

EECS475  
Introduction to Cryptography

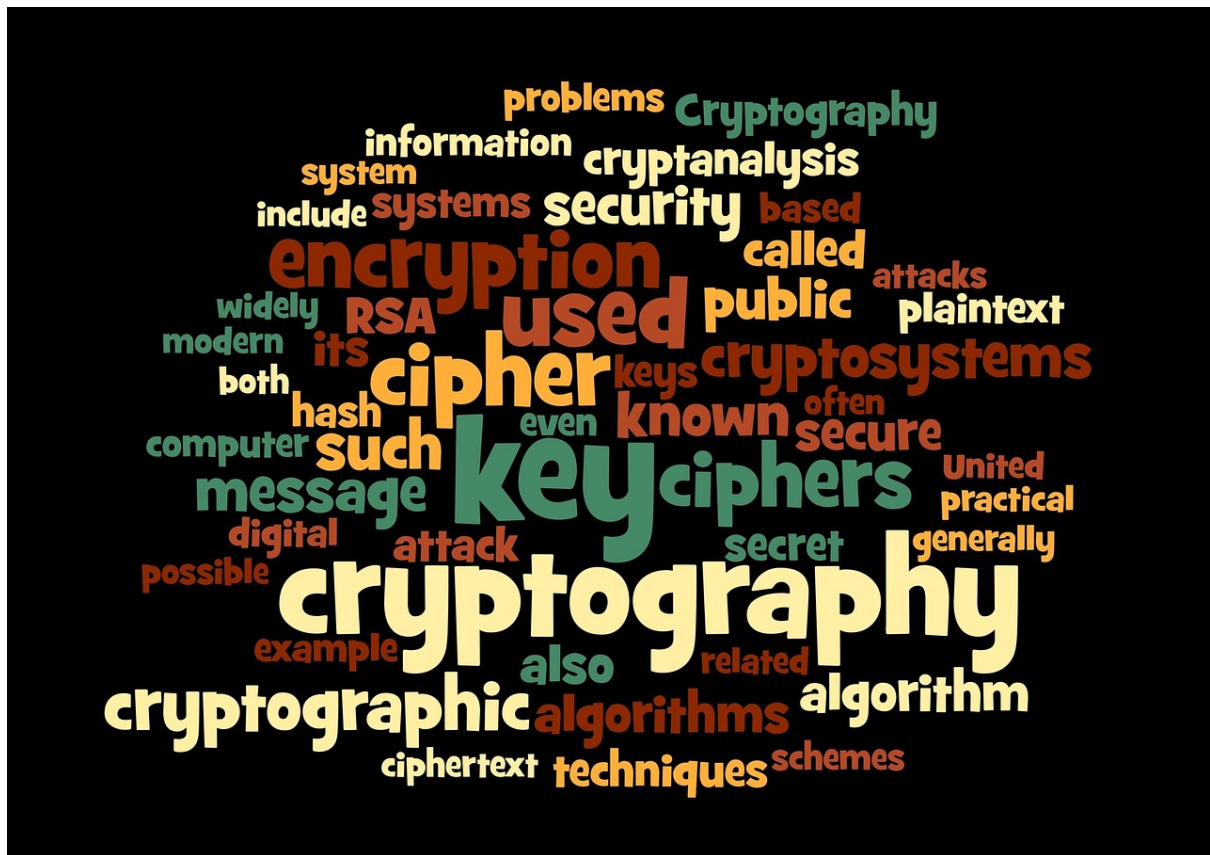
Winter 2023 All Course Members

September 1, 2023

## Abstract

This is an accumulated scribe notes taken by the course members<sup>1</sup> of **EECS475**, an upper-level course taught at University of Michigan by **Mahdi Cheraghchi**. Topics include various historic ciphers, perfect secrecy, symmetric encryption (including pseudorandom generators, stream ciphers, pseudorandom functions/permutations), message authentication, cryptographic hash functions, and public key encryption.

We'll use *Introduction to Modern Cryptography* [KL20] as our main references.



This course is offered in Winter 2023, and the date on the covering page is the last updated time.

