# Cryptography lecture notes

# Contents


**I Introduction to modern cryptography** **5**

**Front page** **7**

**1 Introduction to security** **9**

1.1 What cryptography is and is not . . . . . 9

1.2 Fundamental security principles . . . . . 10

1.3 Security parameter . . . . . 11

1.4 Security level . . . . . 14

**II Symmetric cryptography** **17**

**2 Randomness in cryptography** **19**

2.1 One-time pad . . . . . 19

2.2 Pseudorandom generators . . . . . 24

2.3 Linear feedback shift registers . . . . . 26

2.4 True randomness . . . . . 28

**3 Block ciphers** **29**

**4 Hash functions** **31**

**III Asymmetric cryptography** **33**

**5 Elementary number theory** **35**

5.1 Modular arithmetic . . . . . 35

**6 Algebraic structures** **39**

**7 Public-key encryption** **41**

**8 The Diffie–Hellman key exchange** **43**

**9 Digital signatures** **45**

**IV Other topics** **47**

**10 Cryptanalysis** **49**

**A Refreshers** **51**

A.1 Set notation . . . . . 51

A.2 Probability theory . . . . . 53

A.3 Asymptotic notation . . . . . 53

# Part I

# Introduction to modern cryptography

# Front page

*Latest update: 07/01/2021.*

This page contains some useful information related to the course and the lecture notes. Note that there are some useful buttons above the text, in particular one to download the notes as a well-formatted PDF, for offline use.

### Prerequisites

The course assumes that you are familiar with previous maths courses from your degree. In particular, we will be using concepts from Probability theory and Discrete mathematics, and make extensive use of modular arithmetic. Refreshers for these topics can be found in Appendix A and Section 5.1.

### Exercises

Besides the exercise lists that you will have for practices or seminars, these notes also have some exercises embedded into the explanations. These are mostly easy exercises, designed to be a sort of 'sanity check' before moving to the next topic. Hence, our recommendation is that you stop and think about every exercise you encounter in these notes, instead of rushing through the content. You might not always be able to write a full and formal solution, but make sure to get at least an intuition on each exercise before moving on.

### SageMath

*SageMath* (often called just *Sage*) is a powerful computer algebra system that we will use during the course to illustrate many concepts. It follows the Python syntax, which you will be familiar with, and comes equipped with many functions that are useful for cryptography.

These notes will sometimes provide chunks of Sage code, so that you can play with some of the schemes that we will introduce. You are also encouraged to

try and implement other schemes, or use Sage to double-check your solutions to exercises.

There are two ways that you can use Sage:

- Download it from https://www.sagemath.org/download.html. This will allow you to run SageMath locally. It also comes packaged with a Jupyter-style notebook, for ease of use.
- Use it online at https://cocalc.com/app. This also comes in both terminal and notebook flavors. CoCalc has a freemium model, and in the free version you will get an annoying message telling you that your code will run really slow. Nevertheless, the free version is more than enough for the purpose of this course.

The documentation at https://doc.sagemath.org/ is pretty good, although most of the functions that we will use are self-explanatory.

### Bibliography

1. Boaz Barak. An Intensive Introduction to Cryptography. *Available freely from* https://intensecrypto.org/public/.

2. Richard Crandall and Carl B Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006.

3. Christophe De Canniere and Bart Preneel. Trivium. In *New Stream Cipher Designs*, pages 244–266. Springer, 2008.

4. Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

5. Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

6. Mike Rosulek. The Joy of Cryptography, 2017. Available freely from http://web.engr.oregonstate.edu/~rosulekm/crypto.

### Changelog

- 07/01/2021. Refreshes on Appendix A and Section 5.1 added. Added code for LFSR. Some typos fixed in Sections 1 and 2.
- 04/01/2021. Sections 1 and 2 uploaded.

Notes written by Javier Silva, using Bookdown and pandoc.

# Chapter 1

# Introduction to security

We use cryptography on a daily basis: our wireless communications or web traffic are encrypted, companies protect their data with cryptographic algorithms, and so on. We all have a basic or intuitive understanding of how cryptographic algorithms work. In this chapter, we want to make this intuition more precise and give you tools to think about cryptographic algorithms more formally, and reason scientifically about security.

Therefore, in this section we will:

1. Introduce three basic principles of cryptographic algorithm design;
2. Introduce the notion of security parameter and security level.

## 1.1 What cryptography is and is not

Cryptography is a field that lies halfway between mathematics and computer science, and is occupied with building algorithms that protect communications in some way, for example ensuring privacy or integrity of a message sent through an insecure channel.

In this course, we will describe some of the most important cryptographic algorithms. They are the foundation for many security mechanisms and protocols that are part of the digital world. Thus, when you finish this course, you will have the basis to understand these mechanisms. But it is also important that you understand what is *not* covered in this course, and what are the limitations of what you will learn. In particular, a well-known course on cryptography[1] mentions three warnings that you should take into consideration:

[1] D. Boneh. Cryptography I, Coursera. Available at https://www.coursera.org/learn/crypto.

1. Cryptography is not the solution to all security problems;
2. Cryptography is not reliable unless implemented and used properly;
3. Cryptography is not something that you should try to invent yourself, as there are many and many examples of broken ad-hoc designs.

## 1.2 Fundamental security principles

Let us consider a common example. When we type our WiFi password to connect to a network, we are assuming that what we are doing is "secure" because only us have some secret information (the password), that allows us to do this. In the context of cryptography, we call this secret information the *secret key*.

But what if an attacker tries all the possible keys until he finds the right one? This is what is called a *brute-force attack*. We will consider a few different scenarios:

- Our secret key is a 4-digit number. Then, in the worst case, the attacker will need to try
$$10^4 = 10000$$
potential keys. Assuming one try per second, this will take a bit less than three hours.
- Our secret key is a 12-character string of digits and English letters. Since there are 10 possible digits and 26 possible letters for each position, the number of potential keys is
$$36^{12} = 4738381338321616896.$$
At the same rate, the attacker will need, approximately, $1.5 \cdot 10^{11}$ *years* to try all of them. For reference, this number is roughly half of the number of stars in the Milky Way.

