Foundations of Cryptography

Lecture Notes

Alberto Boffi

Contents

1 What is Cryptography?

1.1 General Definition

Definition 1. Cryptography is the study of techniques to allow secure communication and data storage in the presence of attackers.

If we want to define cryptography in a more formal and precise way, we have to refer to the *information theory*, credited by Claude Shannon:

Definition 2. A **communication** is a flow of information over a **channel** that takes place between two endpoints: a **sender**, made of an information source and an encoder, and a **receiver**, made of an information destination and a decoder.

As convention, the two endpoints are usually called "Alice" and "Bob".
The "information" can be modeled as strings of symbols belonging to a **finite alphabet**. Being in the computer science discipline, this alphabet is usually $\Sigma = \{0, 1\}$.
The ordinary readable string containing the information the sender wants to transmit is called **message** (or **plaintext**), while the encoded version of the message that is actually transmitted over the channel is the **ciphertext**.

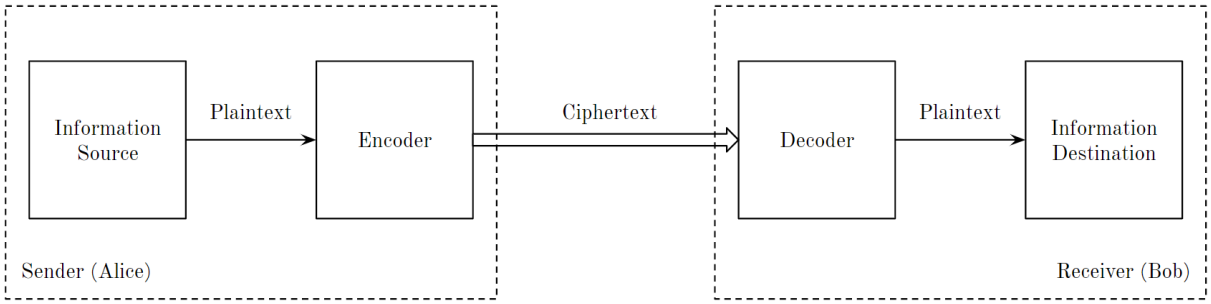


Figure 1: Communication Scheme

Definition 3. A **cryptosystem** is a tuple (P, C, K, E, D), where:

- The **plaintext space** $P = \{0, 1\}^l$ is the set of all possible **plaintexts** ptx;
- The **ciphertext space** $C = \{0, 1\}^{l'}$ is the set of all possible **ciphertexts** ctx. Please note that, in general, $l \neq l'$, i.e. the length of the plaintext does *not* necessarily have to coincide with that of the ciphertext;
- The **key space** $K = \{0, 1\}^\lambda$ is the set of all possible **keys** k;
- The **encryption function** E is a function defined as $E : P \times K \to C$;
- The **decryption function** D is a function defined as $D : C \times K \to P$.

Definition 4. A **cipher** is an implementation of the encryption function E and decryption function D such that, for every plaintext $ptx \in P$, $\exists k, k' \in K$ such that $D(E(ptx, k), k') = ptx$. In case $k = k'$ the cipher is called **symmetric cipher**, if instead $k \neq k'$ **asymmetric cipher**.

A *cipher* includes therefore an encryption algorithm that, given a key and a plaintext, produces a ciphertext, and a decryption algorithm that, given another key and any ciphertext produced during the encryption, returns the original plaintext. As we'll see, the functions E and D can also coincide.

1.2 Security Requirements

Before diving into the study of the different kinds of ciphers, we need to know how to evaluate them, that is, what security requirements the ciphers must meet.

Assumption 1. *An ideal attacker has complete knowledge of the cryptosystem in use.*

This means that the attacker can access the plaintext, ciphertext and key space, and call the functions E and D as many times as he/she wants. Therefore, contrary to what some may think, hiding these elements cannot be considered a security requirement.
We will understand the important of this assumption later on, and the reason why we can safely make it lies in Kerckhoff's second principle for a good cipher:

Proposition 1 (Kerckhoffs's Principle)**.** *A cipher must be secure even if the cryptosystem is known to the enemy.*

This means that the only thing that the attacker doesn't know is the specific plaintext the sender is communicating during a particular communication, or the specific keys in use.

The main features a cipher can offer are the following. Depending on the domain of application, we may be interested in some of them with respect to the others:

- **Confidentiality**: The plaintext cannot be derived from the ciphertext without having the decryption key;
- **Integrity**: The information transmitted cannot be tampered with along the channel;
- **Authenticity**: The identity of the sender is guaranteed to the receiver.

2 Symmetric Cipher

As previously discussed, a symmetric cipher is a cipher in which the required encryption and decryption keys are the same.

2.1 Confidentiality

The first constraint we will analyze is also the most popular and discussed when talking about cryptography: *confidentiality*.
To be able to correctly judge the level of confidentiality of our cipher, the first thing to do is to consider ourselves inside a framework where we assume to be under some kind of attack. In particular, we'll consider the most basic model of an attack, called **ciphertext-only attack**. In a ciphertext-only attack, the attacker can simply intercept and read the ciphertext transmitted over the channel.

2.1.1 Perfect Cipher

Definition 5. Let ctx_{sent} be the ciphertext transmitted over the channel, and ptx_{sent} the corresponding plaintext. A cipher is said to be **perfect** iff $\forall ptx \in P$ and $\forall ctx \in C$ we have $P(ptx_{sent} = ptx) = P(ptx_{sent} = ptx | ctx_{sent} = ctx)$.

