbb{K}$ has order p^n where p is the characteristic of K and $n \geq 1$.*

Theorem A.6 (Existence and uniqueness of a finite field)**.** *For every prime p and every positive integer n, there exists a finite field $\mathbb{F}_q$ with $q = p^n$ elements. In particular, every finite field $\mathbb{F}_q$ with $q = p^n$ elements is isomorphic to the splitting field of*
$$x^q - x = \prod_{a \in \mathbb{F}_q} (x - a) \in \mathbb{Z}_p[x] \tag{A.17}$$

Theorem A.7 (Order of finite subfields)**.** *Let $\mathbb{F}_q$ be the finite field with $q = p^n$ elements. Then every subfield of $\mathbb{F}_q$ has order p^m, where m is a positive divisor of n. Conversely, if m is a positive divisor of n, then there exists exactly one subfield of F_q with p_m elements.*

Theorem A.8 (Order of the multiplicative field)**.** *For every finite field $\mathbb{F}_q$, the multiplicative group $\mathbb{F}_q^* = \mathbb{F}_q \setminus \{0\}$ is cyclic with order $q - 1$.*

Definition A.18 (Primitive element)**.** *A generator of the cyclic group $\mathbb{F}_q^*$ is called primitive element of $\mathbb{F}_q$.*

Now we are ready to analyse the key concepts for finite fields and polynomials. Let us take two finite fields $\mathbb{H}$ and $\mathbb{K}$ such that $\mathbb{H}$ is a subfield of $\mathbb{K}$, i.e.,
$$\mathbb{H} < \mathbb{K} \tag{A.18}$$
Now we want to show that $\mathbb{K}$ is a simple algebraic extension of $\mathbb{H}$ defined by any primitive element $\xi \in \mathbb{K}$, i.e.,
$$\mathbb{K} = \mathbb{H}(\xi) \tag{A.19}$$

Let's start by showing that ξ is algebraic over $\mathbb{H}$. Since $\mathbb{H}$ is a finite field, it is isomorphic to the splitting field of

$$h(x) = x^q - x \tag{A.20}$$

where $q = |\mathbb{H}|$. Since $\mathbb{H}$, by definition of extension field, contains all the elements of $|K|$ and the algebraic element, then the cardinality of $\mathbb{H}$ is equal to the cardinality of $\mathbb{K}$ minus 1. We can therefore say that $\mathbb{H}$ is isomorphic to the splitting field of

$$h(x) = x^{|\mathbb{K}|-1} - x \tag{A.21}$$

ξ is a root of $x^{|\mathbb{K}|-1} - 1 \in \mathbb{H}[x]$. Moreover, the extension $\mathbb{H}[x]$ is contained in $\mathbb{K}$, hence it must be that ξ is algebraic in $\mathbb{H}$. Thanks to Theorem A.6

There exists a minimal polynomial for ξ over $\mathbb{H}$. This polynomial is irreducible over $\mathbb{H}$.

The finite field $\mathbb{F}_{p^n}$ is a simple extension of $\mathbb{Z}_p$, defined by a primitive element of F_{p^n}. This element must have a minimal polynomial with degree n over $\mathbb{Z}_p$ thus, there always exists an irreducible polynomial with degree n in $\mathbb{Z}_p[x]$.

Theorem A.9 (Primitive elements)**.** *If g is a primitive element of a finite field $\mathbb{F}_q$, then every primitive element is in*

$$\{g^h : gcd(q-1, h) = 1, h = 1, 2, \dots, q-2\}$$

In particular, the number of primitive elements is $\varphi(q-1)$.

Theorem A.10 (Primitive elements)**.** *Let $\mathbb{F}_p$, with p prime, be a finite field and g a primitive element, then, for all r, such that $r|p-1$, it holds*

$$g^{\frac{p-1}{r}} \not\equiv 1 \mod p$$

Definition A.19 (Primitive polynomial)**.** *Let $g(x) \in \mathbb{F}_p[x]$ be an irreducible monic polynomial with degree $n = deg(g(x))$. If $g(x)$ is the minimal polynomial of a primitive element $\alpha \in \mathbb{F}_{p^n}$ then $g(x)$ is called a primitive polynomial.*

This means that if we want to check if a polynomial $f(x)$ is primitive, we have to take a value $\alpha \in \mathbb{F}_{p^n}$ such that and check if α is primitive. Thanks to definition A.18, we know that α is primitive in $\mathbb{F}_{p^n}$ if it's a generator of $\mathbb{F}^*_{p^n}$. But α is a generator of $\mathbb{F}^*_{p^n}$ if $\alpha^{\frac{|\mathbb{F}^*_{p^n}|}{i}} \not\equiv 1$ for all i that divide $|\mathbb{F}^*_{p^n}|$, hence a polynomial is primitive if

$$\alpha^{\frac{|\mathbb{F}^*_{p^n}|}{r}} \equiv \alpha^{\frac{|p^n-1|}{r}} \not\equiv 1 \quad \forall r : r|p^n - 1$$

Theorem A.11 (Irreducibility criterion)**.** *Let $f(x) \in \mathbb{F}_p[x]$ be a monic polynomial with degree m. Then, $f(x)$ is irreducible in $\mathbb{F}_p[x]$ if and only if the value of*

$$gcd(f(x), x^{p^h} - x)$$

is a constant (i.e., a zero-degree polynomial) for all the values of $h \leq \lfloor \frac{m}{2} \rfloor$

Theorem A.12 (Reducibility criterion)**.** *Let $f(x) \in F_p[x]$ be a polynomial with degree $n = deg(f(x))$. $f(x)$ is reducible if there exists $a \in Fp[x]/\langle f(x)\rangle$ such that $a^{p^n} \not\equiv a$.*

Definition A.20 (Polynomial remainder field)**.** *Let $p \in \mathbb{N}$ be a prime number, $n \in \mathbb{N} \setminus \{0\}$. Let $Q(x) \in \mathbb{F}_p$ be an irreducible polynomial of degree n. We call $\mathbb{F}_{p^n}$ the field*

$$\mathbb{F}_{p^n} = {}^{\mathbb{F}_p[x]}\!/_{\langle Q(x)\rangle}$$

The field $\mathbb{F}_{p^n}$ is the field of the remainders' class with coefficients in $\mathbb{F}_p$ in which two polynomials are equivalent if they have the same remainder when divided by $Q(x)$.