**Exercise 1.1.** *A WiFi password is* 10 *bits long. Assume that an attacker tries one password per second. How long does it take to find the key by brute force? What if the password is* $\lambda$ *bits long?*

The idea is that, if our password is generated in a good way, this will take too long! So we are implicitly thinking that our scheme is secure because an attacker has limited time or limited money to buy hardware to perform very fast attacks and find our password. This leads to the first fundamental principle of modern cryptography:

**Principle 1**. *Security depends on the resources of the attacker. We say that a cryptographic scheme is secure if there are no efficient attacks.*

Cryptographic algorithms need to be carefully reviewed by the scientific community, and directions for implementation and interoperability must be given before they are adopted. This is done by institutions such as NIST, the National Institute of Standards and Technology in the US.[2] Thus, coming up with new, secure algorithms is difficult. In fact, the algorithms that are used in practice are public, known to everyone, and in particular to potential attackers. For instance, in the case of a WiFi password, it is a publicly-known fact that WPA is used. This is a general design principle in cryptography: security must come from the choice of a secret key and not from attackers not knowing which algorithm we are using.

**Principle 2** (Kerckhoffs's principle). *Design your system so that it is secure even if the attacker knows all of its algorithms.*[3]

So what makes our systems secure is the fact that, although the attacker knows the algorithm, it does not know the secret key that we are using. Back to the WiFi example, an attacker knows that the WPA standard is used, but they just don't know our password.

Another implicit assumption that we are making when we think that our connection is "secure" is that our secret key is *sufficiently random*, that is, that there are *many possibilities* for the secret key. This leads us to the third and last principle.

**Principle 3**. *Security is impossible without randomness.*

As we will see, randomness plays an essential role in cryptographic algorithms. In particular, it is always essential that secret keys are chosen to be sufficiently random (i.e. they should have enough entropy). A bad randomness source almost always translates into a weakness of the cryptographic algorithm.

## 1.3 Security parameter

Obviously, cryptographic algorithms need to be efficient to be used in practice. On the other hand, we have seen that no attack should be efficient at all, i.e. they should be *computationally infeasible*. Before we go on, we need to determine the meaning of efficiency, so that the concept is formal and quantifiable. At the same time, and because of Principle 1, we need to relate efficiency with security somehow.

The way to achieve this is through a natural number that we call the *security parameter*, usually denoted by $\lambda$. The information about both security and efficiency will be expressed in terms of the security parameter.

[2] https://www.nist.gov/.

[3] *Kerckhoffs's principle is named after Auguste Kerckhoffs, who published the article* La Cryptographie Militaire* in 1883.*

**Definition 1.1.** *An algorithm $\mathcal{A}$ is said to be* efficient *(or* polynomial-time*) if there exists a positive polynomial $p$ such that, for any $\lambda \in \mathbb{N}$, when $\mathcal{A}$ receives as input a bitstring of length $\lambda$, it finishes in $p(\lambda)$ steps.*

We note that here we are interested in having a rough estimate on the running time, so we count each basic bit operation as one step. We use equivalently the terms *running time*, *number of steps*, *number of operations*.

An important observation is that a *single polynomial* must work for any value of $\lambda$. Otherwise, any algorithm would be considered efficient. The intuition behind the definition is that we allow the running time of the algorithm to grow when the input gets larger, but not "too much too fast". Let us consider two examples to illustrate the concept of polynomial-time algorithms:

- Algorithm $\mathcal{A}$ takes two $\lambda$-bit integers $m, n$ and adds them.
- Algorithm $\mathcal{B}$ takes a $\lambda$-bit integer $n \in \{0, \ldots, 2^\lambda - 1\}$ and finds its prime factors in the following way: for each $i = 1, \ldots, n$, it checks whether $i$ divides $n$, and if that is the case it outputs $i$.

Are they efficient? The first one is efficient, because the number of operations is $O(\lambda)$, while the second one is inefficient because the number of operations is $O(2^\lambda)$. In other words, the number of operations grows *exponentially* when we increase the size of the input. As is well known, exponential functions grow much faster than polynomials, and so in this case we will not be able to find a polynomial to satisfy Definition 1.1. Thus, algorithm $\mathcal{B}$ is not efficient.

Below, you can find a Sage implementation of each of the two algorithms, with the tools to compare their running times for different sizes of $\lambda$. Observe that, by increasing the security parameter, soon the second algorithm starts taking too long to terminate.

```
# Choose the security parameter
sec_param = 12

# Generate two random numbers of bit length lambda
n = randrange(2^(sec_param-1),2^(sec_param)-1)
m = randrange(2^(sec_param-1),2^(sec_param)-1)

print ("n =",n,"\nm =",m)

# Define algorithm A, which adds the two numbers
def algorithm_a(n,m):
    n+m

# Measure the time it takes to run algorithm A
```

```
%time algorithm_a(n,m)

# Define algorithm A, which tries to factor a number
def algorithm_b(n):
    for i in range(1,n+1):
        if mod(n,i)==0:
            i

# Measure the time it takes to run algorithm B
%time algorithm_b(n)
```

**Exercise 1.2.** *Decide whether the following algorithms run in polynomial time:*

- *An algorithm that takes as input two integers $n, m$ (in base 2) and computes the sum $n + m$.*
- *An algorithm that takes as input an integer $n$ and prints all the integers from 1 to $\ell$, if:*
  - $\ell = n$.
  - $\ell = n/2$.
  - $\ell = \sqrt{n}$.
  - $\ell = 10^6$.
  - $\ell = \log_2 n$.

We are now in position to discuss the efficiency and security of a cryptographic scheme in more grounded terms. For cryptographic schemes that require secret keys, the security parameter $\lambda$ is the bit length of the key. All the algorithms that compose some cryptographic scheme, like an encryption or signature scheme, should run in time polynomial in $\lambda$.

