Cryptography lecture notes

Contents

Part I

Introduction to modern cryptography

Front page

Latest update: 21/01/2021.

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 5.3.

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

- 21/01/2021. Minor fixes and additions in Sections 4 and 5.
- 18/01/2021. Remainder of Section 5 uploaded. Some typos fixed in Sections 3 and 4.
- 13/01/2021. Section 4 uploaded.
- 08/01/2021. Section 3 uploaded.
- 07/01/2021. Refreshes on Appendix A and Section 5.3 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* λ *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 λ. 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 λ, 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 λ. 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 λ-bit integers m, n and adds them.
- Algorithm $\mathcal{B}$ takes a λ-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 λ. 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 ℓ, 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 λ 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 λ.

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 λ, 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* Enc *algorithm uses the secret key* k *to encrypt a message* m*, and outputs the encrypted message*

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

- *The* Dec *algorithm uses the secret key* k *to decrypt an encrypted message* c*, recovering*

$$\mathsf{m}$$

as

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

In this context, m *is called the* plaintext*, and* 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 λ.[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 KeyGen should be a (probabilistic) polynomial time algorithm that takes as input 1^λ, which is the string with λ ones. This is a way to write that KeyGen should be polynomial in λ.

will study later in the course. In that case, λ 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 λ. The impact of this is that, the higher the value of λ, 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 λ. 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 λ 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 ith 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 m and any key 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 λ. 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* k*, all ciphertexts* 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 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 λ. In other words, for every ciphertext c and every message m, there exists a key 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 k, message m and ciphertext 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* m *and any ciphertext* c*, there is always exactly one key* k *such that*

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

For arbitrary m *and* c*, which is that key, expressed in terms of* m *and* 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 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 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 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 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 ℓ 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 k'.

Observe that, since they have already "stretched" the key once, they could potentially take parts of 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 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, k is not, so an attacker has no way of learning the new key k'.

However, there are some caveats to this. Since the input of the function is a set of size 2^ℓ, there are at most 2^ℓ 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 2ℓ zeroes.*
2. *G outputs the input, followed by a string of ℓ 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 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 ℓ.
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 2ℓ 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* ℓ. It starts with a seed 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 ℓ 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

In this section, we focus on block ciphers, which are a more popular alternative to stream ciphers. Block ciphers are interesting not only for encryption, but they also have some interesting theoretical implications, since many other cryptographic primitives (like pseudorandom generators) can be built from block ciphers. In this section, we will learn:

1. What is a block cipher, and what are the properties of a good block cipher;
2. The different modes of operation of a block cipher;
3. Two prime examples of block ciphers: DES and AES.

3.1 Overview of block ciphers

Recall that the one-time pad, and stream ciphers in general, encrypt bits one by one. In contrast, block ciphers will split our plaintext in blocks of fixed length, and encrypt each of this as a single unit.

Definition 3.1. *A* block cipher of length ℓ *is an encryption scheme that encrypts a message of fixed length* ℓ.

When encrypting an arbitrarily large message, we will split it into blocks of length ℓ and encrypt each block, using the same key, unlike in the previous section where we tried to stretch the key. Because of this, we will require a good block cipher to satisfy two new properties, that we informally describe below:

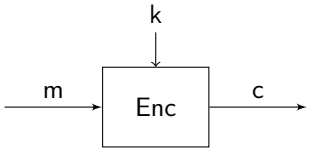
- *Confusion:* each bit of the ciphertext depends on several parts of the key. In other words, the relation between key and ciphertext must not be clear to any attacker.

- *Diffusion:* small changes in the plaintext result in significant changes in the ciphertext. More precisely, in any modern block cipher, it is expected that a single bit change in the plaintext should result in at least half of the bits of the ciphertext changing.

Later in this section, we will see some concrete examples of block ciphers used in practice. For now, let us assume that we already have some block cipher

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

that we will use as a black box. That is, for now we do not know what happens inside each of the algorithms, only that they work and they are secure. For example, we assume that Enc takes as input a plaintext m of length ℓ and produces a ciphertext c corresponding to m. We represent this by the diagram



This will allow us to discuss block ciphers in a more general way.

3.2 Modes of operation

Assume that we want to encrypt a message m of length ℓn with a block cipher. When the message length is not a multiple of the block length, some extra bytes are added to complete the last block. This is called *padding*.[1] We start by splitting the message in blocks

$$\mathsf{m}_1, \ldots, \mathsf{m}_n,$$

each of them of length ℓ, so that they can be fed into our block cipher. The question is: do we encrypt each block in parallel? Is that secure? Or should we somehow make the blocks influence each other? The way we proceed here is determined by the *mode of operation* that we choose.

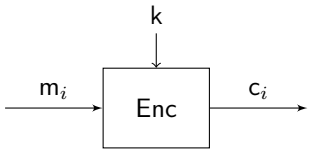
3.2.1 Electronic codebook (ECB) mode

In ECB mode, we take the most straightforward approach, and encrypt each block on its own:

$$\mathsf{c}_i = \mathsf{Enc}_k(\mathsf{m}_i).$$

The ECB mode is represented in the following diagram:

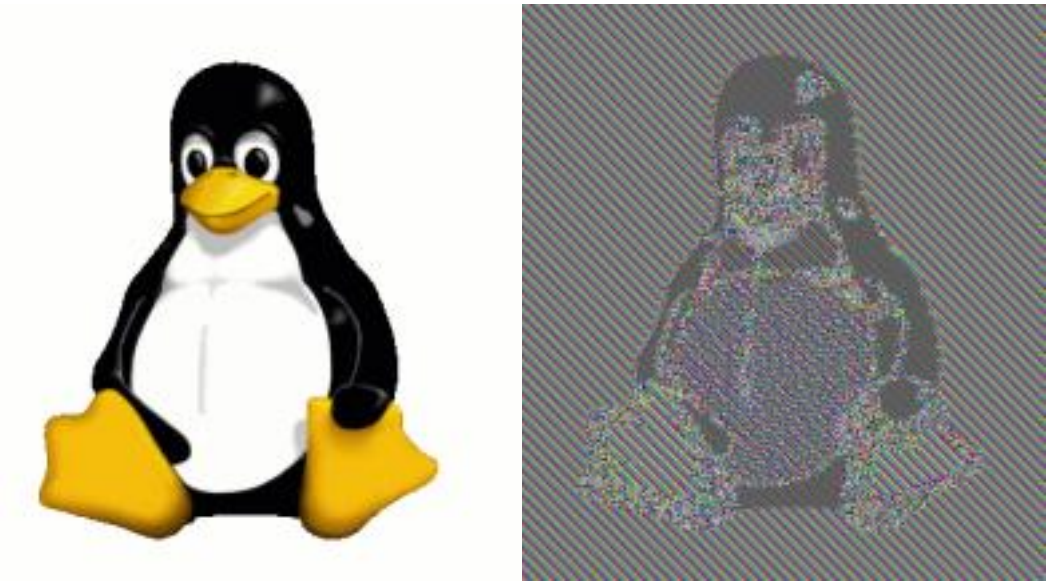
[1] One must be careful when choosing padding, as some choices are vulnerable to certain attacks in some modes of operation. An example of this is the padding oracle attack: https://en.wikipedia.org/wiki/Padding_oracle_attack.