A.5.1 Ruffini's theorem

The following theorem can be used to check if a polynomial is irreducible or not.

Theorem A.13 (Ruffini)**.** *Let $f(x) \in K[x] \setminus \{0\}$ be a polynomial. If $\alpha \in K$, then the remainder of the division of $f(x)$ by $x - \alpha$ is $f(\alpha)$, namely*

$$f(x) = (x - \alpha) \cdot q(x) + f(\alpha)$$

In particular, if α is a root of the polynomial, namely $f(\alpha) = 0$, then $x - \alpha$ divides $f(x)$.

This theorem is very useful since a polynomial of degree at most 3 has a divisor of degree 1 if it's reducible. This means that if we have a polynomial $f(x) \in K[x]$ of degree smaller or equal to 3, then we check, for each $a \in K$, if $f(a) = 0$. If we find a value $a \in K$ such that $f(a) = 0$, then we know, thanks to Theorem A.13, that $x - a$ divides $f(x)$, hence $f(x)$ is reducible. This is true only if the degree of the polynomial is smaller or equal to 3. If this isn't true, we have to use the criterion in A.12.

A.5.2 Summary

Let's summarise what we have seen in this section regarding finite fields. If we take

- a field $\mathbb{F}_p$,
- an element $\alpha \notin \mathbb{F}_p$, called **primitive element** (which is an algebraic element) such that $\mathbb{F}^*_{p^n} = \langle\alpha\rangle$, and
- a polynomial $g(x) \in \mathbb{F}_p[x]$, called **primitive polynomial** (which is a minimal polynomial) such that $g(\alpha) = 0$ and $deg(g) = n$

then we can use $g(x)$ to build the field of residual classes of the division by $g(x)$, namely

$$\mathbb{F}_p(\alpha) \simeq {}^{\mathbb{F}_p[x]}\!/_{\langle g(x)\rangle} \simeq \mathbb{F}_{p^n}$$

A.6 Characterisation of finite fields with polynomials

The number of irreducible polynomials over $\mathbb{F}_p$ with degree d and d prime is:

$$N_d(p) = \frac{p^d - p}{d}$$

The number of irreducible polynomials over $\mathbb{F}_p$ with degree n and n composite is:

$$N_n(p) = \frac{p^n - \sum_{d \in \{a|n\} \setminus \{n\}} N_d(p)}{n}$$

The number of primitive polynomials over $\mathbb{F}_{p^n}$ with degree d is:

$$M_d(p) = \frac{\varphi(p^n - 1)}{d}$$

To find a primitive element in $\mathbb{F}_{p^n}$ we consider $\alpha \in \mathbb{F}_{p^n} \setminus \mathbb{F}_p$ and check that

$$\alpha^i \not\equiv 1 \mod p \quad \forall i \in \{a : a|p^n - 1\}$$

If $g = \alpha$ is a primitive element of a finite field $\mathbb{F}_q$, then every primitive element is in

$$\{g^h = \alpha^h : gcd(q-1, h) = 1, h = 1, 2, \ldots, q-2\}$$

Let $p \in \mathbb{N}$ be a prime number, $n \in \mathbb{N}^+$ a positive number and $Q(x) \in \mathbb{F}[x]$ an irreducible polynomial. Consider a field

$$\mathbb{F}_{p^n} = {}^{\mathbb{F}_p[x]}\!\diagup_{\langle Q(x) \rangle}$$

Every primitive polynomial with degree n has n roots which are the primitive elements of the field $\mathbb{F}_{p^n}$ and can be computed, starting from one primitive element α, as

$$\alpha^{p^i} \quad \forall i \in \{0, 1, \ldots, n-1\}$$

This means that we can build a primitive polynomial by taking one primitive element α and computing its roots. The roots

$$\{\alpha^i : gcd(i, p^n - 1)\}$$

are used to write one primitive polynomial

$$f(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^{i_1}) \cdots (x - \alpha^{i_l}) \quad s.t.\ gcd(i_x, p^n - 1) = 1$$

or equivalently

$$f(x) = \prod_{i\ s.t.\ 0 \le i \le p^n - 1\ \wedge\ gcd(i, p^n - 1) = 1} (x - \alpha^i)$$

A.7 Interesting theorems

Here is a list of theorems that are very important for different reasons in cryptography.

A.7.1 Fermat's little theorem

Fermat's little theorem comes from Euler's theorem. In fact, it's a specification of Euler's theorem for when n is a prime number.

Theorem A.14 (Fermat's little theorem). *Let p be a prime number, $(\mathbb{Z}_p^*, \cdot)$ be a multiplicative group and $a \in \mathbb{Z}_p^*$, then*

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

A.7.2 Lagrange theorem

The following theorem can be used to list all the subgroups of a group.