<sup>1</sup>Maintain by course staff via selecting/rearranging the submitted scribe notes. For a complete list of contribution, please see [Appendix A](#).

# Contents

<b>1</b>	<b>Introduction to Cryptography</b>	<b>3</b>
1.1	Language of Cryptography . . . . .	3
1.2	Correctness and Security . . . . .	4
1.3	One-Time Pad . . . . .	6
1.4	Computational Security . . . . .	8
<b>2</b>	<b>Message Security</b>	<b>10</b>
2.1	Pseudorandom Generators . . . . .	10
2.2	Eavesdropping Security . . . . .	15
2.3	Chosen Plaintext Attack Security . . . . .	17
2.4	Pseudorandom Functions . . . . .	20
2.5	Modes of Operations and Encryption in Practice . . . . .	24
<b>3</b>	<b>Message Authentication</b>	<b>29</b>
3.1	Security vs. Authenticity . . . . .	29
3.2	Message Authentication Codes . . . . .	31
3.3	Chosen Message Attack Security . . . . .	31
3.4	Authenticated Encryption Scheme . . . . .	37
<b>4</b>	<b>Symmetric Public Key Message Authentication</b>	<b>41</b>
4.1	Cryptographic Hash Family . . . . .	41
4.2	Forming Attacks . . . . .	43
4.3	Merkle-Damgård Construction . . . . .	44
4.4	Arbitrary Length UFCMA MAC . . . . .	45
<b>5</b>	<b>Asymmetric Public Key Message Security</b>	<b>48</b>
5.1	Number Theory . . . . .	49
5.2	Group Theory . . . . .	53
5.3	Diffie-Hellman Key Exchange . . . . .	56
5.4	Public Key Message Encryption . . . . .	58
5.5	RSA Cryptosystem . . . . .	62
<b>6</b>	<b>Asymmetric Public Key Message Authentication</b>	<b>67</b>
6.1	Digital Signatures . . . . .	67
6.2	RSA Signatures . . . . .	69
6.3	Identification Schemes . . . . .	70
<b>7</b>	<b>Post-Quantum and Lattice-Based Cryptography</b>	<b>73</b>
7.1	Post-Quantum Cryptography . . . . .	73
7.2	Lattice-Based Cryptography . . . . .	73
<b>A</b>	<b>Acknowledgement</b>	<b>77</b>
A.1	Winter 2023 . . . . .	77

---

<b>B</b>	<b>More on Group Theory</b>	<b>79</b>
B.1	Groups . . . . .	79
B.2	Isomorphisms . . . . .	84
B.3	Homomorphisms . . . . .	87
B.4	Quotient Groups . . . . .	91

# Chapter 1

## Introduction to Cryptography

### Lecture 1: Introduction

Here are some important links:

4 Jan. 10:30

- [Course information and syllabus](#).
- [Piazza](#).
- [Slack channel](#).
- Or just see [here](#)!

If you have anything want to say, please don't hesitate to email us via [eeecs475-staff@umich.edu](mailto:eeecs475-staff@umich.edu).

### Lecture 2: The Cryptographic Methodology: Modeling encryption

We first see the general picture of cryptography, i.e., the cryptographic methodology:

9 Jan. 10:30

- Form a realistic *model* of the scenario, adjusting as necessary to allow for possibility of solution.
- Precisely define* the desired functionality and security properties of a potential solution.
- Constructing and analyze a solution, *ideally* proving that it satisfies the desired properties.

#### 1.1 Language of Cryptography

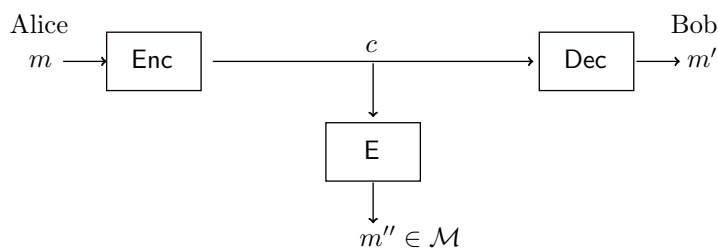
Let's first define some formal notions to help us *speak*:

**Definition 1.1.1 (Plaintext).** Given a space of message  $\mathcal{M}$ ,  $m \in \mathcal{M}$  is called a *plaintext*.