The main advantage of this approach is that, since each block is independent, we can make the operations in parallel, potentially saving computation time.

However, this mode presents several weaknesses. For example, since each block is encrypted in exactly the same way, two identical messages result in two identical ciphertexts. So an eavesdropper can see when the same message was sent twice. Even if he does not know the content, this provides the attacker with some partial information, which is something that we would like to avoid.

Furthermore, the ECB mode is particularly bad when encrypting "meaningful" information. A very visual example comes from encrypting an image. Assume that we split the image into small squares of pixels, so that the bit length of each of these matches the length of our block cipher, and then use ECB-mode encryption on each square. Below you can see the result on an example image.[2]



Because the blocks are encrypted independently, a human eye can still easily distinguish the underlying information. This illustrates ECB mode's lack of diffusion.

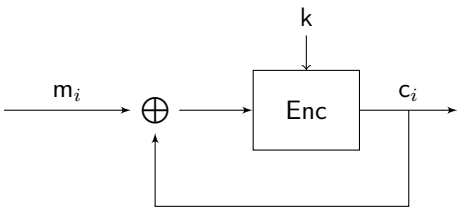
[2]Source: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation. Images by Larry Ewing (lewing@isc.tamu.edu) and GIMP (https://www.gimp.org/).

3.2.2 Cipher block chaining (CBC) mode

So we have seen that we want our blocks to interact in some way. To achieve this, the CBC mode takes the following approach: the idea is to create a *feedback loop*, in which each ciphertext produced by the block cipher is fed back into the input of the next iteration, by computing the XOR with the new input. More precisely:

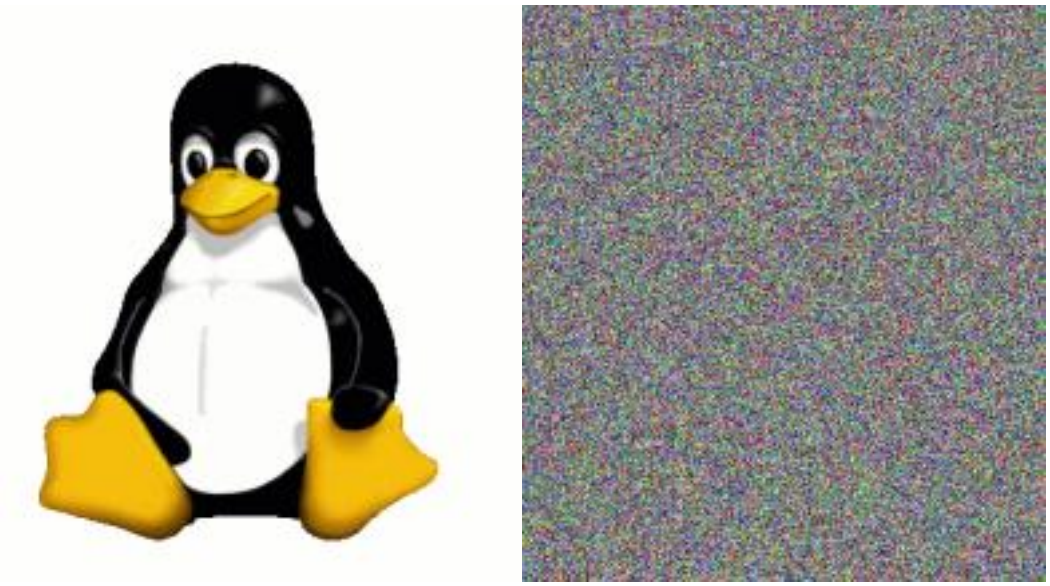
$$\mathsf{c}_i = \mathsf{Enc}_k(\mathsf{m}_i \oplus \mathsf{c}_{i-1}).$$

The CBC mode is represented in the following diagram:



Note that this does not work for the first block, since there is no previous ciphertext, and so we introduce something to replace it, which we call the *initialization vector* (often denoted by IV). By choosing the IV at random, we also introduce randomness in our scheme, making the procedure non-deterministic. Observe also that, due to the recursive nature of the definition, the encryption of a block is not only influenced by the previous block, but by *every* block that came before, and also the IV. There is no need for the IV to be secret, although it should not be reused, so if a new encryption sessions starts, a new IV should be chosen.

With this approach, we achieve a much higher diffusion. Looking again at the same picture, the result is now very different:



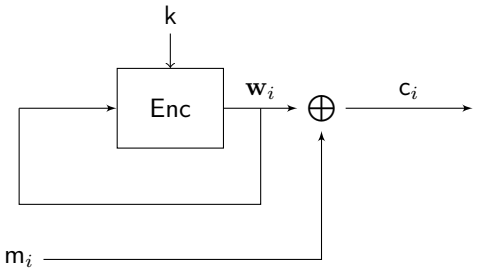
This is due to the fact that the encryption of each block influences the next. Thus, two identical blocks (for example, two squares of white in the corner of the picture) do not produce the same output anymore. The downside of this approach is that, since we need a ciphertext before we can compute the next, we cannot parallelize the computations.

3.2.3 Output feedback (OFB) mode

The next mode of operation actually turns a block cipher into a stream cipher, by recomputing a key each time through the Enc algorithm. That is, it is a stream cipher in which the key stream is produced in blocks of length ℓ:

$$\begin{aligned} \mathbf{w}_i &= \mathsf{Enc}_k(\mathbf{w}_{i-1}), \\ \mathsf{c}_i &= \mathsf{m}_i \oplus \mathbf{w}_i. \end{aligned}$$

The OFB mode is represented in the following diagram:



Again, we need an IV to feed into Enc in the first iteration. Observe that the block cipher and the feedback loop do not involve the message at all, which is simply XOR'ed with the result of each iteration of the loop to produce the ciphertext, as in any stream cipher.