A classical example of this is an encryption scheme, which is the cryptographic primitive that will be the focus of most of the course. We first introduce the notion of symmetric encryption scheme.[4]

**Definition 1.2.** *A* symmetric encryption scheme *is composed of three efficient algorithms:*

$$(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}).$$

- *The* KeyGen *algorithm chooses some key* k *of length $\lambda$, according to some probability distribution.*

[4] In the second half of the course, we will deal with the notion of *asymmetric encryption schemes*, in which there are two different keys, a *public key* that is used for encryption and a *secret key* that is used for decryption.

- *The* $\mathsf{Enc}$ *algorithm uses the secret key* $\mathsf{k}$ *to encrypt a message* $\mathsf{m}$*, and outputs the encrypted message*

$$\mathsf{c} = \mathsf{Enc}_{\mathsf{k}}(\mathsf{m}).$$

- *The* $\mathsf{Dec}$ *algorithm uses the secret key* $\mathsf{k}$ *to decrypt an encrypted message* $\mathsf{c}$*, recovering*

$$\mathsf{m}$$

*as*

$$\mathsf{Dec}_{\mathsf{k}}(\mathsf{c}) = \mathsf{m}.$$

*In this context,* $\mathsf{m}$ *is called the* plaintext*, and* $\mathsf{c}$ *is said to be its corresponding* ciphertext*.*

Technically, the fact that the algorithms are efficient is expressed as requiring that the three algorithms run in time polynomial in $\lambda$.[5]

On the other hand, observe that, if an attacker wants to try all the possible secret keys, it needs $O(2^\lambda)$ steps to do so. This is not polynomial time in the security parameter (again, it is exponential), so it is not efficient according to Definition 1.1.

## 1.4 Security level

Ideally, we would like that the best possible attack against a scheme is a brute-force attack, in which an attacker (also called *adversary*) needs to try all the possibilities. However, very often there exist much more sophisticated attacks that need less time. This motivates the following definition:

**Definition 1.3.** *A cryptographic scheme has* $n$-bit security *if the best known attack requires* $2^n$ *steps.*

When the best known attack is a brute-force attack, then $n = \lambda$, but we will see many examples of the opposite, which makes $n$ significantly smaller. In a few lessons, we will see the example of hash functions, for which, in the best case,

$$n = \frac{\lambda}{2}.$$

If we require a security level of 80 bits, this forces us to choose $\lambda = 160$, at the least. Another example is RSA, which is a famous encryption scheme that we

[5] In many cryptography books, you will find that $\mathsf{KeyGen}$ should be a (probabilistic) polynomial time algorithm that takes as input $1^\lambda$, which is the string with $\lambda$ ones. This is a way to write that $\mathsf{KeyGen}$ should be polynomial in $\lambda$.

will study later in the course. In that case, $\lambda$ needs to be 1024 to achieve a security level of roughly 80 bits.

Although all the algorithms that compose a scheme, like $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ in the encryption case, are still efficient, their running time typically increases with $\lambda$. The impact of this is that, the higher the value of $\lambda$, the more expensive the computations are.

But what is a good security level? Suppose you have some cryptographic algorithm that has $n$-bit security for key length $\lambda$. How do you decide what $n$ is appropriate for your scheme to be secure? How is $n$ to be chosen so that it is *infeasible* (i.e. inefficient for an adversary) to recover the key?

There is no unique answer to this question. As we saw in Principle 1, security is a matter of resources. If an adversary needs to use computational power to perform $2^n$ steps to attack your system, this will cost him money (electric power, hardware, etc). If your cryptographic tools are protecting something that is worth only 10€, an attacker will not be willing to spend a lot of money attacking it. RFID tags are a good example of this. On the other hand, if you are protecting valuable financial information, or critical infrastructure, you would better make sure that this costs the adversary a *lot* of resources.

A general rule of thumb is that a cryptosystem is expected to give you at the very least an 80-bit security level. By today's computing power levels, this is considered even a bit weak, and acceptable security levels are more around the 100-bit mark. This does not mean that any attack below $2^{100}$ can be easily run on your PC at home! The website https://www.keylength.com/ maintains a list of key size recommendations suggested by different organizations.

The following table, taken from Mike Rosulek's book *The Joy of Cryptography* gives some estimates of computational cost in economic terms.[6]

| Security level | Approximate cost | Reference |
|---|---|---|
| 50 | $3.50 | cup of coffee |
| 55 | $100 | decent tickets to a Portland Trailblazers game |
| 65 | $130000 | median home price in Oshkosh, WI |
| 75 | $130 million | budget of one of the Harry Potter movies |
| 85 | $140 billion | GDP of Hungary |
| 92 | $20 trillion | GDP of the United States |
| 99 | $2 quadrillion | all of human economic activity since 300,000 BC |
| 128 | really a lot | a billion human civilizations' worth of effort |

[6]Note that he uses the English definition of billion, that is, $10^9$. Same for the other amounts. Also, the table seems to be based on data from 2018, so the up-to-date numbers might vary slightly.

**Exercise 1.3.** *Determine the security level when:*

- *My password consists of* 20 *random letters of the Catalan alphabet.*
- *Same as above, but including also capital letters.*
- *My password is a word of the Catalan dictionary (88500 words).*

# Part II

# Symmetric cryptography

# Chapter 2

# Randomness in cryptography

As we saw above, and made explicit in Principle 3, we require randomness to guarantee secure cryptography. In this section, we will give some thought to how to obtain this randomness in the first place, and what to do when we do not have enough of it. As a motivating example, we will start by describing a well-known encryption scheme.

We will learn about:

1. The one-time pad encryption scheme;
2. Pseudorandom generators;
3. Sources of randomness.

## 2.1 One-time pad

The *one-time pad (OTP)* is an old encryption scheme, which was already known in the late 19th century, and was widely used in the 20th century for many military and intelligence operations.