In other terms, after eavesdropping the communication, the attacker has not any additional information with respect to what he previously knew.
But this definition is not constructive. If a perfect cipher actually exists, how we can build one? Let's first look at a characteristic feature of a perfect cipher:

Theorem 1 (Shannon 1949)**.** *If a cipher with $|P| = |C|$ is perfect, then $|K| \geq |P|$.*

We can intuitively look at this theorem as follows.
We are considering the case where the number of plaintexts is the same as the number of ciphertexts: for instance, suppose $P = \{ptx_1, ptx_2, ptx_3\}$ and $C = \{ctx_1, ctx_2, ctx_3\}$.
If the number of keys is enough, we can theoretically map every plaintext in each possible ciphertext, depending on the key used. If instead the number of keys is not enough, some ciphertexts will not be mapped by every possible plaintext. This means that every time one of these ciphertexts is transmitted, the attacker will get more information than he had before. To understand it better, let's continue with the same example, and suppose that the key space has only two keys: $K = \{k_1, k_2\}$. We can define a possible encryption function as follow: $E(ptx_1, k_1) = ctx_1$, $E(ptx_1, k_2) = ctx_2$, $E(ptx_2, k_1) = ctx_2$, $E(ptx_2, k_2) = ctx_3$, $E(ptx_3, k_1) = ctx_2$, $E(ptx_3, k_2) = ctx_3$.

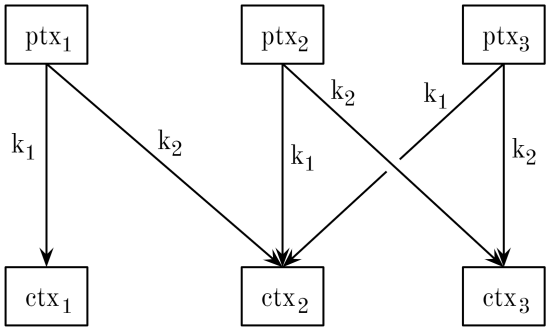


Figure 2: Example of a Non-Perfect Cipher

Remember that this mapping is known to the attacker, since, for the assumption stated before, the cryptosystem is fully known to him.
For the sake of simplicity, let's just focus on ptx_1, but the same reasoning can be applied to each plaintext: before the transmission the attacker can state that $P(ptx_{sent} = ptx_1) = P(ptx_{sent} = ptx_2) = P(ptx_{sent} = ptx_3) = \frac{1}{3}$, as he has no idea which message will be transmitted. Let's now suppose that ctx_3 is transmitted other the channel. At this point, since there is no key mapping ptx_1 to ctx_3, the attacker knows that $ptx_{sent} \neq ptx_1$. He can consequently tell that $P(ptx_{sent} = ptx_1 | ctx_{sent} = ctx_3) = 0 \neq P(ptx_{sent} = ptx_1)$. Therefore, we can state that the cipher is *not* perfect.

Moreover, we can observe that the hypothesis of the theorem about $|P| = |C|$ is not so far-fetched: intuitively, in the encryption, to be sure that the same key does not map multiple plaintexts in the same ciphertext, we need at least as many ciphertexts as the number of plaintexts, i.e. $|P| \leq |C|$. In the decryption we can apply the opposite reasoning: to be sure that the same decryption key does not map multiple ciphertexts in the same plaintext we need $|P| \geq |C|$. So at the end we obtain $|P| = |C|$.

The only real example of secure cipher we have today is the **Vernam cipher**, also called **one-time pad** (**OTP**).
In its most popular form, it considers $P = C = \{0,1\}^l, K = \{0,1\}^\lambda$. The encryption function E performs a bitwise XOR between the plaintext and the key to obtain the ciphertext. The decryption function D is exactly the same function as E: by performing a bitwise XOR between the ciphertext and the key, it returns the original plaintext. The key is called "**pad**". More formally:

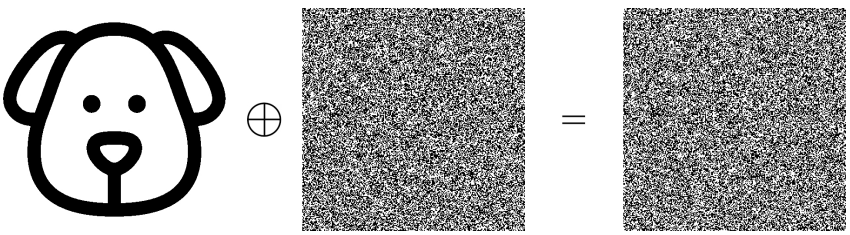
$E(ptx, k) = ptx \oplus k, D(ctx, k) = ctx \oplus k.$

However, keep in mind that this is not the only possible implementation of a OTP cipher. Another, conceptually equivalent variant, is the one in which the encryption function performs a modulo l addition between each character of the plaintext and the corresponding character of the key. In other terms, it works on the additive group $\mathbb{Z}_l$.

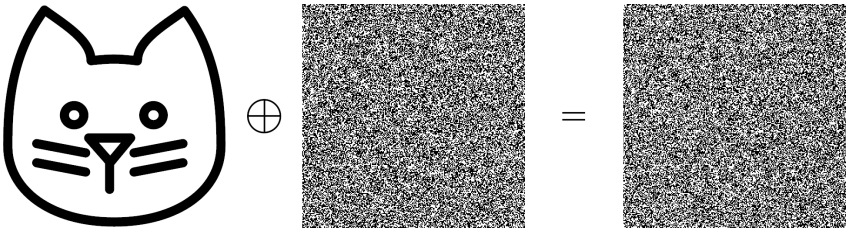
The robustness of OTP is due to some strict assumptions:

1. The pad must have at least the same length l as the plaintext;
2. The pad must be chosen randomly, i.e. it must be uniformly distributed in the set of all possible keys;
3. The pad must never be reused (hence the name *one-time* pad).

About point 3, the best way to understand the danger of key-reusing is by visualizing it graphically. Suppose we want to use the OTP cipher on images, and we encrypt the dog image with a certain key:



Let's now encrypt another image (the cat image) with the same key:



The two results, taken individually, appear pretty random. But when we XOR them together, we get the XOR of the two original plaintexts, and this is what it looks like:

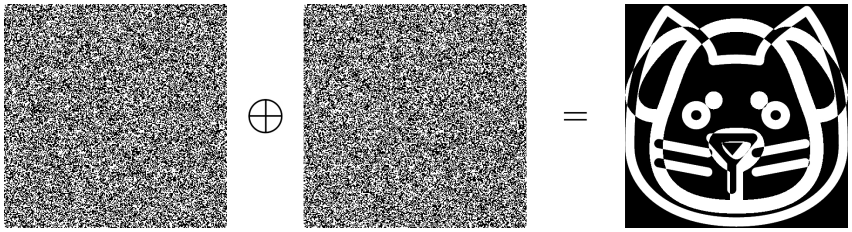


Figure 3: Danger of Key Reuse in OTP Cipher

It's therefore evident that the attacker can get information about every plaintext that has been encrypted with the same key.
To test it yourself: https://t.ly/7YJX.