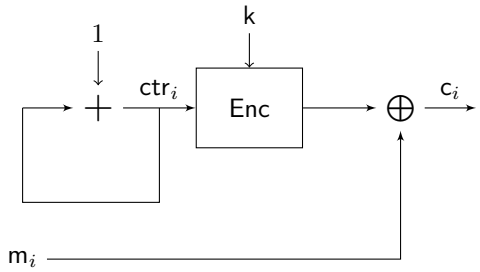
While, as the CBC mode, the computation cannot be performed in parallel, the fact that the loop does not depend on the message at all allows us to precompute a bunch of key blocks in advance, for later use with a message.

3.2.4 Counter (CTR) mode

Similarly to the OFB mode, the CTR mode produces a stream cipher from a block cipher. It works by keeping a public counter ctr that is chosen randomly, and is increased by 1 after each iteration. The counter works as an IV that updates after encrypting each block.

$$\begin{aligned} \mathsf{ctr}_i &= \mathsf{ctr}_{i-1} + 1, \\ \mathsf{c}_i &= \mathsf{m}_i \oplus \mathsf{Enc}_k(\mathsf{ctr}_i). \end{aligned}$$

After the last iteration, the current value of the counter is also sent, together with the ciphertext. The CTR mode is represented in the following diagram:



Exercise 3.1. *We have not discussed the decryption procedure of any of the modes of operation. Given a* Dec *algorithm that recovers plaintexts encrypted with* Enc*, describe how decryption works for each mode of operation. Recall that the values of the IV and counter are public.*

3.3 DES and AES

Now that we know how to use block ciphers, let us look a bit into some of the most famous ones: the *Data Encryption Standard (DES)*, and its successor the *Advanced Encryption Standard (AES)*. DES was designed by a team at IBM in 1974, and became the first official encryption standard in the US in 1977.

It remained as the recommended encryption scheme until 1999, when it was replaced by AES.[3]

3.3.1 Data Encryption Standard (DES)

DES is a block cipher of length 64, which uses a key of 56 bits. Essentially, it is composed of 16 identical rounds, each of them consisting of the following. During each round, a *round key* k_i of 48 bits is derived from the master key. In round i, the algorithm receives m_{i-1}, the output of the previous round (or the original message, for $i = 1$), and computes m_i, the output of the current round. In between, the following step happen:

1. Split m_{i-1} in two halves $\mathbf{L}_{i-1}, \mathbf{R}_{i-1}$ of 32 bits each.
2. Derive the round key k_i from k.
3. Set $\mathbf{L}_i = R_{i-1}$ and $\mathbf{R}_i = L_{i-1} \oplus f(\mathbf{R}_{i-1}, \mathsf{k}_i)$.
4. Return $\mathbf{m}_i = (\mathbf{L}_i, \mathbf{R}_i)$

After round 16, the end result m_{16} is the ciphertext.

Decryption of DES is almost the same as encryption, starting from the last round key. Observe that one half of the input of each round is not encrypted, just moved around, and so in total each half of the plaintext is encrypted 8 times by XOR'ing it with a function of the round key. There are a couple of details that we have not specified yet:

- How to derive round keys from the master key k. Without getting into much detail, the k_i is obtained from k by performing some rotations and permutations on the positions of the bits, and then some bits are ignored.
- How the function f works. First, the function expands the 32-bit input $\mathbf{R}_{i-1}$ to a 48-bit string, by repeating some of the bits in specified positions. The result is then XOR'ed with the round key k_i. The result from this operation is then split into 8 blocks of 6 bits each, and fed into what is known as *substitution boxes (S-boxes)*, which are functions specified by a lookup table. Each box outputs a string of 4 bits, so in total we have a string of 32 bits. Finally, the positions of the bits in this string are permuted, and the result is the output of the function f.

[3] Some interesting bits of history around DES and NSA involvement can be found in Chapter 3 of Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009..

The design of the cipher, specially the function f, might look arcane. Indeed, since the design and standardization of DES was not a public process, the reason behind some design choices is still not completely clear. What is known, however, is that the f function and the S-boxes were designed to thwart any attack known at the time (and even some that were not known to the public). The takeaway message here is that the S-boxes and the final permutation play a big role in achieving a good level of diffusion, propagating change through the whole ciphertext in the following rounds. Indeed, we have the following result.

Proposition 3.1. By the end of the fifth round of DES, every single bit of the current ciphertext depends on all the bits of the plaintext and all the bits of the key.

No sophisticated efficient attacks are known against DES to date. However, the key size is simply too small for today's standards (look again at the table at the end of section 1), and so it was eventually replaced by AES. Some variants of DES, like 3DES, which essentially means applying DES three times in a row, are still in use, and have withstood any attacks so far.

3.3.2 Advanced Encryption Standard (AES)

AES is a block cipher with block length 128 bits. Unlike DES, which used 56-bit keys, AES supports keys of bit length 128, 192 and 256, and has between 10 and 14 rounds, depending on the key length. Moreover, while in DES only half of the block was encrypted in each round, the full block is encrypted in every round now. On a very high level, each round consists of the following steps, called *layers*:

1. *Key addition layer:* a round key of length 128 is derived from the master key, in a process called *key schedule*.
2. *Byte substitution layer:* similarly to DES, AES uses 16 S-boxes defined by lookup tables, replacing each byte of the message by a new byte specified by the corresponding S-box. This layer introduces confusion.
3. *Diffusion layer:* the position of the bytes are permuted. Then, blocks of four bytes are combined using some matrix operations.

Regarding security, no attack more efficient than brute force is known to date. Thus, the security level provided by AES is λ, where $\lambda \in \{128, 192, 256\}$ is the bit size of the key.

The following video has a very clear and concise overview of the inner workings of AES.

Chapter 4

Hash functions

In this section, we take a detour from encryption to look at other cryptographic primitives. You might have encountered hash functions before, in a different field. However, we will see that hash functions in cryptography require some special properties. We will:

1. Briefly discuss some issues in cryptocurrencies, and how they can be solved with hash functions.
2. Define hash functions and their main properties.
3. Learn about the birthday paradox attack on hash functions.
4. Learn how to extend the domain of a hash function through the Merkle-Damgärd transformation.

4.1 Some issues in cryptocurrencies

Traditional currency is centralized, which means that there is an authority that dictates money policy, establishes ownership, and manages the whole system. On the other hand, in recent decades there has been a substantial effort in using cryptographic tools to build what we know as *cryptocurrencies*, which aim to be completely decentralized.

In this section, we discuss some issues that arise in decentralized systems. This is a very high level overview, based on the Bitcoin[1] approach, and omits many technicalities for the sake of the exposition. Nevertheless, it will be enough to motivate the use of hash functions.