Theorem A.15 (Inverse of Lagrange's theorem for abelian group). *Let $(G, \cdot)$ be a finite abelian group of order n. For each divider m of n (i.e., $m|n$), there exists at least a subgroup H with order m.*

A.7.3 Chinese remainder's theorem

Theorem A.16 (Chinese remainder's theorem). *Let $n_1, \ldots, n_k \in \mathbb{N} \setminus \{0, 1\}$ be natural numbers such that*

$$gcd\{n_i, n_j\} = 1 \quad \forall 1 \leq i, j \leq k, i \neq j$$

Let n be the product of $n_1, \ldots, n_k$,

$$n = n_1 n_2 \ldots n_k$$

Then the function

$$\psi : \mathbb{Z}_n \to \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}$$

defined as

$$x \mod n \mapsto (x \mod n_1, \ldots, x \mod n_k)$$

is an isomorphism of rings (i.e., it's bijective).

Theorem A.17 (Chinese remainder's theorem). *Let $n_1, ..., n_k$ be k positive integers pairwise coprime, and let $x_1, ..., x_k$ be k elements of $\mathbb{Z}$. The following system of modular congruences*

$$\begin{cases} X \equiv x_1 \mod n_1 \\ X \equiv x_2 \mod n_2 \\ \vdots \\ X \equiv x_k \mod n_k \end{cases}$$

has an unique solution $\bar{X}$ such that $0 < \bar{X} < N$ with $N = \prod_{i=1}^{k} n_i$

The integer $\bar{X} \equiv X \mod N$ used in Theorem A.17 can be computed as

$$\bar{X} \equiv X \equiv \left(\sum_{i=1}^{k} M_i \cdot M_i' \cdot x_i \right) \mod N \tag{A.22}$$

where

- $M_i = \frac{N}{n_i}$
- $M_i' = M_i^{-1} \mod n_i$

Appendix B

Algorithms

B.1 Square and multiply

The square and multiply algorithm can be applied in two variants:

- The left-to-right square and multiply algorithm.
- The right-to-left square and multiply algorithm.

In both cases, we have to write the exponent n in some base b, hence we can write n as

$$n = n_{t-1}b^{t-1} + n_{t-2}b^{t-2} + \cdots + n_1 b^1 + n_0$$

B.1.1 Left-to-right square and multiply

The left-to-right square and multiply algorithm uses the following formula to rise a to the power of n:

$$\begin{aligned} c = a^n &= ((\cdots((a^{n_{t-1}})^b \cdot a^{n_{t-2}})^b \cdots)^b \cdot a^{n_1})^b \cdot a^{n_0} \\ &= ((\cdots(a^{n_{t-1}b} \cdot a^{n_{t-2}})^b \cdots)^b \cdot a^{n_1})^b \cdot a^{n_0} \\ &= ((\cdots a^{n_{t-1}b^2} \cdot a^{n_{t-2}b} \cdots)^b \cdot a^{n_1})^b \cdot a^{n_0} \\ &= (a^{n_{t-1}b^{t-2}} \cdot a^{n_{t-2}b^{t-3}} \cdots a^{n_1})^b \cdot a^{n_0} \\ &= a^{n_{t-1}b^{t-1}} \cdot a^{n_{t-2}b^{t-2}} \cdots a^{n_1 b} \cdot a^{n_0} \\ &= a^{n_{t-1}b^{t-1} + n_{t-2}b^{t-2} + \cdots + n_1 b + n_0} \end{aligned}$$

If we consider a base-two representation of the exponent n, we obtain the following formula:

$$c = a^n = ((\cdots((a^{n_{t-1}})^2 \cdot a^{n_{t-2}})^2 \cdots)^2 \cdot a^{n_1})^2 \cdot a^{n_0}$$

The computational cost of this method, expressed in terms of squarings and multiplications needed to compute the final result, is (on average) $t-1$ squarings, plus $\frac{1}{2}(t-1)$ multiplications, with $t = \lceil \log_2 n \rceil$, namely

$$T_{ltr}(n) = (\lceil \log_2 n \rceil - 1) + \left(\frac{1}{2}(\lceil \log_2 n \rceil - 1)\right)$$

B.1.2 Right-to-left square and multiply

The right-to-left square and multiply algorithm uses the following formula to rise a to the power of n:

$$\begin{aligned} c = a^n &= (a^{n_{t-1}})^{b^{t-1}} \cdot (a^{n_{t-2}})^{b^{t-2}} \cdots (a^{n_1})^{b^1} \cdot (a^{n_0})^{b^0} \\ &= a^{n_{t-1}b^{t-1}} \cdot a^{n_{t-2}b^{t-2}} \cdots a^{n_1 b^1} \cdot a^{n_0 b^0} \\ &= a^{n_{t-1}b^{t-1} + n_{t-2}b^{t-2} + \cdots + n_1 b^1 + n_0 b^0} \end{aligned}$$

If we consider a base-two representation of the exponent n, we obtain the following formula:

$$c = a^n = (a^{n_{t-1}})^{2^{t-1}} \cdot (a^{n_{t-2}})^{2^{t-2}} \cdots (a^{n_1})^{2^1} \cdot (a^{n_0})^{2^0}$$

The computational cost of this method, expressed in terms of squarings and multiplications needed to compute the final result, is (on average) $t-1$ squarings, plus $\frac{1}{2}(t-1)$ multiplications, with $t = \lceil \log_2 n \rceil$, namely

$$T_{rtl}(n) = (\lceil \log_2 n \rceil - 1) + \left(\frac{1}{2} (\lceil \log_2 n \rceil - 1) \right)$$

Algorithm 11 The right-to-left square and multiply algorithm.

```
procedure SquareAndMultiply(b, e)
    bin ← Bin(e)                    ▷ Get the binary encoding of the exponent e
    len ← Len(bin)                  ▷ The number of bits used to encode e
    pow ← 1                         ▷ The result of b^e
    for i ∈ {0, ..., len − 1} do
        if bin[len − i − 1] ≠ 0 then
            pow ← pow · bin[len − i − 1] · b
        end if
        b ← b · b
    end for
    return pow
end procedure
```

Appendix C

Multiprecision arithmetic

C.1 Notation

A number N can be written in base b as

$$N = \sum_{i=0}^{n-1} N_i b^i \tag{C.1}$$

where

- N_i is a **digit**, namely a value $0 \leq N_i < b$ (with $N_{n-1} \neq 0$).
- b is the **base**.