The idea is extremely simple. Let us first recall the *exclusive or* (XOR) logic operation. Given two bits $b_0, b_1 \in \{0,1\}$, the operation is defined as

$$\mathsf{XOR}(b_0, b_1) = b_0 \oplus b_1 = \begin{cases} 0 & \text{if } b_0 = b_1, \\ 1 & \text{if } b_0 \neq b_1. \end{cases}$$

Equivalently, the operation corresponds to the following truth table:

| $b_0$ | $b_1$ | $b_0 \oplus b_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

We extend the notation to bitstrings of any length, i.e., given two bistrings $\mathbf{b}_0$ and $\mathbf{b}_1$ of the same length, we define

$$\mathbf{b}_0 \oplus \mathbf{b}_1$$

to be the bistring that results from XOR'ing each bit of $\mathbf{b}_0$ with the bit in the same position of $\mathbf{b}_1$.

Assume that Alice wants to send an encrypted message to Bob. The one-time pad works as follows. Key generation consists of choosing as a secret key a uniformly random bitstring of length $\lambda$ as the key:

$$\mathsf{k} = k_1 k_2 \dots k_\lambda.$$

We denote this process by $\mathsf{k} \leftarrow \{0,1\}^\lambda$. Let $m$ be a message that Alice wants to encrypt, written as a bitstring[1]

$$\mathsf{m} = m_1 m_2 \dots m_\lambda$$

of the same length. Then, the one-time pad encryption scheme works by XOR'ing each message bit with the corresponding key bit. More precisely, for the $i$th bit of the message, we compute

$$\mathsf{c} = \mathsf{m} \oplus \mathbf{k},$$

which is sent to Bob. Note that, because the XOR operation is its own inverse, the decryption algorithm works exactly like encryption. That is, Bob can recover the message by computing

$$\mathsf{m} = \mathsf{c} \oplus \mathsf{k}.$$

The first property that we want from any encryption scheme is *correctness*, which means that for any message $\mathsf{m}$ and any key $\mathsf{k}$, we have that

$$\mathsf{Dec}_\mathsf{k}(\mathsf{Enc}_\mathsf{k}(\mathsf{m})) = \mathsf{m},$$

that is, if we encrypt and decrypt, we should recover the same message. Otherwise Alice and Bob will not be able to communicate.

[1] If the message is written with a different set of characters, like English letters, it is first processed into a bitstring, e.g. by associating to each letter its ASCII code in binary (https://en.wikipedia.org/wiki/ASCII).

**Proposition 2.1.** The one-time pad is a correct encryption scheme.

*Proof.* Using the definitions of encryption and decryption, we have that

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = \mathsf{Dec}_k(m \oplus k) = (m \oplus k) \oplus k = m \oplus (k \oplus k) = m \oplus \mathbf{0} = m,$$

where $\mathbf{0}$ means the string of zeroes of size $\lambda$. In the last two steps, we used, respectively, that XOR'ing any string with itself produces $\mathbf{0}$, and that XOR'ing any string with $\mathbf{0}$ does not change the string.

□

Here is a straightforward implementation of the one-time pad. In this example, we want to send a message with 12 ASCII characters, so each character will require 8 bits. Thus, we choose a key length of 96.

```
from sage.crypto.util import ascii_integer
from sage.crypto.util import bin_to_ascii

# Set a security parameter
sec_param = 96

# Define the XOR operation:
def xor(a,b):
    return mod(int(a)+int(b),2) # You will learn why this is equivalent
                                # to XOR later in the course

### KEY GENERATION
# Generate a random key of length sec_param
k = random_vector(GF(2),sec_param)

### ENCRYPTION
# Choose a message
m = "Hello there."
# Process the message into a bitstring
m_bin = str(BinaryStrings().encoding(m))

# Encrypt the message bit by bit
c = ""
if (len(m_bin)<=sec_param):
    for i in range(len(m_bin)):
        c += str(xor(m_bin[i],k[i]))
    print("Ciphertext: "+c)
else:
    print("Message too long. Need a longer key.")
```

```
### DECRYPTION
# We use the same ciphertext obtained in the encryption part.

# Decrypt the ciphertext bit by bit
m_bin = ""
if (len(c)<=sec_param):
    for i in range(len(c)):
        m_bin += str(xor(c[i],k[i]))
    print("Plaintext: "+bin_to_ascii(m_bin))
else:
    print("Ciphertext too long. Need a longer key.")
```

The one-time pad receives its name from the fact that, when the key is used only once, the scheme has *perfect secrecy*. This means that the ciphertext produced reveals absolutely no information about the underlying plaintext, besides its length. We formalize this by saying that, given a ciphertext and two messages, the ciphertext has the same probability of corresponding to each of the messages.

**Definition 2.1.** *An encryption scheme has* perfect secrecy *when, for a uniformly random key* $\mathsf{k}$*, all ciphertexts* $\mathsf{c}$ *and all pairs of messages* $\mathsf{m}_0, \mathsf{m}_1$*,*

$$\Pr[\mathsf{c} = \mathsf{Enc}_{\mathsf{k}}(\mathsf{m}_0)] = \Pr[\mathsf{c} = \mathsf{Enc}_{\mathsf{k}}(\mathsf{m}_1)].$$

Intuitively, the perfect secrecy of the OTP stems from these two observations:

- Look again at the truth table of the $\mathsf{XOR}$ operation, and observe that a 0 in the plaintext could equally come from a 0 or a 1 in the plaintext, depending on the key bit. Similarly, a 1 in the ciphertext could also come from a 0 or a 1 in the plaintext. In other words, if the key is chosen uniformly at random, each bit of the ciphertext has a probability of $1/2$ of coming from a 0, and a probability $1/2$ of coming from a 1.
- Because of the above, an adversary that intercepts a ciphertext $c_1 c_2 \dots c_\lambda$ cannot know the corresponding plaintext, as any given plaintext can be encrypted to *any* bitstring of length $\lambda$. In other words, for every ciphertext $\mathsf{c}$ and every message $\mathsf{m}$, there exists a key $\mathsf{k}$ and a message such that

  $$\mathsf{Enc}_{\mathsf{k}}(m) = \mathsf{c} \qquad \text{and} \qquad \mathsf{Dec}_{\mathsf{k}}(\mathsf{c}) = m.$$

  So any ciphertext could correspond to any message, and there is no way to do better, regardless of the computational power of the attacker!