**Definition 1.1.2 (Ciphertext).** Given a space of encrypted message  $\mathcal{C}$  (also called *ciphertext space*),  $c \in \mathcal{C}$  is called a *ciphertext*.

Then, our model will be

- A sender uses an algorithm  $\text{Enc}(\cdot)$  which takes a [plaintext](#)  $m \in \mathcal{M}$  to a [ciphertext](#)  $c \in \mathcal{C}$ .
- A receiver uses an algorithm  $\text{Dec}(\cdot)$  which takes some [ciphertext](#)  $c \in \mathcal{C}$  to a [plaintext](#)  $m' \in \mathcal{M}$ .
- An eavesdropper is represented by an algorithm  $E(\cdot)$  that takes  $c \in \mathcal{C}$  and output  $m''$ .



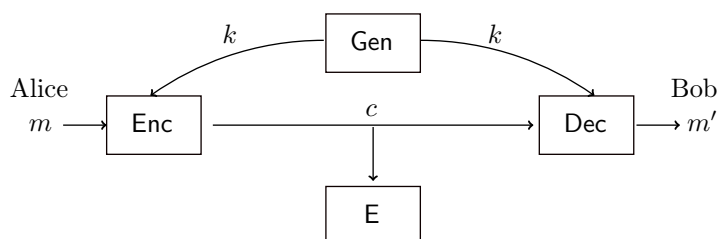
**Remark.** I will use hyperlinks extensively to help you follow the document, but for [plaintext](#) and [ciphertext](#), this is the last time I explicitly reference it: it's used all over the place!

Often time, we will have some sorts of private key  $k$  in the key space  $\mathcal{K}$  which is assumed to be unknown to the eavesdropper, and the key is generated by  $\text{Gen}(\cdot)$ .<sup>1</sup> We summarize the above discussion to the following.

**Definition 1.1.3 (Encryption scheme).** An *encryption scheme* is a tuple  $\Pi = (\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$ .

There are several common [schemes](#), one of which is called [symmetric key encryption](#).

**Definition 1.1.4 (Symmetric key encryption).** *Symmetric key encryption* is the [scheme](#) that a secret key is available to the sender and the receiver in advance, but not the eavesdropper.



## 1.2 Correctness and Security

Ideally, the [scheme](#) we just introduced should have the functionality that we expect, specifically, the [correctness](#) and the security.

### 1.2.1 Correctness

Consider the following.

**Definition 1.2.1 (Correctness).** An [encryption scheme](#)  $\Pi = (\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$  satisfies *correctness* if for all  $m \in \mathcal{M}$ ,

$$\text{Dec}(\text{Enc}(m)) = m.$$

**Remark.** In the case of [symmetric key encryption](#), we incorporate with the key  $k \in \mathcal{K}$  for [correctness](#), i.e., we now require that for all  $k \in \mathcal{K}$  and  $m \in \mathcal{M}$ ,

$$\text{Dec}_k(\text{Enc}_k(m)) = m.$$

### 1.2.2 Shannon Secrecy

[Definition 1.2.1](#) is natural, but what about security? At minimum,  $E(\cdot)$  should not be able to recover  $m$  from  $c$ . Or more generally, we have the following.

<sup>1</sup>We often just sample a uniform random key  $k$  from  $\mathcal{K}$  by invoking  $\text{Gen}$ , hence often time there is no explicit input to  $\text{Gen}$ .

**Definition 1.2.2** (Kerckhoff's principle). The *Kerckhoff's principle* states that an encryption scheme should remain secure even if all its algorithm are known to the public or attacker.

**Remark.** Kerckhoff's principle might be awkward at first, but this is an essential reason of the success of the modern cryptography.

But this description is vague, and we want a precise, mathematical definition of security!

## Lecture 3: Shannon Secrecy and Perfect Secrecy

**As previously seen.** We have seen one crucial property of a desired scheme, i.e., the correctness.

11 Jan. 10:30

We now want to discuss the security aspect. The idea is quite simple.