For instance, if we want to store a number in memory, we can consider $b = 256$, hence a digit is a byte in memory. Considering $b = 2^{64}$, a digit is a memory word for 64-bit architectures.

A number N written as in Equation C.1 can be stored in an array of n cells, each containing a value N_i. If a number is stored in the array `a`, the most significative symbol N_{n-1} is stored in position $n - 1$, hence in

```
a[n-1]
```

Using the same notation we can write a number x as

$$x = \sum_{i=0}^{l-1} x_i b^i$$

C.1.1 Additions and subtraction

Addition and subtraction work like pen-and-paper addition and subtraction, hence we consider one digit at a time starting from the less significant one and we compute the sum or the subtraction between single digits. When computing the sum we have to propagate the carrying the column on the left. When computing the subtraction we have to carry the borrow to the column on the left.

Algorithm 12 shows the algorithm for adding two numbers. Algorithm 13 shows the algorithm for subtracting two numbers.

Algorithm 12 The algorithm for computing a multi-precision addition.

Input: $A = A_{n-1}b^{n-1} + \cdots + A_1 b + A_0$
Input: $B = B_{n-1}b^{n-1} + \cdots + B_1 b + B_0$
Output: $x = A + B = x_n b_n + \cdots + x_1 b + x_0$
$x = (x_0, \ldots, x_{n-1}) \leftarrow (0, \ldots, 0)$
$k_{out} \leftarrow 0$ ▷ The remainder of the previous sum
for $i \leftarrow \{0, \ldots, n-1\}$ **do**
    $k_{in} \leftarrow k_{out}$
    $x_i \leftarrow (A_i + B_i) \mod b$
    $tmp \leftarrow (x_i + k_{in}) \mod b$ ▷ Add the previous remainder
    $k_{out} \leftarrow (x_i < A_i) + (tmp < x_i)$ ▷ True conditions are evaluated as 1, false conditions as 0
    $x_i \leftarrow tmp$
end for
$x_n \leftarrow k_{out}$
return x

Algorithm 13 The algorithm for computing a multi-precision subtraction.

Input: $A = A_{n-1}b^{n-1} + \cdots + A_1 b + A_0$
Input: $B = B_{n-1}b^{n-1} + \cdots + B_1 b + B_0$
Output: $x = A - B = x_n b_n + \cdots + x_1 b + x_0$
$x = (x_0, \ldots, x_{n-1}) \leftarrow (0, \ldots, 0)$
$k_{out} \leftarrow 0$
for $i \leftarrow \{0, \ldots, n-1\}$ **do**
    $k_{in} \leftarrow k_{out}$
    $x_i \leftarrow (A_i - B_i) \mod b$
    $tmp \leftarrow (x_i - k_{in}) \mod b$
    $k_{out} \leftarrow (x_i > A_i) + (tmp > x_i)$ ▷ True conditions are evaluated as 1, false conditions as 0
    $x_i \leftarrow tmp$
end for
$x_n \leftarrow k_{out}$
return x

C.2 Multiplication

There exist multiple algorithms for computing the product between two numbers. Each algorithm is better suited for numbers of different sizes. The best algorithms we know are:

- **Schoolbook multiplication**. Its complexity is $\Theta(n^2)$ and it's suited for inputs encoded smaller than 320 bits, namely
$$n < 320$$
- **Karatsuba**: Its complexity is $\Theta(3n^{\log_2(3)})$ and it's suited for inputs between 320 and 10000, namely
$$320 < n < 10000$$
- **Toom-Cook**: Its complexity is $\Theta(n^{\log_d(2d-1)})$ with $d \geq 3$ and it's suited for inputs between 10000 and 32000, namely
$$10000 < n < 32000$$
- **Schönage-Strassen**: Its complexity is $\Theta(n \log(n) \log(\log(n)))$ with $d \geq 3$ and it's suited for inputs greater than 32000, namely
$$n > 32000$$

C.2.1 Schoolbook multiplication

Algorithm 14 The multi-precision schoolbooks multiplication algorithm.

Input $A = A_{n-1}b^{n-1} + \cdots + A_1 b + A_0$
Input $B = B_{n-1}b^{n-1} + \cdots + B_1 b + B_0$
Output $x = A \cdot B = x_{2n-1}b^{2n-1} + \cdots + x_1 b + x_0$
$x \leftarrow 0$
for $i \in \{0, \ldots, n-1\}$ **do**
  $x \leftarrow x + A_i B b^i$
end for
return x

C.3 The Montgomery modular multiplication algorithm

Many cryptographic operations are based on modular arithmetic, hence it's a good idea to find efficient algorithms for computing modular operations. The Montgomery multiplication algorithm allows computing the modular multiplication between two integers at almost the same cost as a normal integer multiplication. In particular, a Montgomery multiplication requires

$$2n^2 + 2n$$

single-precision multiplications whereas a regular multi-precision multiplication requires

$$n^2$$

single-precision multiplications.

To describe the Montgomery multi-precision multiplication algorithm we have to analyse the algorithm for computing the single-precision multiplication, first.

Algorithm 15 The optimised multi-precision schoolbooks multiplication algorithm.