[1] https://bitcoin.org/bitcoin.pdf.

A *coin*, the monetary unit of a cryptocurrency, is nothing more than a unique bitstring ID that identifies is, and is accordingly called its *identifier*. An immediate problem arises regarding transferring ownership of a coin.

Problem 1 (Double-spending). *Suppose that A buys something from B on the internet and pays with a coin ID. What prevents A from using the same coin ID to buy something else from a different party C?*

On very general terms, the solution is to publish every transaction that happens, so that the journey of each coin can be traced and thus its ownership can be established. In the problem above this means that, when A buys from B, the message "A transfers the coin ID to B" is added to the *public ledger*. Then, after the ledger awards B ownership of the coin, they can send whatever A bought. Moreover, if A tries to spend the same coin again, C will notice in the ledger that the coin no longer belongs to A, and the transaction will be denied. So, ignoring the logistics of checking and storing and increasingly large ledger, we would have solved the issue. But we still have to deal with the following problem.

Problem 2. *Who keeps track of this public ledger? Who adds the new transactions? If there is no central authority, how do users agree on which transactions happened?*

More concretely, imagine that there is a ledger of transactions

$$t_1, t_2, \ldots, t_n,$$

and two different options t_{n+1}, t'_{n+1} are claimed to be the next transaction by different parties:

$$t_1, t_2, \ldots, t_n \begin{cases} \nearrow t_{n+1} \\ \searrow t'_{n+1} \end{cases}$$

This situation is called a *fork*. The system is designed in such a way that users are encouraged to keep a *consensus* on a ledger of valid transactions. The general idea is that you have more of a say if you have more computational power, or more precisely if you have spent more CPU cycles in adding transactions to the ledger. So we need a way to "prove" that you have spent these cycles.

Let

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

for some $k, \ell \in \mathbb{N}$, where in general k is much larger than ℓ, be an efficiently computable function. Suppose that we are interested in the problem of finding $\mathbf{x}$ such that $H(\mathbf{x}) = \mathbf{0}$, where $\mathbf{0}$ is the string of zeros of length ℓ. If we know nothing about H, the best we can do is try random inputs until we find a good

one. On average, it would require 2^ℓ attempts to find such $\mathbf{x}$. That is, whoever shows a solution $\mathbf{x}$ has "proven" that he spent $O(2^\ell)$ evaluations of H in solving the problem. This concept is known as a *proof of work*.

But back to transactions and the ledger: assume that the transaction t_{n+1}, involving coin ID, is to be added to the ledger. Then, our proof of work consists of producing $\mathbf{x}$ such that the first T bits of $H(ID|\mathbf{x})$ are 0, for some T. Once you have the solution, you can add the transaction, including ID and $\mathbf{x}$, to the ledger. Note that solving the problem takes time $O(2^T)$, whereas checking a solution is efficient, as it amounts to evaluating the function H just once with the transaction as input.

But isn't this a lot of trouble to get someone's transaction up in the ledger? The solution here is extremely simple: motivate the users of the network by awarding them newly mint coins when they successfully add a new transaction to the ledger. These users are the so-called *miners*, and the act of mining a cryptocurrency is just finding the right preimages of the function H.

Thus, each addition to the ledger has two outcomes: transferring ownership of existing money and minting new money. And here's the catch: as a miner, your new money is just valid a hundred transactions after your contribution to the ledger. Give a fork, honest users are encouraged to look at the longest ledger and ignore the rest. This encourages miners to work on the single longest ledger too, because the alternatives will be rejected by the users and thus the transactions in them virtually never happened. Therefore, miners might spend CPU cycles for nothing if they decide to work on shorter ledgers of a fork.

Another issue is that we want each new transaction to be "bound" to the previous ones. If the new transaction did not depend on the previous, nothing would prevent a malicious user from double-spending. This is also achieved through the function H. Given previous transactions $t_1, \ldots, t_n$, the new transaction t_{n+1} will include its own *transaction identifier* $H(t_1, \ldots, t_n)$ besides ID and $\mathbf{x}$. However, the ledger gets larger and larger, and we want the identifier to be small to keep things efficient. So H is mapping a very large set into a smaller one. The upshot is that there is no way for transaction identifiers to be unique.

Problem 3. *What if there are two sets of transactions $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$ with the same identifier $H(t_1, \ldots, t_n) = H(t'_1, \ldots, t'_n)$?*

Fortunately, although it is clear that there is no possible function H such that the outputs are unique, it will be enough if it is *hard* to find a pair of inputs with the same output, so that this issue with the identifiers cannot be exploited in practice.

4.2 Hash functions

The central piece of this whole apparatus seems to be the function H, which we have not looked into yet. Clearly we will require some unconventional properties

from this function. Let us summarize what we discussed about it:

- The input is larger than the output, possibly by much.
- Given $\mathbf{y}$, it should be hard to find $\mathbf{x}$ such that $H(\mathbf{x}) = \mathbf{y}$.
- It is hard to find $\mathbf{x}, \mathbf{x}'$ such that $\mathbf{x} \neq \mathbf{x}'$ and $H(\mathbf{x}) = H(\mathbf{x}')$.

With this intuition in mind, let us introduce the solution to all of our problems: *hash functions*. At their core, hash functions are nothing more than functions that take an arbitrarily-long bitstring and output a bitstring of fixed length.

Definition 4.1. *A* hash function *is an efficiently computable*[2] *function*

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

for some $\ell \in \mathbb{N}$, and where $\{0,1\}^*$ denotes the set of all bitstrings of any length. The process of computing a hash function is often called hashing*, and the output is referred to as the* hash.

Note that, unlike encryption, hash functions do not use any secret key. From a functionality point of view, this is all we need: a function that compresses bitstrings and is efficient enough to compute. However, to ensure the security of the cryptocurrency model described above, we will need an extra property.

We observe that hash functions must be public and deterministic, because different parties need to be able to arrive to the same result to verify a transaction.

The hash function is taking arbitrarily-large messages and producing fixed-length ones. That it, it is mapping a larger set into a smaller set. Therefore, there must be different strings that produce the same hash. Given a bitstring $\mathbf{b}$, there might exist $\mathbf{b}' \neq \mathbf{b}$ such that

$$H(\mathbf{b}) = H(\mathbf{b}').$$

Nevertheless, we want this pair of bitstrings to be hard to find. This, and the observations at the beginning of Section 4.2, motivate the following set of definitions.