If these three assumptions are met, there is no way an attacker can get information about the plaintext by observing the ciphertext. The only information he could infer is the length of the plaintext (since it is the same of the length of the ciphertext), but this is already a public information of the cryptosystem. We can therefore state:

Theorem 2. *OTP is not vulnerable to brute force attacks.*

In fact, by brute forcing all the possible keys, an attacker would get all the possible plaintexts, but the plaintext space is an information he already had before.

This cipher is still used in government scope, but, unfortunately, for our purposes is impractical. The reasons why actually lie in the assumptions stated before:

1. The first problem is to communicate the key to the receiver in a secure way, and in the case of OTP this means to communicate a message at least as long as the plaintext. The advantage in communicating the key, instead of directly communicate the plaintext, is that in this way we can send a unique long bitstream that can be used to extract multiple keys, until the sum of the length of the messages to sent reaches the length of the bitstream. But this process makes the data to communicate even bigger. Nowadays, where OTP is used, the key is communicated in person, even if this is still a method sensitive to corruption or theft. But this is obviously not an option for our daily purposes;

2. Generating a key that is completely random is not an easy task at all. True-random generators exist, but they are harder to implement and slower than pseudorandom generators;

3. We need as many keys as the possible messages that can be sent from here on, theoretically forever. In practical applications, some key reuse problems have arisen.

Since building a OTP cipher is not practical, we have to leave the ambition of a perfectly secure cipher. The idea is to go towards something more feasible to implement, that, even if not perfect in theory, is unbreakable in practice.

2.1.2 Computationally-Secure Cipher

We have seen that, in a perfect cipher, the attacker cannot know the plaintext associated to a ciphertext without knowing the key. But what if the plaintext can be actually obtained, yet the process of doing this is extremely computationally difficult?

Proposition 2 (Nash)**.** *A* ***secure cipher*** *is a cipher that is* ***computational hard to break****.*

The statement refers to NP problems and it is telling that, in order for a cipher to be secure, decrypting the message without knowing the key should be reduced to solving a NP problem.
Note that this concept is based on the non-proved assumption that $P \neq NP$. If one day it will be proved that $P = NP$, every concept described from now on will immediately collapse, and modern cryptography will have to be entirely thrown away.

But how we can use this definition to our advantage?

We have seen that the problems with the implementation of a perfect cipher derive from the generation and distribution of long keys. We can therefore think to manage a communication as follows:
Basic Idea: Alice and Bob exchange a key that is shorter than the plaintext. At this point, both Alice and the Bob use the same deterministic function to extend the length of key, in order to reach the length of the plaintext. With the same key, Alice and Bob can now communicate using a OTP protocol.

The function used to stretch the key goes under a particular name in the literature:

Definition 6. A **Pseudorandom Number Generator** (**PRNG**) is a *deterministic* function $PRNG : \{0,1\}^{\lambda} \to \{0,1\}^{\lambda+n}$, where n is called PRNG **stretch**.

The problem now is that, since the exchanged key is shorter than the plaintext, the cipher isn't perfect anymore. However, if the PRNG is hard to break, using Proposition 2 we can say that the cipher is by definition secure. In particular, still denoting with λ the length of the original exchanged key, and assuming that the attacker can only do computations with polynomial complexity with respect to λ:

Definition 7. A **Cryptographically Safe Pseudorandom Number Generator** (**CSPRNG**) is a PRNG whose output cannot be distinguished from an uniform random sampling of $\{0,1\}^{\lambda+n}$ in $O(poly(\lambda))$.

Although possible, building and testing a CSPRNG from scratch is neither efficient nor practical. The last step is therefore to figure out how to practically implement it. Actually, a CSPRNG is built by means of a particular building block, called *block cipher*. Let's first define and analyze a block cipher, then we'll move to describing how it is used to build a CSPRNG.

Definition 8. A **block cipher** is an encryption algorithm that takes as input a key k and a string ptx of a **fixed length** b (called **block**), and returns an output of the same length. Given k, the encryption function $E(ptx, k)$ corresponds to one of the possible permutations over the set of the 2^b possible input blocks.

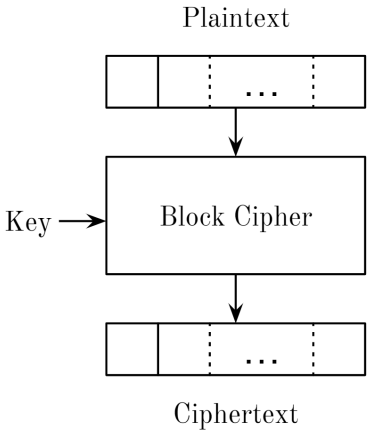


Figure 4: Block Cipher

When dealing with cryptography through a theoretical prospective, a block cipher usually goes under the name of **Pseudorandom Permutation** (**PRP**), but both terms mean exactly the same thing.

Definition 9. A block cipher that accepts keys of length λ is said to be **broken** if, without knowing the key, the plaintext can be derived from the ciphertext with less than 2^λ operations.

This means that in an unbroken block cipher the only way to derive the plaintext is to brute force the key, which costs $T(\lambda) = O(2^\lambda)$.

There are some examples of widespread block ciphers. In the 1970s the standard was the *Data Encryption Algorithm* (*DEA*, aka *DES*), but the keys were only long 56 bits, and nowadays it can be broken in a reasonable amount of time. A variant is *Triple DES*, that we can superficially see as the same algorithm but with a three-times recursive encryption. *Triple DES* can still be found in some legacy systems, but it's officially deprecated.
The standard algorithm is now **Advanced Encryption Standard** (**AES**). In *AES*, the block has a length of 128 bits, and the key can be 128, 192 or 256 bits long.
Keep in mind that the time required to brute-force a cipher that uses a 256-bits key is not even astronomically quantifiable, and also a 128-bits lengths can be considered more than enough.