Input $A = A_{n-1}b^{n-1} + \cdots + A_1 b + A_0$
Input $B = B_{n-1}b^{n-1} + \cdots + B_1 b + B_0$
Output $x = A \cdot B = x_{2n-1}b^{2n-1} + \cdots + x_1 b + x_0$
$x \leftarrow 0$
for $j \in \{0, \ldots, n-1\}$ **do**
  $k \leftarrow 0$
  for $i \leftarrow \{0, \ldots, n-1\}$ **do**
    $(u, v)_b \leftarrow A_i B_j$
    $v \leftarrow v + k$
    $u \leftarrow u + (v < k)$ ▷ True = 1, False = 0
    $v \leftarrow v + x_{i+j}$
    $u = u + (v < x_{i+j})$ ▷ True = 1, False = 0
    $x_{i+j} \leftarrow v$
    $k \leftarrow u$
  end for
  $x_[j + n] \leftarrow k$
end for
return x

C.3.1 Single precision multiplication

Let N be a positive integer and let us consider the ring

$$\mathbb{Z}_N = \{0, 1, \ldots, N-1\} \tag{C.2}$$

In this ring, we call Montgomery transformation the permutation of the elements of $\mathbb{Z}_n$ defined as:

$$\mu : (\mathbb{Z}_N, +, \cdot) \to (\tilde{\mathbb{Z}_N}, +, *) \tag{C.3}$$

The Montgomery transformation let us move to a different ring with:

- A support $\tilde{\mathbb{Z}_N}$, called **Montgomery domain**, which is a permutation of $\mathbb{Z}_N$.
- The same modular addition as $\mathbb{Z}_N$.
- A new modular multiplication $*$ which is more efficient than $\cdot$.

This means that we can

1. apply the Montgomery transformation to move to $(\tilde{\mathbb{Z}_N}, +, *)$,
2. compute the multiplication in $\tilde{\mathbb{Z}_N}$ using $*$, and
3. go back to $(\mathbb{Z}_N, +, \cdot)$.

The Montgomery transformation is a homomorphism of rings hence it holds:

$$\mu(a + b \mod N) = \mu(a \mod N) + \mu(b \mod N) \tag{C.4}$$

and

$$\mu(a \cdot b \mod N) = \mu(a \mod N) * \mu(b \mod N) \tag{C.5}$$

Montgomery transformation

Now we have to define operatively the Montgomery transformation. Let us consider an element $x \in \mathbb{Z}_N$ and a positive integer $R > N$, called **Montgomery radix**, coprime with N (i.e., $gcd(R, N) = 1$). Let us define the parameters R' and N' as

$$gcd(R, N) = 1 = R \cdot R' - N' \cdot N \tag{C.6}$$

The numbers R' and N' should satisfy the relations

$$R' \equiv R^{-1} \mod N \quad \text{s.t. } 0 \leq R' < N \tag{C.7}$$

and

$$N' = -N^{-1} \mod R \quad \text{s.t. } 0 \leq N' < R \tag{C.8}$$

Thanks to these parameters, we can define the Montgomery transformation as

$$\tilde{x} = \mu(x) = x \cdot R \mod N \tag{C.9}$$

and the inverse Montgomery transformation as

$$x = \mu^{-1}(\tilde{x}) = \tilde{x} \cdot R' \mod N \tag{C.10}$$

Computing the Montgomery transformation boils down to computing the Montgomery radix R. The Montgomery radix must satisfy the following conditions:

- $R > N$,
- $gcd(R, N) = 1$,

hence a good choice for R is

$$R = 2^k$$

for some k, which also allows an efficient implementation since a multiplication for 2^k is simply a k-bit left shift.

Montgomery multiplication

Now that we can move to and from the Montgomery domain, we have to define the $*$ operation. Let us consider the following problem. Given $x \in \mathbb{Z}_n$ we want to compute

$$x = \frac{x}{R} \mod N \quad 0 < x < R \cdot N \tag{C.11}$$

as efficiently as possible. This operation is called **Montgomery reduction**. If we encode x in base R and the least significant bit of x in base R is a 0, then we efficiently implement $\frac{x}{R}$ as a 1-bit right shift. The problem is that the representation of x in base R doesn't always have a 0 in the least significant position. In particular, the least significant digit in base R is a 0 only if x is divisible by R. To solve this problem, we can add a positive integer $t \in \mathbb{N}$ to x such that:

- $x + t \equiv 0 \mod R$, namely the value $x + t$ should be divisible by R.
- $x + t \equiv x \mod N$, namely the result of the division $\frac{x}{R}$ should be the same as $\frac{x+t}{R}$.

We obtain the system

$$\begin{cases} x + t \equiv 0 \mod R \\ x + t \equiv x \mod N \end{cases} \tag{C.12}$$

The second equation tells us that it must hold

$$t \equiv 0 \mod N \tag{C.13}$$

hence t must be divisible by N and we can write t as

$$t = t'N \tag{C.14}$$

with $t' \in \mathbb{N}^+$. If we replace Equation C.14 into the first equation of the system in C.12 we get

$$\begin{cases} x + t'N \equiv 0 \mod R \\ t = t'N \end{cases} \tag{C.15}$$

We can now isolate t' and rewrite the first equation in C.15 as

$$t' \equiv (-N^{-1})x \equiv N'x \mod R \tag{C.16}$$

We can then compute t as

$$t = t'N = (N'x)N$$

The value of N' can be computed offline knowing N and R using Euclid's algorithm. Algorithm 18 shows the algorithm for computing the Montgomery reduction.

Algorithm 16 The Montgomery reduction algorithm.