Definition 4.2. *Let H be a hash function. We say that H is:*

- collision-resistant *if it is hard to find two bitstrings $\mathbf{b}, \mathbf{b}'$ such that $\mathbf{b} \neq \mathbf{b}'$ and $H(\mathbf{b}) = H(\mathbf{b}')$. In this case, the pair $(\mathbf{b}, \mathbf{b}')$ is called a* collision *of H.*

[2] *You might wonder what "efficiently computable" means in this case, if the input size could be anything. To be precise, we say that the function is efficiently computable if it can be evaluated in time polynomial in ℓ when the input is of length polynomial in ℓ.*

- second preimage-resistant *if, given* $\mathbf{b}$*, it is hard to find* $\mathbf{b}' \neq \mathbf{b}$ *such that they form a collision.*
- preimage-resistant *if, given* h *sampled uniformly at random, it is hard to find a bitstring* $\mathbf{b}$ *such that* $H(\mathbf{b}) = h$*.*

These properties are related by the following result.

Proposition 4.1. Let H be a hash function

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

If H is collision-resistant, then it is second preimage-resistant.

Exercise 4.1. *Try to prove the proposition above by proving the contrapositive: assume that you can break second preimage resistance, and show how to use that to break collision resistance.*

Informally, second-preimage resistance implies preimage resistance for any hash function that performs some "meaningful" compression of the input. This means that, for any hash function used in practice, if it is second-preimage resistant then it its preimage resistant, although the statement cannot be formally proven, due to some pathological counterexamples.

4.3 Birthday attacks

Assume that we are an adversary trying to attack a hash function

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

that is, we are trying to find a collision. The straightforward approach is the following: we choose random strings and compute their hashes, until two strings return the same hash. In the worst case, this requires $2^\ell + 1$ tries, since there are at most 2^ℓ different outputs. Therefore, it looks like the brute-force attack takes time $O(2^\ell)$ to succeed. This would suggest that a hash function with output length ℓ gives us a security level of ℓ.

In this section, we look into a generic attack that works for any hash function, which is based on the well-known *birthday paradox* from probability theory, and greatly improves over the above estimation. Consider the following problem.

Problem 4. *There is a room with* 40 *independent students. How likely is that any two of them share the same birthday?*

On first sight, one might think that this probability is quite low. After all, there are 356 days in the year, and only 40 students. Let us compute the actual probability, by solving a related problem: what is the probability of none of the 40 students sharing their birthday?

We start by numbering the students from 1 to 40, according to any criterion. To be able to reason more formally about the problem, we introduce the function

$$\mathsf{bd} : \{1, \ldots, 40\} \to \{1, \ldots, 365\},$$

which associates to each student its birthday. Then, the probability of student #2 not sharing a birthday with student #1 is

$$\Pr[\mathsf{bd}(1), \mathsf{bd}(2) \text{ are different}] = \frac{364}{365},$$

since there are 364 days of the year that are not the birthday of student #1. Let's introduce student #3 into the picture, and let us consider the events:

- A: $\mathsf{bd}(3)$ is different from $\mathsf{bd}(1)$ and $\mathsf{bd}(2)$.
- $\mathsf{B} : \mathsf{bd}(1), \mathsf{bd}(2)$ are different.

Clearly, the intersection event is

- $\mathsf{A} \cap \mathsf{B} : \mathsf{bd}(1), \mathsf{bd}(2), \mathsf{bd}(3)$ are pairwise different.

Then, using conditional probabilities, we have that

$$\Pr[\mathsf{A} \cap \mathsf{B}] = \Pr[\mathsf{B}] \cdot \Pr[\mathsf{A}|\mathsf{B}]$$

We already know $\Pr[\mathsf{B}]$, so we are just missing the second term. If the birthdays of students #1 and #2 are different, then the probability of #3 having a different birthday from them is

$$\Pr[\mathsf{A}|\mathsf{B}] = \frac{363}{365},$$

since there are 363 days that are neither the birthday of #1 or #2. Thus, the probability of the three students having different birthdays is

$$\Pr[\mathsf{A} \cap \mathsf{B}] = \frac{364}{365} \cdot \frac{363}{365}.$$

By iterating this process for each student, we arrive at the conclusion that the probabilities of all 40 students having different birthdays is

$$\frac{364}{365} \cdot \frac{363}{365} \cdot \cdots \cdot \frac{326}{365} \approx 0.108768.$$

In conclusion, the probability of two students sharing a birthday is approximately

$$1 - 0.108768 = 0.891232.$$

This is actually a pretty high probability. This discrepancy between what one might naively expect and what actually happens is known as the *birthday paradox*. Below, you can find the solutions to Problem 4 for different numbers of students (rounded to six decimal positions).

Students	Probability
10	0.116948
20	0.411438
40	0.891232
80	0.999914
128	0.999999

So what does any of this have to do with breaking a hash function? What we have just done is computing the probability of finding two students such that the birthday function returns the same value on them. That is, we have found a collision of the birthday function! In doing so, we have assumed that the output of the birthday function behaves as the uniform distribution on $\{1, \ldots, 365\}$. But, isn't that exactly the effect that we want from a good hash function? That outputs look random and unrelated? So the moral of the story is that finding collisions in a hash function is actually much more likely that expected. More precisely, it can be proven with some careful probabilities analysis that, for any hash function H which outputs bitstrings of length ℓ, there is a decent probability of finding a collision after $\sqrt{2^\ell}$ evaluations.

Compare this with our initial estimation. At the beginning of the section, we bounded a brute force attack by $O(2^\ell)$. However, we now see that an attacker has a good probability of finding a collision in time $O(2^{\frac{\ell}{2}})$. Thus, we conclude that a hash function with output length ℓ gives us $\ell/2$ bits of security. Or the other way around, if we want ℓ bits of security, we need our hash function to have output length 2ℓ.

4.4 The Merkle-Damgård transformation

As was the case for encryption, we often build hash functions in two steps. First, we build a hash function for fixed-length inputs, e.g.

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

and then we extend them to arbitrarily-large input. We will not get into the details of concrete constructions, but will simply mention the SHA family of hash functions, which is the standard used in practice most of the time.[3]

A common way to realize this second step is to use the *Merkle-Damgård transformation*,[4] which describes how to build from H another hash function $\mathbf{H}$ that

[3] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms.

[4] You might also see the same concept named the Merkle-Damgård transform, or the Merkle-Damgård construction.

takes as input any string of length at most $2^\ell - 1$, and outputs a hash of length ℓ. It is clear that repeated applications of the transformation can make the input go as large as we want.