Obviously, it's rare that the message we want to encrypt is exactly of the same length of the block. In particular, the case where $len(plaintext) < len(block)$ doesn't cause any problem, since it's sufficient to extend the plaintext with a sequence of fixed bits ("padding"), until we reach the length of the block. But what most of the time happens is that $len(plaintext) > len(block)$, and so the block cipher is always used in conjunction with some mechanism to overcome this problem, that comes under different names:

Electronic CodeBook (ECB)
It's the most intuitive method, and it's based on the principle of *split-and-encrypt*: we split the plaintext in multiple segments and we encrypt each segment with the same key.

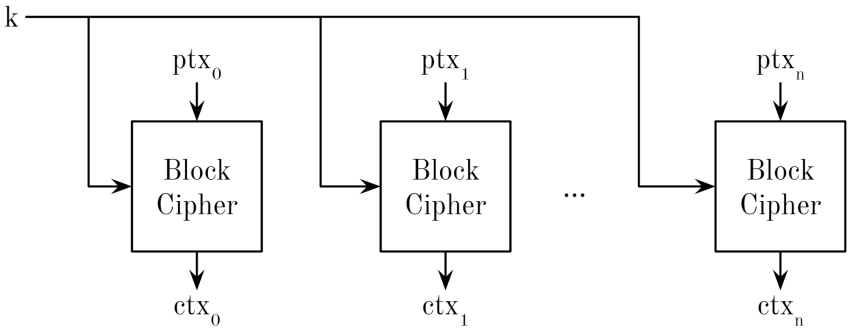


Figure 5: ECB Schema

It's fast, easy to implement and even suitable for parallelization. The decryption is straight forward as well.
The problem arises when in the original message there are repetitions, and two or more blocks come out to be the same: in this case, even the corresponding outputs will coincide. Therefore, even if the attacker does not know the corresponding plaintext, he knows that such parts are repeated, and that's a huge and really dangerous information. This is usually shown with the "ECB penguin" example: the one on the left is the original image, while on the right its encrypted version computed in ECB mode. Due to lot of repeated sections in the original image, the result it's pretty bad in terms of confidentiality:

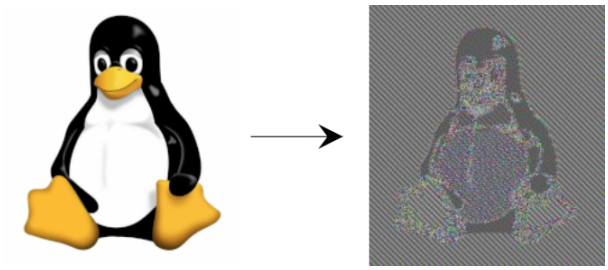


Figure 6: ECB Encryption

Counter (CTR) Mode

To overcome the drawback of the Electronic CodeBook, this time we use a counter, made of numbers of fixed length, sequentially increased. We encrypt each value of the counter and then we XOR it with each segment of the plaintext. This process will return an output that it's always different.
The starting number of the counter is indifferent, and we can also decide to use an increment different than 1, but 1 is secure enough and results in a more efficient computation.

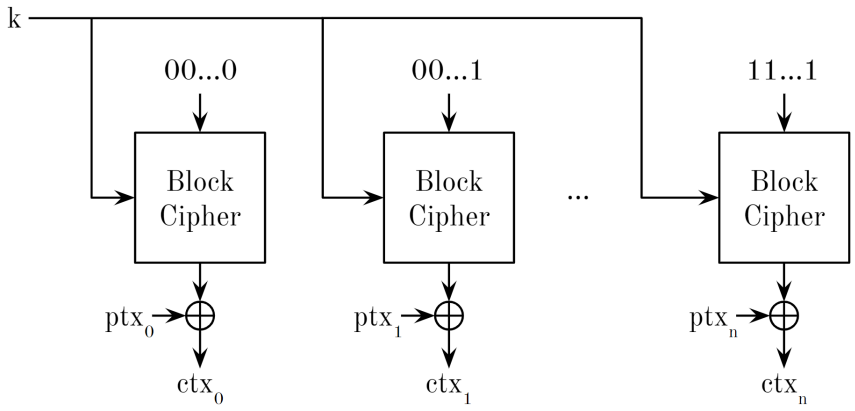


Figure 7: CTR Mode Schema

We can also repeat the experiment of the encryption of the penguin image, using now the CTR mode. The difference in the result is pretty evident:

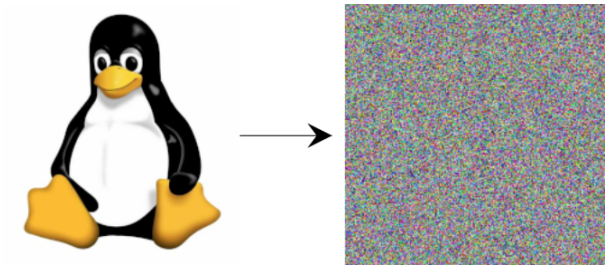


Figure 8: CTR Mode Encryption

The CTR mode is actually the definitive implementation of our original idea about the construction of a computationally-secure cipher. In fact, in its schema we can actually spot a PRNG (highlighted in the figure below), that stretches the key received as input. If the block ciphers are not broken, the PRNG is also a CSPRNG. The concatenation of the outputs of each block cipher is the output string of the CSPRNG, which is then XORed with the plaintext in order to obtain the ciphertext, as in the OTP cipher.