Input N
Input R
Input $0 < x \leq R \cdot N$
Output $xR^{-1} \mod N$
procedure MONTGOMERYREDUCTION(N, R, x)
    $t' \leftarrow xN' \mod R$ ▷ Cost: 1
    $x \leftarrow x + Nt'$ ▷ Cost: 1
    $x \leftarrow \frac{x}{R}$ ▷ Cost: 1 because it's a shift
    if $x \geq N$ **then**
        $x \leftarrow x - N$
    end if
    return x
end procedure

We can now use the Montgomery reduction $xR^{-1} \mod N$ to compute the Montgomery multiplication. Given two elements $\tilde{x}, \tilde{y} \in \tilde{\mathbb{Z}_N}$ we can compute $\tilde{x} * \tilde{y} \mod N$ as

$$\tilde{x} * \tilde{y} \equiv (\tilde{x} \cdot \tilde{y}) \cdot R^{-1} \mod N \tag{C.17}$$

where $\cdot$ is the usual multiplication (not the Montgomery multiplication). The result of $\tilde{x} * \tilde{y} \mod N$ still belongs to $\tilde{\mathbb{Z}_N}$, in fact

$$\begin{aligned}
\tilde{x} * \tilde{y} &\equiv (\tilde{x} \cdot \tilde{y}) \cdot R^{-1} \mod N && \text{(C.18)}\\
&\equiv (x \cdot R \cdot y \cdot R) \cdot R^{-1} \mod N && \text{(C.19)}\\
&\equiv x \cdot y \cdot R^2 \cdot R^{-1} \mod N && \text{(C.20)}\\
&\equiv x \cdot y \cdot R \mod N && \text{(C.21)}\\
&\equiv \tilde{xy} \mod N && \text{(C.22)}
\end{aligned}$$

Thanks to this definition, the Montgomery multiplication requires:

- One multiplication for computing $\tilde{x} \cdot \tilde{y}$.
- One multiplication for computing the reduction.
- One shift for computing the reduction.
- One addition for computing the reduction.

This is worst than a single multiplication (when no modulo is involved) but better than computing the modulo reduction. Algorithm 17 shows the Montgomery multiplication algorithm.

Algorithm 17 The Montgomery multiplication algorithm.

Input $\tilde{x} = x \cdot R \mod N$
Input $\tilde{y} = y \cdot R \mod N$
Output $x \cdot y \cdot R \mod N$
procedure MONTGOMERYMULTIPLICATION($\tilde{x}$, $\tilde{y}$)
  return MONTGOMERYREDUCTION($\tilde{x} \cdot \tilde{y}$)
end procedure

Note that the Montgomery multiplication primitive can be used to compute the Montgomery transformation, too. In particular, given a message $m \in \mathbb{Z}_N$ and the radix R, we can compute

$$\begin{aligned}
m \times R^2 &= m \cdot R^2 \cdot R^{-1} \mod N\\
&= m \cdot R \mod N\\
&= \mu(m) \mod N
\end{aligned}$$

Note that we have considered m in $\mathbb{Z}_n$ and not in $\tilde{\mathbb{Z}_N}$, however, the Montgomery multiplication still applies.

C.3.2 Multi-precision multiplication

Now that we know how to efficiently multiply two single-precision values using the Montgomery multiplication algorithm, we want to extend the same idea to multi-precision multiplication. Let us write a multi-precision modulo N written in base b as in Equation C.1 with d digits as

$$N = (N_{d-1}, N_{d-2}, \ldots, N_1, N_0) \quad \text{(C.23)}$$

and a number x, written in base b with the same number of digits as

$$x = (x_{d-1}, x_{d-2}, \ldots, x_1, x_0) \tag{C.24}$$

Note that if x can be represented with less than d bits, we still have to consider the number of bits required to encode N.

Multi-precision Montgomery reduction

Before describing the multi-precision algorithm, let us highlight that we represent with

$$x^{(i)}$$

the value of x computed at the i-th iteration and not the i-th power of x. The initial value of x is simply represented as x. Let us also consider the Montgomery radix $R = b^d$. The core of the Montgomery algorithm is the Montgomery reduction. Let us then understand how to handle this operation. Since $R = b^d$, we can write the reduction operation as

$$\frac{x}{R} \equiv \frac{x}{b^d} \mod N \tag{C.25}$$

But since $b^d = b \cdot \cdots_{d\ times} \cdot b$, then we can write the reduction as

$$\left(\cdots \left(\left(\frac{x}{b} \mod N \right) \cdot \frac{1}{b} \mod N \right) \cdots \frac{1}{b} \right) \tag{C.26}$$

This means that the division by R can be seen as the division by b iteratively applied d times. Let's start by analysing the innermost division, namely $\frac{x}{b} \mod N$. This is a single-precision reduction in base b, hence we can write

$$x = \frac{x}{R} \mod N \iff x^{(0)} = \frac{x + t'}{b} \mod N \tag{C.27}$$

or, using an algorithmic notation,

$$x \leftarrow \frac{x + t'}{b} \mod N \tag{C.28}$$

As for the single precision Montgomery multiplication, we have to choose t' such that it doesn't modify the result of the division but allows us to write the nominator as a multiple of b. As a result, we can write

$$\begin{cases} x^{(0)} + t' \equiv 0 \mod b \\ x^{(0)} + t' \equiv x^{(0)} \mod N \end{cases} \tag{C.29}$$