**Intuition.** Seeing the ciphertext should give the eavesdropper no information about the probability distribution of the message space.

This leads to the following.

**Definition 1.2.3** (Shannon secrecy\*). An encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *Shannon secret* if for any message distribution  $\mathcal{D}$  over  $\mathcal{M}$ , and any fixed  $\overline{m} \in \mathcal{M}$  and  $\overline{c} \in \mathcal{C}$ ,

$$\Pr_{\substack{m \leftarrow \mathcal{D} \\ k \leftarrow \text{Gen}}} (m = \overline{m} \mid \text{Enc}_k(m) = \overline{c}) = \Pr_{\substack{m \leftarrow \mathcal{D} \\ k \leftarrow \text{Gen}}} (m = \overline{m}).$$

**Notation.** For a variable with an overline, e.g.,  $\overline{m}$ , it means that it's fixed.

**Remark.** To interpret Definition 1.2.3, we see that the left-hand side is the *posterior* probability after seeing the ciphertext  $\overline{c}$ , while the right-hand side is a *priori* distribution.

In fact, Definition 1.2.3 can be difficult to work with. Therefore, we will derive an equivalent but more convenient definition. Starting with Shannon secrecy\*, by the definition of conditional probability we have that

$$\begin{aligned} \Pr_{m,k} (m = \overline{m} \mid \text{Enc}_k(m) = \overline{c}) &= \frac{\Pr_{m,k} (m = \overline{m} \wedge \text{Enc}_k(\overline{m}) = \overline{c})}{\Pr_{m,k} (\text{Enc}_k(m) = \overline{c})} \\ &= \frac{\Pr_{m,k} (m = \overline{m} \wedge \text{Enc}_k(\overline{m}) = \overline{c})}{\Pr_{m,k} (\text{Enc}_k(\overline{m}) = \overline{c})} = \Pr_m (m = \overline{m}) \times \frac{\Pr_k (\text{Enc}_k(\overline{m}) = \overline{c})}{\Pr_{m,k} (\text{Enc}_k(m) = \overline{c})}, \end{aligned}$$

where the last equality is from the independence of  $m$  and  $k$ .<sup>2</sup> We know the ratio of the fraction should equal 1 if we assume Definition 1.2.3, leading to an equivalent definition of Shannon secrecy\*:

**Definition 1.2.4** (Shannon secrecy). An encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *Shannon secret* if for any message distribution  $\mathcal{D}$  over  $\mathcal{M}$ , and any fixed  $\overline{m} \in \mathcal{M}$  and  $\overline{c} \in \mathcal{C}$ ,

$$\Pr_{k \leftarrow \text{Gen}} (\text{Enc}_k(\overline{m}) = \overline{c}) = \Pr_{\substack{m \leftarrow \mathcal{D} \\ k \leftarrow \text{Gen}}} (\text{Enc}_k(m) = \overline{c}). \quad (1.1)$$

**Note.** This is trivial when  $\Pr(m = \overline{m}) = 0$ , so  $\overline{m} \in \text{supp}(\mathcal{D})$  should hold true.

### 1.2.3 Perfect Secrecy

If we take a particular distribution  $\mathcal{D}$ , such as the uniform distribution (where all outcomes are equally likely), then Shannon secrecy means that the probability that we get a particular outcome is the same for all possible messages  $\overline{m}$ .

<sup>2</sup>The key is naturally independent of the message.

**Remark.** The right-hand side of Equation 1.1 does not rely on  $\bar{m}$  at all.

This observation leads to the following.

**Definition 1.2.5 (Perfect secrecy).** An encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *perfect secret* if for all  $m_0, m_1 \in \mathcal{M}$  and for all  $\bar{c} \in \mathcal{C}$ ,

$$\Pr_{k \leftarrow \text{Gen}}(\text{Enc}_k(m_0) = \bar{c}) = \Pr_{k \leftarrow \text{Gen}}(\text{Enc}_k(m_1) = \bar{c}).$$

**Intuition.** Basically, the probability distributions of  $\text{Enc}_k(m_0)$  and  $\text{Enc}_k(m_1)$  should be the same.