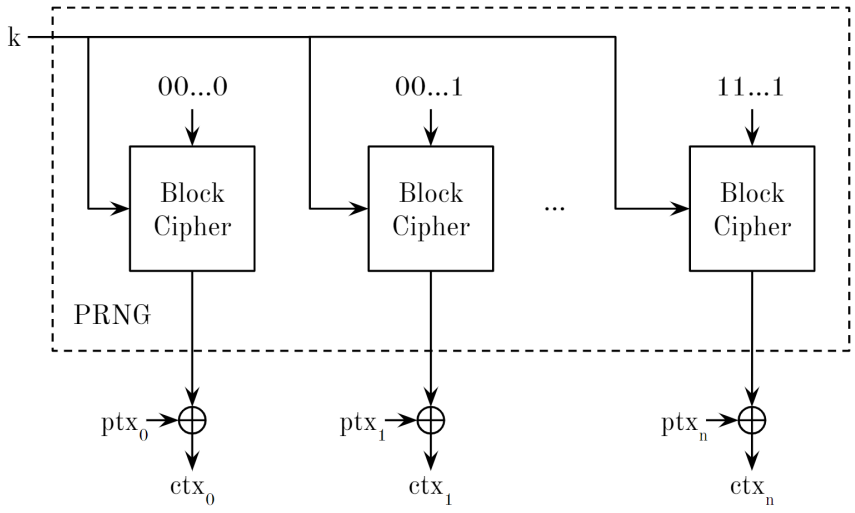


Figure 9: Definitive Implementation of a Computationally-Secure Cipher

2.1.3 Practical Implementation of a Block Cipher

The algorithms behind the commonly used block ciphers will not be covered, but it is important to discuss the process that leads to their creation.
As we have seen, to prove that a PRNG is a CSPRNG we should prove that the block cipher employed is not broken, i.e. that deriving the plaintext or the key has at least a cost $T(\lambda) = O(2^\lambda)$. However, unfortunately, we are not (yet) able to do this. In fact, proving the exponential lower bound complexity of an algorithm would imply proving that $P \neq NP$. The only thing we can do is to prove that a PRNG is *not* a CSPRNG, in case we are able to derive the plaintext with less than 2^λ operations.

For this reason, the commonly used block ciphers are the result of public contests, where cryptographers propose their algorithms and other cryptographers try to break them.
At the beginning, introducing cryptography, we said that a public cryptosystem must not compromise security. We see now that a public cryptosystem actually *improves* security, since it means more revision, more testing and so more guarantees of robustness.

2.1.4 Note on Known-Plaintext Attacks

Since now we have analyzed confidentiality under the hypothesis of a ciphertext-only attack. But a ciphertext-only attack is not the only kind of attack that can compromise the constraint of confidentiality. Another possible threat is the so-called **known-plaintext attack**. In a known-plaintext attack, the attacker knows at least a sample of couples (ptx, ctx), where ctx is the encryption of ptx. A popular example of this kind of attacker was Alan Turing when, in the early 40s, he discovered repetitive messages in the Enigma code corresponding to the greeting to Adolf Hitler.

A limit case of a known-plaintext attack is the **chosen-plaintext attack**: in this case the attacker actually chooses the plaintext, and can consequently see the corresponding ciphertext. It's a rarer condition. We can anyway just focus on general known-plaintext attacks, as chosen-plaintext attacks are just a proper subset.

To analyze the situation, we consider the extreme case, where $P = \{ptx_1, ptx_2\}$. At this point, if the encryption function is deterministic (as the ones seen so far), the cipher is not secure. In fact, even if the attacker knows only a pair (ptx_1, ctx_1), he will be able to understand every message exchanged: if he reads ctx_1, he knows that the original message is ptx_1, vice versa ptx_2.
This means that, in order for the cipher to be secure, the encryption function must be non-deterministic. Actually, there exists different paths we can follow to reach this goal:

A first, simple way is to manipulate the key every time it is used. Another common method is to choose each time a different number, called **nonce**, as starting point of the counter. But the most popular way to do it is probably the technique of *rekeying*, employed through what goes under the name of **symmetric ratcheting**.
This method consists in producing a different key for every message through the use of a PRNG, that

takes an input string of length λ and stretches it into an output of length 2λ: the first half of the output is used as a key, while the second half is used as input to the same function to produce another key, and so on. The initial input used to generate all the keys from there on is called **seed**.

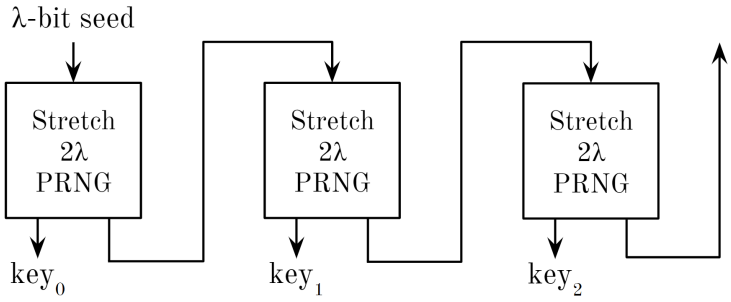


Figure 10: Symmetric Ratcheting

The terms "ratcheting" derives from the fact that, as a ratchet can only turn on one side but cannot turn back, this mechanism ensures that, even finding a key, there's no way to know the previous ones.

2.2 Integrity

We move now from confidentiality to the constraint of *integrity*. Please note that, as strange as it may seem talking about cryptography, now we are not interested anymore in hiding the message to transmit, but just in guaranteeing that the content of the message can't be manipulated along the channel.

As we did for confidentiality, we need to model the attack framework that we assume is in place. In this case, we consider a **bit-flipping attack** (or **active attack**), in which the attacker can intercept and tamper with the data.
To understand the danger of this kind of attack, try to imagine a bank transaction: by simply changing some bits in the ciphertext, once decrypted, the message can contain a different amount of money with respect to what was originally sent.

Basic Idea: To provide integrity, we can append to the message a tag that the attacker is not able to forge, which can guarantee that the message has not been tampered with. This tag is also called **Message Authentication Code** (**MAC**)[1].
From a high level perspective, we can see this idea as a set of two functions, the first one used by the sender, and the second one by the receiver:

- $compute\_MAC(string, key) = tag$
- $verify\_MAC(string, tag, key) = \{true|false\}$

In practice, we can implement this concept using different approaches.

CBC-MAC
The idea behind a CBC-MAC basically consists in using as tag an encrypted version of the message. But since now, as we will see, the tag will not be decrypted by the receiver, we can use the block ciphers in a different way with respect to the CTR mode, in order to produce a shorter output string.
In particular, the block ciphers are connected in cascade in such a way that the input of each one is the output of the previous block XORed with the corresponding section of the message (the first section can be simply XORed with a 0).
In case the message has a fixed prefix, the tag has to be encrypted one more time for the technique to be secure.