We formalize the above discussion in the following result.

**Proposition 2.2.** The one-time pad encryption scheme has perfect secrecy.

*Proof.* By the discussion above, we have that for any key $\mathsf{k}$, message $\mathsf{m}$ and ciphertext $\mathsf{c}$,

$$\Pr[\mathsf{c} = \mathsf{Enc}_\mathsf{k}(\mathsf{m})] = \frac{\#\{\text{keys } \mathsf{k} \text{ such that } \mathsf{c} = \mathsf{Enc}_\mathsf{k}(\mathsf{m})\}}{\#\{\text{possible keys}\}} = \frac{1}{2^\lambda}.$$

□

**Exercise 2.1.** *We said above that for every message* $\mathsf{m}$ *and any ciphertext* $\mathsf{c}$*, there is always exactly one key* $\mathsf{k}$ *such that*

$$\mathsf{Enc}_\mathsf{k}(m) = \mathsf{c} \qquad \textit{and} \qquad \mathsf{Dec}_\mathsf{k}(\mathsf{c}) = m.$$

*For arbitrary* $\mathsf{m}$ *and* $\mathsf{c}$*, which is that key, expressed in terms of* $\mathsf{m}$ *and* $\mathsf{c}$*?*

This is all well and good, but obviously there's a catch. While the security of one-time pad is as good as it gets, it is simply impractical for a very simple reason: we need a key as large as the message, and moreover, we need a new key for each message. Moreover, if we want perfect secrecy, this is unavoidable.

**Proposition 2.3.** Any encryption scheme with perfect secrecy requires a key that is as long as the message, and it cannot be reused.

One reason that highlights how reusing keys in OTP breaks perfect secrecy is the following. Assume that we use the same key $\mathsf{k}$ for two messages $\mathsf{m}_0, \mathsf{m}_1$. Then, an attacker intercepts the ciphertexts

$$\mathsf{c}_0 = \mathsf{m}_0 \oplus \mathsf{k}, \qquad \mathsf{c}_1 = \mathsf{m}_1 \oplus \mathsf{k}.$$

The adversary can compute

$$\mathsf{c}_0 \oplus \mathsf{c}_1 = (\mathsf{m}_0 \oplus \mathsf{k}) \oplus (\mathsf{m}_1 \oplus \mathsf{k}) = \mathsf{m}_0 \oplus \mathsf{m}_1 \oplus (\mathsf{k} \oplus \mathsf{k}) = \mathsf{m}_0 \oplus \mathsf{m}_1 \oplus \mathbf{0} = \mathsf{m}_0 \oplus m_1.$$

That is, the adversary can get the $\mathsf{XOR}$ result of the two messages. Even if they do not know any of the messages on their own, this leaks partial information (e.g. a 0 in any position means that the two messages have the same value on that position).

So it's clear that for OTP to work we need keys as long as the messages, and there is no way around that. But how much of a big deal is that? An issue that we have not addressed yet is the fact that, for any of this to happen, the two parties involved need to agree on a common key $\mathsf{k}$, that must remain secret for anyone else. If an insecure channel is the only medium for communication available:

- they cannot share the key unencrypted, since an attacker could be listening, and grab the key to decrypt everything that comes afterwards.

- they cannot encrypt the key, since they don't have a shared key to use encryption yet!

Later in the course, we will see that there are ways to securely share a key over an insecure channel. But for now, it suffices to say that these methods exist. However, sharing a new key of the size of the message, and a new one for each message, is simply not practical most of the time. Imagine the key sizes for sending audio or video over the Internet. This, ultimately, is what kills the one-time pad.

## 2.2 Pseudorandom generators

Before we move on, let us see if there is still some hope for the one-time pad. What if we start from a short uniformly random key $\mathsf{k}$, and try to expand it to a longer key?

Let us assume that Alice and Bob wish to communicate using the one-time pad, and Alice wants to send a message of length $h$. But they have only shared a key $\mathsf{k} \in \{0,1\}^\ell$, for some $\ell < h$, so they proceed as follows:

1. They agree on a public function

   $$G : \{0,1\}^\ell \to \{0,1\}^h.$$

   That is, $G$ receives a bitstring of length $\ell$ and outputs another of length $h$.

2. Since the function is deterministic, they can both compute

   $$\mathsf{k}' = G(\mathsf{k})$$

   on their own. Now they both know $\mathsf{k}' \in \{0,1\}^h$.

3. They use the one-time pad with key $\mathsf{k}'$.

Observe that, since they have already "stretched" the key once, they could potentially take parts of $\mathsf{k}'$ and apply the function $G$ again to generate new keys on demand. The scheme that results from stretching the randomness of a short shared key to an arbitrary length and encrypt the message through the $\mathsf{XOR}$ operation is known as a *stream cipher*. The initial key used is called the *seed*, and the subsequent keys generated are called the *key stream*.

The function $G$ must be deterministic, otherwise Alice and Bob will not arrive at the same key, and they will not be able to communicate. Also note that, although $G$ is public, $\mathsf{k}$ is not, so an attacker has no way of learning the new key $\mathsf{k}'$.

However, there are some caveats to this. Since the input of the function is a set of size $2^\ell$, there are at most $2^\ell$ outputs, whereas if we had used a uniformly random key of length $h$, we would have $2^h$ potential keys. Recall that perfect secrecy strongly relied on the keys being uniformly random, which clearly will not be the case here.

But, what if the output of $G$ looks "close enough" to random? By this, we mean that no efficient algorithm can distinguish the output distribution of $G$ and the uniform distribution in $\{0,1\}^h$. Then, if an adversary cannot tell that we are using a non-uniform distribution, they will not be able to exploit this fact in their attacks, and so our scheme will remain secure. Is any function $G$ good enough for our purposes?