Similar to modes of operations in block ciphers, the Merkle-Damgård transformation starts by splitting the string $\mathbf{x}$ of length $L \leq 2^\ell$ to be hashed into blocks

$$\mathbf{x}_1, \ldots, \mathbf{x}_n,$$

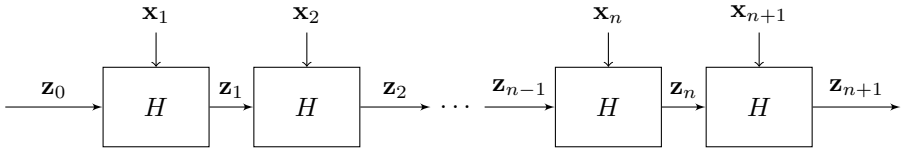
each of them of length ℓ.[5] An additional block $\mathbf{x}_{n+1}$ is added, containing a binary encoding of L. Note that, because $L \leq 2^n - 1$, we can fit the encoding of L in n bits. Then, we recursively compute

$$\mathbf{z}_i = H(\mathbf{z}_{i-1}|\mathbf{x}_i),$$

for $i = 1, \ldots, n+1$, and where $(\mathbf{z}_{i-1}|\mathbf{x}_i)$ means the concatenation of the bitstrings $\mathbf{z}_{i-1}$ and $\mathbf{x}_i$. Then, the hash of $\mathbf{x}$ is

$$\mathbf{H}(\mathbf{x}) = \mathbf{z}_{n+1}.$$

As in modes of operation, there is no "previous block" in the first iteration, and so again we introduce an initialization vector $\mathbf{z}_0$, which can be set to the string of 0's of length n, or any other bitstring. There is no need for the IV to be secret.



Proposition 4.2. If H is a collision-resistant hash function, then $\mathbf{H}$, produced with the Merkle-Damgård transformation, as described above, is also collision-resistant.

[5] As before, use some padding if the length of $\mathbf{x}$ is not a multiple of ℓ.

Part III

Asymmetric cryptography

Chapter 5

Elementary number theory

The second half of the course relies strongly on some ideas from number theory, which is the branch of mathematics that deals with integer numbers and their properties. This section and the next contain mathematical background that we will require to build some asymmetric cryptography. In this section, we will:

1. Review integer and modular arithmetic.
2. Discuss algorithmic aspects of modular arithmetic.

Note. In these notes, we use the convention that $\mathbb{N}$ does not include 0. We will refer to the set of non-negative integers by $\mathbb{Z}_{\geq 0}$.

5.1 Integer arithmetic

Consider the set $\mathbb{Z}$ of integer numbers. A key concept to the whole section is that of *divisibility*.

Definition 5.1. *Let $a, b \in \mathbb{Z}$. We say that b* divides *a if there exists $m \in \mathbb{Z}$ such that*

$$bm = a.$$

We denote that b divides a by $b \mid a$. In this case, we also say that b is a divisor *or* factor *of a, or that a is* divisible *by b, or that a is a* multiple *of b.*

Note that it is crucial that $m \in \mathbb{Z}$ in the definition above. Otherwise, any number b would divide any other number a, since

$$b\frac{a}{b} = a,$$

and then this notion would be pretty meaningless. Divisibility is related to the notion of *integer division*.

Proposition 5.1 (Integer division)**.** Let $a, b \in \mathbb{Z}$, with $b \neq 0$. Then, there exists a unique pair $q, r \in \mathbb{Z}$ such that $0 \leq r < b$ and

$$a = bq + r.$$

The integer q is called the *quotient* of the division, and r is called the *remainder.*

In Sage, quotient and remainder can easily be computed with the commands `a // b` and `a % b`, (or `mod(a,b)`) respectively.

By looking at the above proposition and the definition of divisibility, it is easy to see that $a \mid b$ if and only if the remainder of the division of a by b is 0.

Divisibility allows us to identify a special type of integers that are the building blocks of any other integer number. Clearly, any integer $n \in \mathbb{Z}$ is always divisible by $1, -1, n$ and $-n$, which are called its *trivial divisors.* Some numbers have more divisors, and some do not.

Definition 5.2. *An integer $p > 1$ is said to be a* prime number *if its only divisors are the trivial divisors. A positive integer that is not prime is said to be* composite.

Exercise 5.1. *Decide whether each of these statements is true or false:*

1. 35 *is a divisor of* 7.
2. 4 *is a factor of* 16.
3. 99 *is divisible by* 9.
4. *The remainder of dividing* -21 *by* 8 *is* -5.
5. 19 *is a prime number.*
6. 41 *is a composite number.*

The following two results show some interesting properties of prime numbers. Informally, the first states that any number can be decomposed into its prime factors, and that this decomposition is essentially unique, and the second states that, asymptotically, the chance of choosing a random number smaller than n and finding a prime is $\log(n)/n$.

Proposition 5.2 (Fundamental theorem of arithmetic)**.** Let $n \in \mathbb{Z}$. Then there exist prime numbers $p_1, \ldots, p_\ell$ and integers $e_1, \ldots, e_\ell$ such that

$$n = \pm p_1^{e_1} \ldots p_\ell^{e_\ell}.$$

Moreover, this *decomposition* (or *factorization*) is unique, up to reordering of the factors.

Proposition 5.3 (Prime number theorem)**.** Let $n \in \mathbb{N}$, and let us denote by $\pi(n)$ the number of prime numbers smaller than n. Then

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log(n)} = 1.$$

On a computational level, one might think that the problem of determining whether a number is prime or composite and the problem of finding the factorization of said number are close problems. However, the surprising truth is that the second is believed to be much harder than the first! More precisely, there exist efficient algorithms for determining whether a number is prime, but no efficient factorization algorithm is known for numbers that are a product of two large primes, despite decades of huge efforts in finding one.[1]

Sage contains implementations of the best algorithms known for each case. Try increasing the size of the numbers, and observe that the first algorithm is still very fast, but factorization becomes much slower.

```
from sage.misc.prandom import randrange

# Choose a security parameter, which will determine the size of our numbers.
sec_param = 160

### Primality testing
# Pick a random number of bitlength sec_param
n = randrange(2^(sec_param-1),2^(sec_param))
# Run a primality test
%time print(n in Primes())              # Sage contains the class Primes().
                                        # By checking whether n is in Primes(),
                                        # it is actually running a primality test internally.

### Factorization
# Pick two primes of bitlength half of sec_param.
# This is so that their product has bitlength sec_param.
p = random_prime(2^((sec_param/2)-1),2^(sec_param/2))
q = random_prime(2^((sec_param/2)-1),2^(sec_param/2))
# Compute their product
n = p*q
%time print(factor(n))
```

[1] For those with a background in complexity theory, primality is a problem in P and factorization is a problem in NP.