[1] Please note that the name is misleading: the Message *Authentication* Code provides integrity, not authenticity.

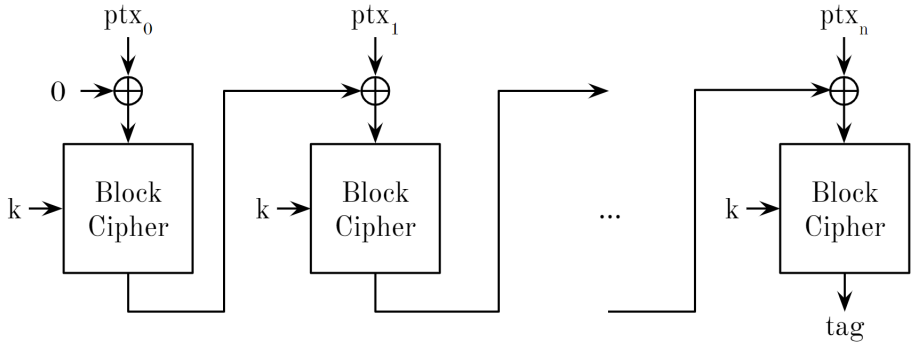


Figure 11: CBC-MAC Computation

At this point, we can describe the communication flow as follows:

- Alice, given the message msg and the key k, calculates the CBC-MAC tag and sends it to Bob together with msg;
- Bob, using the message received by Alice and the key k, calculates the CBC-MAC from his side;
- Bob compares his CBC-MAC with the one sent by Alice. If they are the same, the message has not been tampered with.

This method works since any attempt from the attacker to forge the message would cause a change even in the CBC-MAC calculated by Bob. To forge the message, the attacker should also change the CBC-MAC accordingly, but he cannot do it without the key.

The problem with this technique is the efficiency: calculating the CBC-MAC tag every time a message is sent is expensive and slow. We land therefore on the next method.

Cryptographic Hash Functions

Definition 10. A **Cryptographic Hash** is a function $H : \Sigma^* \to \Sigma^n$ for which the following problems are computationally hard:

1. **Resistance to first preimage attack**: Given $d = H(s)$, find s (i.e., invert H).
   In the ideal case, the computation required is $T(n) = O(2^n)$;
2. **Resistance to second preimage attack**: Given s and $d = H(s)$, find $r \neq s$ such that $H(r) = d$.
   In the ideal case, the computation required is $T(n) = O(2^n)$;
3. **Resistance to collision**: Find r, s with $r \neq s$ such that $H(s) = H(r)$.
   In the ideal case, the computation required is $T(n) = O(2^{\frac{n}{2}})$.

The output of the function is called **hash** or **digest**.

Working with bits, in our case a cryptographic hash function can be more explicitly defined as $H : \{0,1\}^* \to \{0,1\}^n$. Therefore, what it does is taking an arbitrary-length-input and convert it into a fixed-length-output. This means that the function is not injective: there will necessarily be multiple input being mapped into the same output. It's however not possible to compress the input indefinitely, as there always exists a minimum number of bits, called **entropy**, that can be used to represent a certain information. But not every function of this kind is a cryptographic hash function, as they must respect the three requirements. For example, a simple function that just takes the first n bits of the input string is not resistant to both second preimage attack and collision.

The second preimage attack can be reduced to the first preimage attack, and this means that if H does not satisfy the first property, it will necessarily not satisfy the second too. The opposite is, in general, not true. This also explains why, in the ideal case, the time complexity of the two attacks is the same. The ideal complexity required to find a collision is less because the computation can be speeded up by exploiting the math behind the probabilistic birthday paradox.

Please note that the first two properties are what, commonly speaking, distinguishes a *cryptographic hash function* from a simple *hash function*, which just tries to avoid conflicts.

As discussed talking about block ciphers, we cannot prove in theory that a function is a cryptographic hash function, since we cannot prove the exponential lower bound complexity of an algorithm. So, even in this case, these functions are the result of public contests.
Old cryptographic hash functions as *SHA-1* and *MD5* has proven to be broken. Nowadays, we can confidently use *SHA-2* and *SHA-3*, that can produce digests of 256, 384 or 512 bits. *SHA-3* has been produced because *SHA-2* actually shares some mechanism with functions that have been proven to be broken, but both *SHA-2* and *SHA-3* are currently unbroken and widely standardized.

Cryptographic hash functions are highly employed in computer security, for example to avoid storing passwords and other important data in clear text, or in digital forensics to obtain a "fingerprint" of the acquired disk. For our purposes, we need cryptographic hash functions to compute a MAC tag that is easy to build and verify.
Contrary to what we have done for the CBC-MAC, now Alice cannot simply use $h(msg)$ as MAC tag to send with the message msg, since it would be completely useless. In fact, given that the hash is produced without using a key, the attacker can simply forge the message and change the tag accordingly.
So, given the key k that Alice and Bob are using, the problem can be solved by using $h(msg + k)$ as a tag.
We can therefore provide a tag that can guarantee integrity, and that can be computed much more efficiently with respect to the CBC-MAC.

It's worth pointing out that this is just the general idea behind how cryptographic hash functions are used to ensure integrity. In reality, the mechanism is more complex and sophisticated, and goes by the name of **HMAC**.

2.3 Authenticity

In this scenario, the attacker is communicating with the receiver by pretending to be someone else. This means that, differently from confidentiality and integrity, there is actually no ongoing attack to model here - we're just considering standard communication, with the only difference being that the receiver was expecting a message from another sender.

Unfortunately, a symmetric cipher cannot be used to provide authenticity, so we can't discuss this property.

3 Asymmetric Cipher

3.1 Background

An **asymmetric cipher** (or **public key cipher**) is a cipher where the encryption and decryption keys are different.

Asymmetric ciphers were born to solve two main problems related to symmetric ciphers:

1. Symmetric ciphers cannot provide authenticity;
2. The security of a symmetric cipher is based on the precondition that the key has been previously exchanged in a secure way. But there is no way to do it, other than encrypting the key itself, which would recursively raise the same problem of exchanging another key.