**Exercise 2.2.** *Consider the stream cipher presented above, with the following choices for the function $G$, for $h = 2\ell$.*

1. *$G$ outputs a string of $2\ell$ zeroes.*
2. *$G$ outputs the input, followed by a string of $\ell$ zeroes.*
3. *$G$ outputs two concatenated copies of the input.*

*In each of these cases, discuss whether the scheme is still secure.*

The above exercise shows that we need to be careful when choosing our function $G$. This leads us to the following definition.

**Definition 2.2.** *A* pseudorandom number generator (PRNG) *is a function*

$$G : \{0,1\}^\ell \to \{0,1\}^h$$

*such that no efficient adversary can distinguish the output distribution of $G$ from the uniform distribution on $\{0,1\}^h$.*

We emphasize the importance of randomness here. A function $G$ whose output cannot be distinguished from uniform randomness by any (efficient) algorithm implies that, for all practical purposes, the output of $G$ can be considered uniformly random in $\{0,1\}^h$. In particular, informally this means that a key stream generated with a PRNG is *unpredictable*, i.e., given some output bits of $G$, there is no way to predict the next in polynomial time, with a success rate higher than 50%. This contrasts with non-cryptographic PRNGs, in which it is enough that the output passes some statistical tests, but might not be completely unpredictable.

**Exercise 2.3.** *Assume that there is a very bad PRNG that outputs one bit at a time, and that bit is a 0 with probability 3/4. This PRNG is used in a stream cipher to produce a ciphertext*

$$c = 01.$$

*In OTP, the probability of the corresponding plaintext being* 00*,* 01*,* 10 *or* 11 *would be* $1/4$ *each. Compute the corresponding probabilities when the bad PRNG described above is in use.*

An interesting property of PRNGs is that, if we manage to build one that stretches the key by just a little, then we can produce an infinitely large key stream, and still maintain essentially the same security guarantees. To illustrate this, let us again consider a function

$$G : \{0,1\}^\ell \to \{0,1\}^{2\ell},$$

and let us assume that it is a PRNG.[2] Consider the following construction of a new function

$$H : \{0,1\}^\ell \to \{0,1\}^{3\ell},$$

which works as follows: on input $\mathsf{k}$,

1. First compute $G(\mathsf{k}) \in \{0,1\}^{2\ell}$.
2. Split the result in two halves $\mathbf{x}, \mathbf{y}$, each of length $\ell$.
3. Compute $\mathbf{z} = G(\mathbf{y}) \in \{0,1\}^{2\ell}$.
4. Output $(\mathbf{x}, \mathbf{z}) \in \{0,1\}^{3\ell}$.

**Proposition 2.4.** If $G$ is a PRNG, then $H$, constructed as described above, is also a PRNG.

We have already seen some bad PRNGs, so what about the good ones? Although there exist some proposals of PRNGs that are believed to be secure and are built "from scratch", what happens in practice is that, when one wants a PRNG, it is common to build it from a block cipher, which is a topic that we will cover later in the course, so we delay the examples of cryptographic PRNGs until then. For completeness, we next look at a function that is enough for most applications of pseudorandom generation, but is not secure for cryptographic use.

## 2.3 Linear feedback shift registers

A *linear feedback shift register* is a type of "stretching function" that produces an output that looks quite random, and it passes some statistical tests, although it is still weak from a cryptographic point of view. We will start with a particular example. Assume that we have a seed $k$, written as a bitstring $\mathsf{k} = k_1 k_2 k_3$. The

[2] We set the output length to be $2\ell$ for simplicity, but the idea could easily be adapted to any other output length.

linear feedback shift register recursively produces each new element of the key according to the formula:

$$k_{i+3} = k_{i+1} \oplus k_i.$$

For example, if the seed is 011, the key stream will be

$$0111001 \quad 0111001 \quad 0111001 \quad 0111001 \quad 0111001 \quad \dots$$

We included the spaces to emphasize the fact that, after a while, the output seems to repeat. This is not something specific to this example, but actually happens to any linear feedback shift register.

Indeed, let us define a general *linear feedback shift register (LFSR) of length* $\ell$. It starts with a seed $\mathsf{k}$, expressed as a bitstring

$$\mathsf{k} = k_1 k_2 \dots k_\ell,$$

and derives each new element of the key stream according to the following: for

$$i > \ell$$

:

$$k_i = p_1 k_{i-1} \oplus \dots \oplus p_\ell k_{i-\ell},$$

for some coefficients $p_j \in \{0, 1\}$, for $j = 1, \dots, \ell$.

**Proposition 2.5.** The output of an LFSR of length $\ell$ repeats periodically, with a period of at most $2^\ell - 1$.

Note the "at most" in the statement. For some choices of the coefficients $p_j$, the period could be much shorter. However, for well-chosen coefficients, we can meet the bound, thus obtaining a period that is exponential in the length of the initial key. The output of a well-chosen LFSR has some good statistical properties. In particular, the output looks "random enough" for most applications. However, there are attacks that allow an adversary to distinguish the output from uniformly random, and thus LFSRs are not suited for cryptography.

Still, a clever combination of a few LFSRs, with a couple of extra details, seems to be enough to realize the stream cipher Trivium, which, to this date, is believed to be secure.

Below is a direct implementation of an LFSR. You can try different sets of feedback coeffients, and see how this impacts the period of the key stream.

```
# Define the XOR operation:
def xor(a,b):
    return mod(int(a)+int(b),2)

# Set a vector of feedback coefficients [p_1, ... , p_n]
feedback_coeffs = [1, 1, 0, 0, 0, 0, 0, 0]
```

```
seed_length = len(feedback_coeffs)

# Sample a uniformly random seed of the same length.
seed = list(random_vector(GF(2),seed_length))
print(seed)

# Choose the length of the required key stream
k = 16

# Run the LFSR
key_stream=seed
for i in range(seed_length,seed_length+k):
    key_stream_temp=0
    for j in range(seed_length):
        key_stream_temp = xor(key_stream_temp,(feedback_coeffs[j]*key_stream[i-j-1]))
    key_stream.append(key_stream_temp)
print(key_stream)
```

## 2.4 True randomness

We have dealt with the problem of stretching a tiny bit of randomness into something usable. But where does this initial randomness come from? It cannot really come from our computers, since these are deterministic, so the answer lies out in the physical world.[3] The general idea is to look for unpredictable processes from which to extract randomness. Some examples are radioactive decay, cosmic radiation, hardware processes like the least significant bit of the timestamp of a keystroke.