5.2 The euclidean algorithm

With Proposition 5.2 in mind, we observe that two integers a, b can have some factors in common in its prime factorization. This gives rise to the following notion.

Definition 5.3. *Let $a, b \in \mathbb{Z}$ different from 0. The* greatest common divisor *of a and b, denoted by*

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

is the largest positive integer k such that $k \mid a$ and $k \mid b$. Two integers a, b are said to be coprime *(or* relatively prime*) when*

$$\gcd(a, b) = 1.$$

Think about the relation between the notions of primality and coprimality. In particular, observe that being prime is a property of a single integer, whereas being coprime refers to a *pair* of integers.

Exercise 5.2. *Find a pair $a, b \in \mathbb{Z}$ for each of the following cells in this table:*

	a, b prime	*a prime, b composite*	*a, b composite*
a, b coprime			
a, b not coprime			

Computing the greatest common divisor of two integers can be achieved easily using the *Euclidean algorithm*, which we describe next. Let $a, b \in \mathbb{Z}$ different from 0. The following procedure outputs $\gcd(a, b)$:

1. Compute the integer division of a by b, obtaining q, r such that $0 \leq r < b$ and
$$a = bq + r.$$
2. If $r = 0$, then output b. Otherwise, return to the previous step, replacing a by b and b by r.

We show an example for the numbers 375 and 99. We start by performing the integer division of 375 by 99, obtaining

$$375 = 99 \cdot 3 + 78.$$

Since the remainder is not 0, we compute the integer division of 99 by 78, obtaining

$$99 = 78 \cdot 1 + 21.$$

We continue with this process until the remainder is 0:

$$78 = 21 \cdot 3 + 15.$$
$$21 = 15 \cdot 1 + 6.$$
$$15 = 6 \cdot 2 + 3.$$
$$6 = 3 \cdot 2 + 0.$$

Since the remainder is 0, the Euclidean algorithm outputs $\gcd(375, 99) = 3$.

The greatest common divisor satisfies the following property.

Proposition 5.4. Let $a, b \in \mathbb{Z}$ different from 0. There exist $x, y \in \mathbb{Z}$ such that

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

It turns out that we can slightly tweak the Euclidean algorithm to compute the integers x, y in the proposition above. This is called the *extended Euclidean algorithm*. The key idea is to use the Euclidean algorithm to compute the greatest common divisor, and then "walk back" through the computations. We illustrate it with the example of 375 and 99 from above.

We got the sequence of remainders $78, 21, 15, 6, 3, 0$, so the last one before 0, in this case 3, was our greatest common divisor. We proceed by arranging the relations between them obtained above to write each in terms of the previous two. We start with

$$3 = 15 - 6 \cdot 2.$$

We now write 6 in terms of the two previous remainders, 15 and 21:

$$6 = 21 - 15 \cdot 1.$$

Combining these two expressions, we can obtain an expression of 3 in terms of 15 and 21:

$$3 = 15 - (21 - 15 \cdot 1) \cdot 2 = -21 \cdot 2 + 15 \cdot 3.$$

Iterating this process, we can work our way back through the sequence of remainders, until we arrive at the beginning, that is, an expression depending only on 375 and 99:

$$\begin{aligned} 3 &= -21 \cdot 2 + 15 \cdot 3 = -21 \cdot 2 + (78 - 21 \cdot 3) \cdot 3 = \\ &= 78 \cdot 3 - 21 \cdot 11 = 78 \cdot 3 - (99 - 78) \cdot 11 = \\ &= -99 \cdot 11 + 78 \cdot 14 = -99 \cdot 11 + (375 - 99 \cdot 3) \cdot 14 = \\ &= 375 \cdot 14 + 99 \cdot (-53). \end{aligned}$$

Thus, we have found that

$$3 = 375 \cdot 14 + 99 \cdot (-53).$$

From a computational point of view, both versions of the Euclidean algorithm are very efficient, even for large numbers, as you can check with the following Sage code.

```
from sage.misc.prandom import randrange

# Choose a security parameter, which will determine the size of our numbers.
sec_param = 128

# Choose a pair of integers of bitlength sec_param
a = randrange(2^(sec_param-1),2^(sec_param))
b = randrange(2^(sec_param-1),2^(sec_param))

# Euclidean algorithm
%time print(gcd(a,b))

# Extended euclidean algorithm. The three outputs correspond
# to gcd(a,b), and the two numbers x,y such that gcd(a,b)=ax+by.
%time print(xgcd(a,b))
```

5.3 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.

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

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

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.[2]

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.

[2]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 5.4. *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 5.3. *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 5.5. 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 identified with one of these by reducing 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 5.5. *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

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

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 5.5.

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

$$\begin{array}{ll} (i) & a + b \equiv \alpha + b \pmod{n}. \\ (ii) & a \cdot b \equiv \alpha \cdot b \pmod{n}. \\ (iii) & \textit{If } b \geq 0, \textit{ then } a^b \equiv \alpha^b \pmod{n}. \end{array}$$

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

Exercise 5.4. *Prove that the logic* XOR *operation and addition modulo* 2 *are the same operation, by checking the four possible cases.*

We now introduce *modular inverses*. Let $a \neq 0$ be a number, and consider the meaning of b being the inverse of a. Over the real numbers, we say that the inverse of 3 is $1/3$, because

$$3 \cdot \frac{1}{3} = 1.$$

That is, every nonzero number $a \in \mathbb{R}$ has an inverse $1/a \in \mathbb{R}$. Consider now the same idea, but over $\mathbb{Z}$. That is, given $a \in \mathbb{Z}$, is there any other integer $b \in \mathbb{Z}$ such that $ab = 1$? It is clear that, unless $a = \pm 1$, there is no such thing as an "integer inverse". However, the answer changes when we consider $\mathbb{Z}_n$ instead of $\mathbb{Z}$.

Definition 5.6. *Let $n \in \mathbb{N}$, and $a \in \mathbb{Z}_n$. We say that $b \in \mathbb{Z}_n$ is the* inverse of a modulo n* if

$$ab \bmod n = 1.$$

If the inverse of a modulo n exists, we say that a is *invertible*. We also say that a is a *unit*.*

As an example, observe that in $\mathbb{Z}_5$ we have

$$(2 \cdot 3) \bmod 5 = 6 \bmod 5 = 1.$$

Therefore, 2 and 3 are inverses modulo 5. In $\mathbb{Z}_3$, we have

$$(2 \cdot 2) \bmod 3 = 4 \bmod 3 = 1.$$

That is, 2 is its own inverse modulo 3.