From the second equation, we obtain

$$t' \equiv 0 \mod N \tag{C.30}$$

hence

$$t' = tN \tag{C.31}$$

Replacing this value in the first equation we get

$$x^{(0)} + tN \equiv 0 \mod b \tag{C.32}$$

Since the operation is modulo b, what really matters is the least significant digit, hence we can write

$$x_0^{(0)} + tN_0 \equiv 0 \mod b \tag{C.33}$$

We can now extract t and obtain

$$\begin{cases} t \equiv (-N_0^{-1})x_0 \equiv N_0' x_0 \mod b \\ t' = tN \end{cases} \tag{C.34}$$

replacing this result in Equation C.27 we obtain

$$x^{(0)} \equiv \frac{x + (N_0' x_0 \mod b)N}{b} \mod N \tag{C.35}$$

Note that the value of t' depends only on the first digit of x, i.e., x_0, and on the first digit of N', i.e., N_0'. If we delay the division by b and the modular reduction by N, we get

$$x^{(0)} \equiv x + (N_0' x_0 \mod b)Nb^0 \tag{C.36}$$

This means that the number x can be represented with $d+1$ digits and the least significant one, i.e., x_0, is 0 (because a multiplication by b is a left shift by one digit). In formulas, we have

$$x^{(0)} = (x_d, x_{d-1}, \ldots, x_2, x_1, 0) \tag{C.37}$$

We now move to the next iteration. Since in the previous iteration we have postponed the division by b, we now want to compute $\frac{x^{(0)}}{b}$. This means that we have to compute

$$x^{(1)} = \frac{\frac{x^{(0)}}{b}}{b} \tag{C.38}$$

or, algorithmically,

$$x \leftarrow \frac{\frac{x}{b}}{b} \tag{C.39}$$

By using the same trick of adding a value t' we obtain

$$x^{(1)} = \frac{\frac{x^{(0)}}{b} + t'}{b} \tag{C.40}$$

and we can compute t' thanks to the system

$$\begin{cases} t \equiv (-N_0^{-1})x_1^{(0)} \equiv N_0' x_1^{(0)} \mod b \\ t' = tN \mod b \end{cases} \tag{C.41}$$

Replacing the value of t' in the system in Equation C.40 we get

$$x^{(1)} = \frac{\frac{x^{(0)}}{b} + (N_0' x_1^{(0)} \mod b)N}{b} \mod N \tag{C.42}$$

Once again we can delay the division by b (which means multiplying the numerator by b to remove the denominator of $\frac{x^{(0)}}{b}$ and forgetting about the b in the main fraction) and the modulo N reduction to obtain a value of x with two trailing zeros. The value of x is, in fact,

$$x^{(1)} = x^{(0)} + (N_0' x_1^{(0)} \mod b)Nb^1 \tag{C.43}$$

or, algorithmically,

$$x \leftarrow x + (N'_0 x_1 \mod b) N b^1 \tag{C.44}$$

and $x^{(1)}$ can be written as a vector of $d+2$ elements with the trailing digits set to 0,

$$x^{(1)} = (x_{d+1}, x_d, x_{d-1}, \ldots, x_2, 0, 0) \tag{C.45}$$

Let us now fast forward to the last iteration. This time we want to compute

$$x^{(d)} = \frac{\frac{x^{(d-1)}}{b^{d-1}}}{b} \tag{C.46}$$

or, algorithmically,

$$x \leftarrow \frac{\frac{x}{b^{d-1}}}{b} \tag{C.47}$$

hence we can write

$$x \leftarrow \frac{\frac{x}{b^{d-1}} + t'}{b} \tag{C.48}$$

By writing the usual system of equations, we can compute t' as

$$\begin{cases} t \equiv (-N_0^{-1}) x_{d-1}^{(d-1)} \equiv N'_0 x_{d-1}^{(d-1)} \mod b \\ t' = tN \mod b \end{cases} \tag{C.49}$$

The final value of x can therefore be written as

$$x \leftarrow \frac{\frac{x}{b^{d-1}} + (N'_0 x_{d-1} \mod b) N}{b} \tag{C.50}$$

or

$$x^{(d)} = \frac{\frac{x^{(d-1)}}{b^{d-1}} + (N'_0 x_{d-1}^{(d-1)} \mod b) N}{b} \tag{C.51}$$

For the last time, we can delay the division by d and the reduction modulo N. Since we have delayed every division by b, we can now write the value of x as

$$x \leftarrow x + (N'_0 x_{d-1} \mod b) N b^{d-1} \tag{C.52}$$

Since we haven't reduced x modulo N for d times, the d least significant bits have a value of 0 and x is represented with $2d$ digits, hence as

$$x^{(d)} = (x_{2d}, x_{2d-1}, \ldots, x_d, 0, \ldots, 0) \tag{C.53}$$

Since we haven't reduced x for d times and the last d digits are set to 0, we can apply the division by b^d by simply doing a right shift of d digits. Similarly to the single precision case, we need to check if the result is bigger than the modulus, and eventually subtract N once. Algorithm 18 shows the Montgomery reduction algorithm as we have just seen.

Multi-precision Montgomery multiplication

Now that we know how to do a Montgomery reduction, we can apply it to algorithm 14 (or to whatever multi-precision multiplication algorithm) to compute a multi-precision algorithm. Algorithm 19 shows the resulting algorithm. The multi-precision Montgomery algorithm can be optimised. The resulting algorithm is shown in Algorithm 20.

Algorithm 18 The multi-precision Montgomery reduction algorithm.

Input: $x = x_{d-1}b^{d-1} + \cdots + x_1 b + x_0$
Input: $N = N_{d-1}b^{d-1} + \cdots + N_1 b + N_0$
Input: $R = b^d$
Output: $xR^{-1} \mod N$
procedure MULTIPRECISIONMONTGOMERYREDUCTION(x, N, R)
  for $i \in \{0, \ldots, d-1\}$ **do**
    $t \leftarrow N'_0 x_i \mod b$
    $x \leftarrow x + tNb^i$
  end for
  $x \leftarrow \frac{x}{b^n}$
  if $x \geq N$ **then**
    $x \leftarrow x - N$
  end if
  return x
end procedure

Algorithm 19 The multi-precision Montgomery multiplication algorithm.