These processes might not produce uniformly random outputs, but from our perspective we have little to none information about their output distribution. These values are not used raw, but processed by a *random number generator (RNG)*, which refines them into what we assume to be uniformly random outputs. These can now be fed into our PRNGs to stretch them.

[3] Assuming, of course, that the universe is not completely deterministic.

# Chapter 3

# Block ciphers

# Chapter 4

# Hash functions

# Part III

# Asymmetric cryptography

# Chapter 5

# Elementary number theory

## 5.1 Modular arithmetic

Modular arithmetic is, informally, clock arithmetic. Let us take the usual analog 12-hour clock, and say it is 11 o'clock now. After three hours, it is 2 o'clock. But wait a minute, shouldn't it be $11+3=14$? Furthermore, after a whole day, shouldn't the clock show $11+24=35$ o'clock? But it is showing 11 instead!

This example shows that the usual integer arithmetic is not useful for modelling the behaviour of a clock. Let us see how we can modify it so that the passing of time makes sense again. We consider the following problem.

**Problem.** *Let $a \in \mathbb{N}$. Assume that the current position of the clock is* 12 *o'clock. What is the position of the clock after $a$ hours?*

The key observation is that full movements around the clock (that is, multiples of 12) do not matter, as they leave the clock in the same position. Recall that the algorithm of integer division tells us how to compute $q, r \in \mathbb{Z}$ such that

$$a = 12 \cdot q + r.$$

We call $q$ the *quotient* and $r < 12$ the *remainder* of the division. In the context of our problem, notice that $q$ is the number of full circles around the clock that happen in $a$ hours. Then, the only real change of position in the clock is determined by $r$, and $q$ does not matter at all.[1]

Generalizing this idea leads to the key concept of modular arithmetic, by replacing 12 by any positive integer. Moreover, observe that there is no need for $a$ to be a positive integer, as a negative value of $a$ can be interpreted as moving counter-clockwise.

[1] The only small caveat is that, when $a$ is a multiple of 12, the remainder will be 0, not 12, although both of these represent the same position. So, to be precise, let us assume that our clock has a 0 instead of 12, so that it perfectly aligns with the remainders.

**Definition 5.1.** *Let $a, n \in \mathbb{Z}$, with $n > 0$. We define the* remainder (or residue) of $a$ modulo $n$ *as the remainder of the integer division of $a$ by $n$, and we denote it by*

$$a \bmod n.$$

**Exercise 5.1.** *Compute the following values:*

$$25 \bmod 8, \qquad 1337 \bmod 7, \qquad 7 \bmod 13, \qquad -13 \bmod 12.$$

Modular arithmetic behaves in a similar way to usual arithmetic, as reflected in the following result:

**Proposition 5.1.** Let $a, b, n \in \mathbb{Z}$, with $n > 0$. Then:

$$\begin{array}{ll} (i) & (a \bmod n) + (b \bmod n) = (a + b) \bmod n. \\ (ii) & (a \bmod n) \cdot (b \bmod n) = (a \cdot b) \bmod n. \\ (iii) & \text{If } b \geq 0, \text{ then } (a \bmod n)^b = (a^b) \bmod n. \end{array}$$

It is clear that, when working modulo $n$, for some positive integer $n$, the only numbers that matter are $0, 1, \ldots, n-1$, since every other number can be linked to one of these when reduced modulo $n$. For example, back to the clock example, we have that $13 \bmod 12 = 1$, $25 \bmod 12 = 1$, and so on. It is not chance that related numbers are precisely those that represent the same position!

The above example seems to suggest that, modulo 12, the numbers

$$\ldots -23, -11, 1, 13, 25 \ldots$$

are "the same". This motivates the introduction of the idea of *congruence*.

**Definition 5.2.** *Let $a, b, n \in \mathbb{Z}$, with $n > 0$. We say that $a$ and $b$ are* congruent *if*

$$(a - b) \bmod n = 0.$$

*In this case, we represent this fact by*

$$a \equiv b \pmod{n}.$$

**Definition 5.3.** *Let $n \in \mathbb{Z}$, with $n > 0$. We define the* set of residue classes modulo $n$ *as the set*

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

*where the operations of addition and multiplication are reduced modulo $n$, according to Proposition 5.1.*

Then, the above discussion can be rephrased as follows. We can say that any $a \in \mathbb{Z}$ is congruent to its residue modulo $n$, that is, $a \bmod n$. Thus, to work with arithmetic modulo $n$, we can restrict ourselves to the set $\mathbb{Z}_n$.

**Proposition 5.2.** Let $a, \alpha, b, n \in \mathbb{Z}$, with $n > 0$. If $a \equiv \alpha \pmod{n}$, then:

- $a + b \equiv \alpha + b \pmod{n}$.
- $a \cdot b \equiv \alpha \cdot b \pmod{n}$.
- If $b \geq 0$, then $a^b \equiv \alpha^b \pmod{n}$.

In particular, this is true when $\alpha = a \bmod n$.

**Exercise 5.2.** *Prove that the logic* XOR *operation and addition modulo* 2 *are the same operation, by checking the four possible cases.*

# Chapter 6

# Algebraic structures

# Chapter 7

# Public-key encryption

# Chapter 8

# The Diffie–Hellman key exchange

# Chapter 9

# Digital signatures

# Part IV

# Other topics

# Chapter 10

# Cryptanalysis

# Appendix A

# Refreshers

## A.1 Set notation

A *set* is a well-defined collection of objects. Such objects are said to be *elements* of the set, or that they *belong* to the set. For example, the set of the vowels is

$$V = \{\mathrm{a}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}\}.$$

In the above line we are giving a name to the set, $V$, and we are specifying the list of its elements: a, e, i, o and u. When describing a set explicitly, we write the list of its elements between braces { }.