The cons of asymmetric ciphers is the efficiency: they are from 10x to 1000x slower than symmetric encryption.

In an asymmetric cipher, each one of the two endpoints has two keys, called **public key** and **private key** (or **secret key**), where the public key is built starting from the private key: $k_{pub} = f(k_{pri})$.
As the name suggests, the public key can be visible to everyone, while the private key is only known by the respective endpoint. The cipher is based on three main features:

- If the message has been encrypted with the private (public) key of a certain endpoint, it can be only decrypted with public (private) key of the same endpoint;
- If the message has been encrypted with the public key, decrypt it without knowing the private key must be computationally hard;
- Compute the private key given the public key (i.e. compute $f^{-1}(k_{pub})$) must be computationally hard.

To give an example, a private key can be the set of two large prime numbers p and q, while the public key the product $p \cdot q$. At this point, finding the prime factors that make up the public key means solving a famous NP problem known as *prime factorization*.

As for symmetric ciphers and cryptographic hash functions, since we cannot prove the exponential lower bound complexity of an algorithm, asymmetric ciphers are the outcome of public contests.

3.2 Confidentiality

Asymmetric encryption can be used to provide confidentiality against a ciphertext-only attack as follows:

- Bob (receiver) generates a keypair (k_{pri}, k_{pub}). He sends k_{pub} to Alice;
- Alice (sender) encrypts the plaintext with Bob's public key: $ctx = E(ptx, k_{pub})$. She sends it to Bob;
- Bob decrypts the ciphertext using his private key: $ptx = D(ctx, k_{pr})$.

Specularly, if Alice in turn generates another pair of private and public key, Bob can send messages to Alice too, producing a secure bi-directional communication.
However, since asymmetric encryption is so much slower than symmetric encryption, in practice an asymmetric cipher is never used to entirely manage a communication. Instead, it is used to do what is called **key encapsulation**, i.e. to securely exchange the key to use from there on with a *symmetric cipher*. Since the key is the only message being exchanged, just a pair (k_{pri}, k_{pub}) for only one of the two endpoints is enough. Moreover, for this reason we don't really need to consider known-plaintext attacks for asymmetric ciphers.

The standard cryptosystem used to provide confidentiality through asymmetric encryption is the **Rivest-Shamir-Adleman**, commonly known as **RSA**, which is actually based on the prime factorization problem.

3.3 Integrity

For the property of integrity, nothing changes with respect to a symmetric cipher: we can still use a MAC tag in the same way.

3.4 Authenticity

With an asymmetric cipher, we can actually provide authenticity. In this scenario, we are not interested anymore in hiding the content of the message, but in guaranteeing the identity of the sender. The way we can do it is by adopting the dual concept of signatures in the digital world:

- Alice (sender) generates a keypair (k_{pri}, k_{pub}). She sends k_{pub} to Bob;
- Alice sends the message msg_A in clear text to Bob;
- Alice encrypts the message with her private key: $ds = E(msg_A, k_{pri})$, where ds is called **digital signature**. She sends the digital signature to Bob;
- Bob (receiver) decrypts the digital signature with Alice's public key: $msg_B = D(ds, k_{pub})$
- Bob compares msg_B with the message msg_A previously sent in clear text by Alice: if they are equal authenticity is guaranteed.

The role of the digital signature is therefore to guarantee that the endpoint who sent the message is the same one that originally transmitted the public key. This mechanism works in theory, but a first problem is that usually we want to provide authenticity in exchanging big documents, and when the message is particularly long the encryption is slow.
We can solve the problem by exploiting the cryptographic hash functions: Alice also sends the hash $h(msg_A)$ of the message, and produces the digital signature by encrypting $h(msg_A)$ instead of msg_A. At this point Bob can decrypt it and compare the result with the hash sent in clear by Alice.
Since the hash has a fixed length regardless of the size of msg_A, we are gaining in efficiency.

Another problem is due to the difference between digital signatures and paper signatures. In the physical world, to sign means to sign the output, while in the digital world it means to sign the code.
To understand this concept, let's imagine that we are signing a digital document. In reality, we are not signing what is written in the document at that specific moment, since the document can contain dynamic content, such as a date that changes according to the current day: when the date updates, the signature remains the same, and this is not the desired behavior.
Actually, there is no real way to solve this problem, other than signing only the types of document that can't change.

As last observation, let's imagine we want to provide confidentiality together with authenticity. In this case, we can combine the mechanisms seen for both properties, in a "matrioska" structure.
We are not going to see this in an explicit way, but in general both Alice and Bob will generate their own (k_{pri}, k_{pub}) pair. Then, Alice will first encrypt the message using her private key, and then she will encrypt the result another time with Bob's public key. Bob will decrypt the ciphertext received with his private key first, and then with Alice's public key.

The RSA standard, other than confidentiality, is built to provide authenticity as well.

3.5 Security Comparison with Symmetric Ciphers

We have seen that the only possible attack to a secure symmetric cipher is brute-forcing the key, which has a cost of $T(\lambda) = O(2^\lambda)$.
Asymmetric ciphers, instead, rely on hard problems for which brute-forcing is *not* the best attack. This implies that break a secure asymmetric cipher requires little less than $O(2^\lambda)$ operations, depending on the particular algorithm employed. Thus:

Proposition 3. *It's not possible to rely on the length of the key to compare the security of an asymmetric cipher with the security of a symmetric cipher.*

We can only use the length of the key to compare two symmetric ciphers. If, instead, we want to compare the security of an asymmetric cipher, we need to know the computational complexity needed to break it.

3.6 Public Key Infrastructure

For the moment, we cannot yet guarantee the full security of an asymmetric cipher. In fact, the asymmetric cipher architecture presented so far is sensitive to another kind of attack, called **Man In The Middle (MITM) attack**. In a MITM attack, the attacker can intercept any message exchanged between Alice and Bob, and send messages in turn. For this reason, he can be (logically) seen as a "man in the middle" of the communication.