Input: $A = A_{d-1}b^{d-1} + \cdots + A_1 b + A_0 \in \tilde{\mathbb{Z}}_N$
Input: $B = B_{d-1}b^{d-1} + \cdots + B_1 b + B_0 \in \tilde{\mathbb{Z}}_N$
Input: $N = N_{d-1}b^{d-1} + \cdots + N_1 b + N_0$
Input: $R = b^d$
Input: $N'_0 = (-N^{-1} \mod R) \mod b$
Output: $ABR^{-1} \mod N$
procedure MULTIPRECISIONMONTGOMERYMULTIPLICATION(A, B, N)
  $x \leftarrow 0$
  for $i \in \{0, \ldots, d-1\}$ **do**
    $x \leftarrow x + A_i B b^i$
    $t \leftarrow N'_0 x_i \mod b$
    $x \leftarrow x + tNb^i$
  end for
  $x \leftarrow \frac{x}{b^d}$
  if $x \geq N$ **then**
    $x \leftarrow x - N$
  end if
  return x
end procedure

Algorithm 20 The optimised multi-precision Montgomery multiplication algorithm.

Input: $A = A_{d-1}b^{d-1} + \cdots + A_1 b + A_0 \in \tilde{\mathbb{Z}}_N$
Input: $B = B_{d-1}b^{d-1} + \cdots + B_1 b + B_0 \in \tilde{\mathbb{Z}}_N$
Input: $N = N_{d-1}b^{d-1} + \cdots + N_1 b + N_0$
Input: $R = b^d$
Input: $N'_0 = (-N^{-1} \mod R) \mod b$
Output: $ABR^{-1} \mod N$
procedure MultiPrecisionMontgomeryMultiplication($\tilde{x}$, $\tilde{y}$, N)
    $x \leftarrow 0$
    for $i \in \{0, \ldots, d-1\}$ **do**
        $x \leftarrow x + A_i B$
        $t \leftarrow N'_0 x_0 \mod b$
        $x \leftarrow \frac{x+tN}{b}$
    end for
    if $x \geq N$ **then**
        $x \leftarrow x - N$
    end if
    return x
end procedure

C.3.3 Radix-2 Montgomery multiplication

The multi-precision Montgomery multiplication is very useful for computing the exponentiation of a number. An example is the RSA cryptosystem which requires rising a message (or a ciphertext) to an exponent e or d. The main component for computing the power of a number is the algorithm for computing the product between two numbers. When considering the Montgomery multiplication algorithm, if the modulus N is odd (like in the RSA example, since N is the product of two prime numbers, which are in turn odd), then we know in advance that $N'_0 \equiv 1 \mod 2$. Thanks to this result and

- reserving an extra digit (bit) for the storage of the employed variables, and
- reserving a further couple of bits in order to postpone the final test ($x \geq N$) as late as possible,

we obtain the radix-2 Montgomery multiplication algorithm (Algorithm 21). This algorithm can be used to efficiently implement an exponentiation algorithm as shown in Algorithm 22.

Algorithm 21 The Radix-2 Montgomery multiplication algorithm.

Input: $A = A_{d-1}b^{d-1} + \cdots + A_1 b + A_0 \in \tilde{Z}_N, \quad A_i \in \mathbb{Z}_2$
Input: $B = B_{d-1}b^{d-1} + \cdots + B_1 b + B_0 \in \tilde{Z}_N, \quad B_i \in \mathbb{Z}_2$
Input: $N = N_{d-1}b^{d-1} + \cdots + N_1 b + N_0, \quad N_i \in \mathbb{Z}_2, \quad N$ odd
Input: $R = 2^{d+2}$
Output: $ABR^{-1} \mod N$
procedure RADIX2MONTGOMERYMULTIPLICATION(A, B, N)
    $x \leftarrow 0$
    for $i \leftarrow 0$ **to** $d + 2$ **do**
        if $x_0 = 1$ **then**
            $x \leftarrow x + N$
        end if
        $x \leftarrow \frac{x}{2} + A_i B$
    end for
    return x
end procedure

Algorithm 22 The RSA encryption algorithm implemented with the optimised multi-precision Montgomery multiplication (the same algorithm can be used to sign, decode and verify a signature) and the left-to-right square and multiply strategy.

Input: m
Input: e
Input: n
Input: $c \equiv m^e \mod n$
Output: $ABR^{-1} \mod n$
procedure RSAENCRYPTION(e, m, n)
    $t \leftarrow \lceil \log_2 e \rceil$ ▷ The number of bits used to encode the exponent
    $R \leftarrow 2^{\lceil \log_2 n \rceil}$ ▷ The radix can be computed as the power of 2 closest to n
    $S \leftarrow R^2 \mod n$
    $\tilde{m} \leftarrow$ RADIX2MMULT(m, S, n) ▷ Compute $\tilde{m} = \mu(m)$
    $\tilde{c} \leftarrow \tilde{m}$
    for $i \leftarrow t - 2$ **downto** 0 **do**
        $\tilde{c} \leftarrow$ MMULT$(\tilde{c}, \tilde{c}, n)$
        if $e_i = 1$ **then**
            $\tilde{c} \leftarrow$ MMULT$(\tilde{c}, \tilde{m}, n)$
        end if
    end for
    $\tilde{c} \leftarrow$ MMULT$(\tilde{c}, 1, n)$ ▷ $\tilde{c} \times 1 \equiv \tilde{c} \cdot 1 \cdot R^{-1} \equiv \tilde{c} \cdot R^{-1} \equiv \tilde{c} \cdot R' \equiv \mu^{-1}(\tilde{c}) \mod n$
    return c
end procedure

Definitions

Theorems Lemmas and principles