Two sets are equal if they have exactly the same elements. An element cannot belong 'twice' to a set. Therefore, we can say that

$$V = \{\mathrm{a}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}\} = \{\mathrm{u}, \mathrm{o}, \mathrm{i}, \mathrm{e}, \mathrm{a}\} = \{\mathrm{a}, \mathrm{a}, \mathrm{e}, \mathrm{e}, \mathrm{e}, \mathrm{i}, \mathrm{o}, \mathrm{u}\}.$$

The symbol $\in$ indicates membership of an element in a set. For example, we can write $\mathrm{a} \in V$, because a belongs to the set $V$. On the other hand, we have that $\mathrm{b} \notin V$, since b is not any of the elements of $V$.

There are some number sets that show up very frequently, so we give them special names.

- $\mathbb{N}$: set of natural numbers.
- $\mathbb{Z}$: set of integer numbers.
- $\mathbb{Q}$: set of rational numbers.
- $\mathbb{R}$: set of real numbers.
- $\mathbb{C}$: set of complex numbers.

Observe that, unlike in the prior examples, all these sets are *infinite*, that is, they have an infinite number of elements. Obviously we cannot describe an infinite set explicitly, as we have done with the set of vowels. What we can do is refer to other sets that we have already defined and define restrictions on them. For example, we can define the set of even natural numbers as the set of natural numbers that are multiples of 2. Formally, we write this set as

$$\{n \in \mathbb{N} \mid n = 2k \text{ for some } k \in \mathbb{N}\}.$$

Here we have introduced some new notation. Instead of explicitly enumerating all the elements of a set, we give some conditions. We read the description above as 'the set of elements of the form indicated on the left of the vertical line, such that they verify the condition on the right'. In this case, the set of natural numbers such that they are of the form $2k$ for some natural value of $k$. Similarly, we can define the set of all real numbers greater than 5 in the following way:

$$\{x \in \mathbb{R} \mid x > 5\}.$$

We denote the size (number of elements) of a set $S$ by $\#S$.

We can produce new sets by considering the *product* of known sets: given two sets $S, T$, we define the set $S \times T$ as the set whose elements are the pairs $(s, t)$, where $s \in S$ and $t \in T$. For example,

$$\{0,1\} \times \{0,1,2\} = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)\}.$$

Observe that $\#(X \times Y) = \#X \cdot \#Y$. It is easy to generalize this definition to products of three or more sets. In particular, given a set $S$, we define

$$S^n = \underbrace{S \times S \times \cdots \times S}_{n \text{ times}}$$

In this course, we will often use the set $\{0,1\}$ of possible bits, the set $\{0,1\}^\ell$ of possible bitstrings of length $\ell$, and the set $\{0,1\}^*$ of bitstrings of any length. Observe that

$$\#\{0,1\} = 2, \qquad \#\{0,1\}^\ell = 2^\ell, \qquad \#\{0,1\}^* = \infty.$$

**Exercise A.1.** *Write these sets using implicit notation:*

- *The set of complex numbers with real part equal to* 1*.*
- *The set of pairs, where the first component of the pair is a rational number and the second component is an odd natural number.*
- *The set of bitstrings of length* 10 *with exactly* 5 *zeros.*

## A.2 Probability theory

We will deal with probability distributions over finite sets. In particular, recall that the *uniform distribution* over a set $S$ is the probability distribution that assigns the same probability $1/\#S$ to each element of $S$. We denote sampling an element $x$ from the uniform distribution over $S$ by

$$x \leftarrow S.$$

For example, the notation

$$\mathbf{b} \leftarrow \{0,1\}^{128}$$

means that $\mathbf{b}$ is a uniformly random bitstring of length 128.

Given an event $A$, we denote the *probability of* $A$ by $\Pr[A]$. Given two events $A, B$, we denote the *probability of* $A$ *conditioned on* $B$ by $\Pr[A|B]$, and if $\Pr[B] \neq 0$ we have that

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]},$$

where $\Pr[A \cap B]$ means the probability of both $A$ and $B$ happening. Recall that, if $A$ and $B$ are independent events, then

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B].$$

**Exercise A.2.** *Compute the probability of a random bitstring of length* 4 *being the string* 1110.

## A.3 Asymptotic notation

Asymptotic notation allows us to easily express the *asymptotic behaviour* of functions, that is, how the function changes for arbitrarily large inputs, with respect to some other function. For example, take the functions defined by

$$f(x) = x, \qquad g(x) = x^2.$$

Both tend to infinity as $x$ tends to infinity, but the second does it "faster". More precisely, let

$$f, g : \mathbb{N} \to \mathbb{N}$$

be two functions. We write

$$f(x) = O(g(x))$$

when there is some $N, M \in \mathbb{N}$ such that, for all $x > N$, we have

$$f(x) \leq M \cdot g(x).$$

We read this as "$f$ is big-O of $g$". Then, back to the initial example, we can say write

$$x = O(x^2).$$

Note that, because the big-O notation "absorbs" constants into $M$, we can write

$$2x = O(x),$$

even though $2x \geq x$ for $x \in \mathbb{N}$.

Big-O notation is useful for representing bounds on the growth speed of a function. By saying, for example, that a function $f$ satisfies

$$f(x) = O(x^3),$$

we are saying that, at worst, the function $f$ grows as fast as a cubic polynomial. Therefore, in particular, it will not grow as fast as a polynomial of degree 4, or an exponential function like $2^x$.

We recall that logarithmic functions grow slower than polynomials of any degree, and polynomials of any degree grow slower than exponential functions. Given two polynomials of different degrees, the one with the higher degree grows faster.

**Exercise A.3.** *Decide whether each of these statements is true or false.*

- $10^{10}x^3 = O(x^4)$.
- $10^x = O(x^4)$.
- $\log(x) = O(x \log x)$.
- $4^x = O(2^x)$.