A MITM attack can pose a threat to the communication since we have not yet seen a way to "bind" the public key to the endpoint that produced it. To understand this, let us suppose that Alice wants to send her public key $k_{pub(A)}$ to Bob. If the attacker generates his keypair $(k_{pri(E)}, k_{pub(E)})$, he can intercept $k_{pub(A)}$ and send $k_{pub(E)}$ to Bob instead. At this point, Bob will treat $k_{pub(E)}$ believing it to be Alice's key. The attacker can now use his private key $k_{pri(E)}$ to decrypt the messages sent by Bob, violating the confidentiality constraint, and to sign messages towards Bob, violating the authenticity constraint. He can furthermore carry on the communication in both directions, talking to Alice pretending to be Bob, and to Bob pretending to be Alice.

To prevent a MITM attack, we need to apply the same mechanism we just saw for providing authenticity

on the message, but now providing authenticity on the public key that Alice sends to Bob. To do that, we introduce a new endpoint, called **Certificate Authority** (**CA**).
The communication consists of two phases. In the first phase, Alice communicates with the Certificate Authority:

- Alice generates a keypair $(k_{pri(A)}, k_{pub(A)})$. She sends $k_{pub(A)}$ to the CA;
- The CA generates a keypair $(k_{pri(C)}, k_{pub(C)})$. It signs Alice's public key: $ds = E(k_{pub(A)}, k_{pri(C)})$;
- The CA sends to Alice a collection of data containing: the digital signature ds, the subject's public key $k_{pub(A)}$, the CA public key $k_{pub(C)}$ and other details, as the signature algorithm or information about Alice. This collection is called **digital certificate**;

The second phase involves the communication between Alice and Bob. Now, we assume that Bob has a list of public keys owned by CAs he trust:

- Alice sends the digital certificate to Bob;
- Bob verifies that $k_{pub(C)}$ is in his list of trusted public keys. If not, he discards the digital certificate;
- If $k_{pub(C)}$ is trusted, Bob decrypts the digital signature ds with $k_{pub(C)}$: $k = D(ds, k_{pub(C)})$;
- Bob compares k with $k_{pub(A)}$: if they are equal, authenticity on Alice's public key is guaranteed.

This communication scheme as it stands works in theory, but in practice there exists a large number of Certificate Authorities, and Bob can only have a small number of them in his list of trusted public keys. So, whenever he receives a public key from a new legitimate CA, he will discard the digital certificate. We therefore need to reduce the number of CAs to trust.

To do this, the key $k_{pub(C)}$ of the CA is in turn authenticated by another higher-level CA, responsible for signing multiple low-level CA's public keys. For the same reason, the public keys of the high-level CAs are signed by other CAs on a higher level, and so on. The top-level CAs are called **Root CAs**: they are the only CAs that self-sign their certificates, and therefore the only CAs we need to trust.

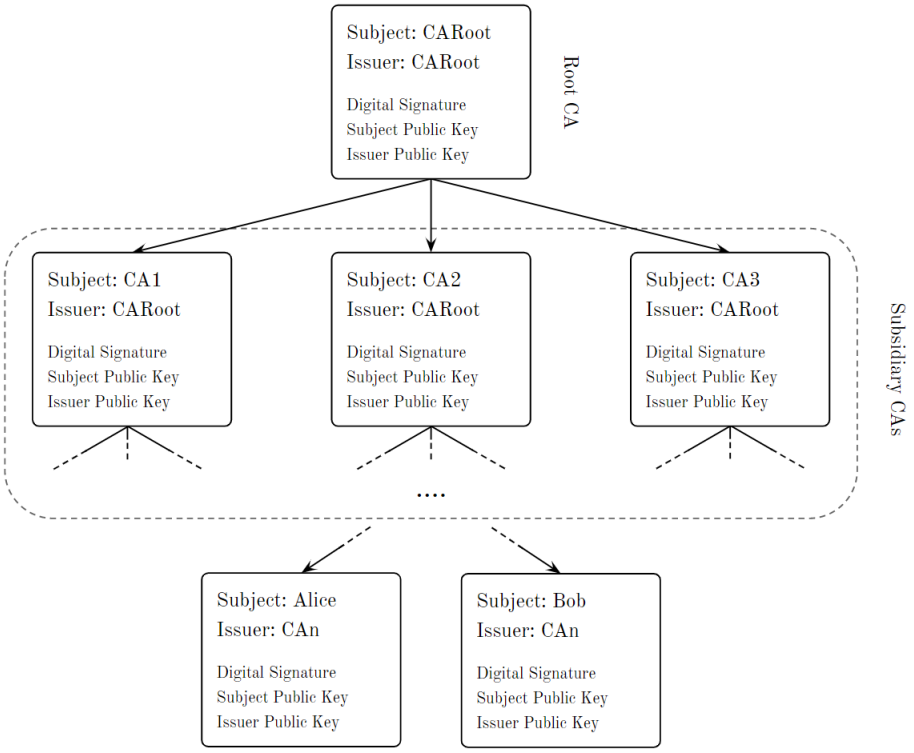


Figure 12: Distribution of Digital Certificates by CAs

All this mechanism of creation and distribution of digital certificates is known as **Public Key Infrastructure** (**PKI**).

4 Open Problems in Cryptography

Despite everything, there are still some open problems in cryptography. Among these we can mention:

- **Homomorphic encryption**: Homomorphic encryption is a form on encryption that allows users to perform computation over encrypted data without decrypting it. It's pretty useful when the user wants to give this data to a third-party as a public cloud infrastructure. We are already able to do it, but in the state-of-the-art the process it's quite slow. The challenge is to find more efficient ways to do it;
- **Quantum computing**: Several NP problems underlying cryptographic algorithms, such as the prime numbers factorization, can be actually quickly solved by quantum computers. Nowadays there exists a research field, called *post-quantum cryptography*, that studies algorithms that are difficult to solve even for a quantum computer;
- **Side-channel information**: Another, different kind of attack that can compromise a secure communication, is the *side-channel attack*. In a side-channel attack, instead of exploiting vulnerabilities in the cipher, the attacker exploits extra information derived from the implementation of the cipher, as power consumption or electromagnetic radiations emitted by the encryption device. The prevention of side-channel attacks is still a much debated and studied topic.