Exercise 5.5. *Consider $\mathbb{Z}_4$. Find which of its elements are invertible, and which are not.*

When working with modular residues in Sage, one nice thing is that we can specify that we are working modulo some n, and Sage will take care of the modular reductions for us, without having to specify every time that we want each operation reduced modulo n. In the example below, you can see that, when asked about $a+b$, we get 7 because the operation is automatically reduced modulo 17, since we have specified that we want our operations involving a, b to be reduced modulo n. Similarly, the inverse of a is automatically computed modulo n, otherwise we would obtain $1/11$ instead of 14.

```
n=17
G=Integers(n)
G

a = G(11)
b = G(13)
a,b
a+b
a^(-1)
```

The above examples and exercise highlight that, given some n, some elements of $\mathbb{Z}_n$ have an inverse, and some do not. This motivates the following definition.

Definition 5.7. *Let $n \in \mathbb{N}$. We denote by $\mathbb{Z}_n^*$ the subset of $\mathbb{Z}_n$ formed by all the invertible elements of $\mathbb{Z}_n$. We define the function φ that, on input n, returns the size of $\mathbb{Z}_n^*$. This function is called* Euler's totient (or phi) function.

In Sage, Euler's function on input n can be computed by `euler_phi(n)`.

Proposition 5.7. Let $n \in \mathbb{N}$. Then $\mathbb{Z}_n^*$ is composed of all the elements $a \in \mathbb{Z}_n$ such that are coprime to n.

The value of the totient function, and thus the size of $\mathbb{Z}_n^*$, is determined by the prime factorization of n. More precisely, we have the following result.

Proposition 5.8. Let $n \in \mathbb{N}$. Then:

If $n = p^e$, where p is a prime and $e \in \mathbb{N}$, we have

$$\varphi(n) = (p-1)p^{e-1}.$$

If $n = pq$, where p, q are coprime, then

$$\varphi(n) = \varphi(p)\varphi(q).$$

Exercise 5.6. *Use the result above to deduce a formula for $\varphi(n)$, when n is the product of two different prime numbers.*

To actually find the inverse of an element a modulo n, we can use the extended Euclidean algorithm. The key idea is to run the extended Euclidean algorithm on a and n. The algorithm produces $x, y \in \mathbb{Z}_n$ such that

$$ax + ny = \gcd(a, n).$$

By reducing both sides of the equation above modulo n, we get that

$$ax \equiv \gcd(a, n) \pmod{n}.$$

Due to Proposition 5.7, we know that a will be invertible modulo n if and only if $\gcd(a, n) = 1$. Thus, if this is the case, we would have

$$ax \equiv 1 \pmod{n},$$

which means that x is the inverse of a modulo n. If we find that $\gcd(a, n) \neq 1$, then a is not invertible modulo n.

5.4 Modular arithmetic, but efficient

There are different approaches to actually do computations modulo n. The end result will not change, but some ways involve easier computations than others. As an example, say that we want to compute

$$3^{75} \bmod 191.$$

The straightforward approach is to compute 3^{75}, which is

$$608266787713357709119683992618861307,$$

by performing 74 multiplications by 3, and then perform the division by 191. But clearly this is a lot of work. A better approach is to perform the exponentiation in smaller increments, and reduce modulo 191 before numbers get too big. More precisely, let us write the base-2 expansion of 75:

$$75 = 2^6 + 2^3 + 2^1 + 2^0. \tag{5.1}$$

Then, we can rewrite

$$3^{75} = 3^{2^6} \cdot 3^{2^3} \cdot 3^{2^1} \cdot 3^{2^0}. \tag{5.2}$$

Now observe that, for any $i \in \mathbb{N}$, we have that

$$3^{2^i} = \left(3^{2^{i-1}}\right)^2,$$

and thus each of these terms can be recursively computed from the previous by squaring. We also perform the reductions modulo 191 at each step, to prevent the numbers from blowing-up in size. Note that this reduction can be done due

to point (iii) in Proposition 5.5.

$$
\begin{aligned}
&3^{2^0} \equiv 3 \pmod{191},\\
&3^{2^1} \equiv \left(3^{2^0}\right)^2 \equiv 9 \pmod{191},\\
&3^{2^2} \equiv \left(3^{2^1}\right)^2 \equiv 81 \pmod{191},\\
&3^{2^3} \equiv \left(3^{2^2}\right)^2 \equiv 6561 \equiv 67 \pmod{191},\\
&3^{2^4} \equiv \left(3^{2^3}\right)^2 \equiv (67)^2 \equiv 4489 \equiv 96 \pmod{191},\\
&3^{2^5} \equiv \left(3^{2^4}\right)^2 \equiv (96)^2 \equiv 9216 \equiv 48 \pmod{191},\\
&3^{2^6} \equiv \left(3^{2^5}\right)^2 \equiv (48)^2 \equiv 2304 \equiv 12 \pmod{191}.
\end{aligned}
$$

Now it simply remains to multiply the four factors of equation (5.2). Again, to avoid big numbers, we reduce modulo 191 after each factor is multiplied:

$$
\begin{aligned}
&3^{2^0} \cdot 3^{2^1} \equiv 9 \cdot 81 \equiv 729 \equiv 156 \pmod{191},\\
&\left(3^{2^0} \cdot 3^{2^1}\right) \cdot 3^{2^3} \equiv 156 \cdot 67 \equiv 10452 \equiv 138 \pmod{191},\\
&\left(3^{2^0} \cdot 3^{2^1} \cdot 3^{2^3}\right) \cdot 3^{2^6} \equiv 138 \cdot 12 \equiv 1656 \equiv 128.
\end{aligned}
$$

Therefore, we conclude that

$$3^{75} \bmod 191 = 128 \qquad \text{(or, equivalently, that } 3^{75} \equiv 128 \pmod{191}\text{)}.$$

This algorithm is known as *square-and-multiply*, and the reason behind the name is clear once we take a step back and slightly rewrite our solution. Observe that, from equation (5.1), we directly deduce that the binary expression of 75 is

$$[75]_2 = 1001010.$$

Then, from the base number, in this case 3, and for each bit in $[75]_2$, starting from the right, we

- *Squaring step:* compute the square of the previous power of 3.
- *Multiplication step:* if the bit is 1, multiply the product so far by the new power of 3. Otherwise, skip this step.

Take a moment to review the example above, and convince yourself that it matches the steps described.

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 ℓ, 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.*

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