**Lemma 1.2.1.** Perfect secrecy is equivalent to Shannon secrecy.

**Proof.** The way we define perfect secrecy suggests that Shannon secrecy implies perfect secrecy. Conversely, we can show that perfect secrecy implies Shannon secrecy. Using the law of total probability,

$$\Pr_{m,k}(\text{Enc}_k(m) = \bar{c}) = \sum_{m' \in \mathcal{M}} \Pr(m = m') \times \Pr(\text{Enc}_k(m') = \bar{c}).$$

From the definition of perfect secrecy, we know that the probabilities for any two messages are the same, so we can replace  $m'$  with  $\bar{m}$ , i.e., we further have

$$\Pr_{m,k}(\text{Enc}_k(m) = \bar{c}) = \underbrace{\sum_{m' \in \mathcal{M}} \Pr(m = m')}_1 \times \Pr_k(\text{Enc}_k(\bar{m}) = \bar{c}) = \Pr_{m,k}(\text{Enc}_k(\bar{m}) = \bar{c})$$

which is the definition of Shannon secrecy. ■

## 1.3 One-Time Pad

It's all great, if we can actually design an scheme which achieves Shannon secrecy.

**Problem.** Can we achieve Shannon secrecy?

**Answer.** Yes, we can! The one-time pad, also known as the Vernam Cipher, achieves Shannon secrecy. ⊛

**Definition 1.3.1 (One-time pad).** The one-time pad scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  defined over  $\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0, 1\}^\ell$  is given by

- $\text{Gen}(\cdot)$ : chooses a key  $k \in \{0, 1\}^\ell$  uniformly at random.
- $\text{Enc}_k(m)$ : outputs  $c = m \oplus k \in \{0, 1\}^\ell$ .<sup>a</sup>
- $\text{Dec}_k(c)$ : outputs  $\bar{m} = c \oplus k \in \{0, 1\}^\ell$ .

<sup>a</sup> $\oplus$  is the bit-wise XOR.

**Lemma 1.3.1.** One-time pad is correct.

**Proof.** Let  $k, m \in \{0, 1\}^\ell$  be arbitrary. Then,

$$\text{Dec}_k(\text{Enc}_k(m)) = (m \oplus k) \oplus k = m \oplus \underbrace{(k \oplus k)}_{0^\ell} = m.$$

■



## Lecture 4: One-time Pad, Limitations of Perfect Secrecy

We now show that **one-time pad** is secure.

18 Jan. 10:30

**Theorem 1.3.1.** **One-time pad** is **perfectly secret**.

**Proof.** Let  $\bar{m} \in \{0, 1\}^\ell$  be any fixed plaintext,  $k$  be a randomly generated key by  $\text{Gen}(\cdot)$ . For any  $\bar{c} \in \{0, 1\}^\ell$

$$\Pr_k(\text{Enc}_k(\bar{m}) = \bar{c}) = \Pr_k(\bar{m} \oplus k = \bar{c}) = \Pr_k(k = \bar{m} \oplus \bar{c}) = 2^{-\ell},$$

where second equality comes from the identity  $\bar{m} \oplus \bar{m} \oplus k = \bar{m} \oplus \bar{c}$  and  $a \oplus a = 0$ , and the last equality follows from the fact that  $k$  is random and the  $\bar{m} \oplus \bar{c}$  is fixed. In all, this means that for any fixed  $m_0, m_1$ , the probability is always equal to  $2^{-\ell}$ , hence it's **perfectly secret**. ■

### 1.3.1 Problems with the One-Time Pad

The **one-time pad** seems to be the perfect code as it satisfies **perfect secrecy**. However, **one-time pad** is rarely used in real life, and neither is the **perfect secrecy** criterion. In fact, there are several problems that come with the **one-time pad**.

- (a) The length of the key must be equal to the message length. This means that if we have to encode a long message, we will need to transmit a key that is as long as this message, which is a waste of space.
- (b) The key has to be truly randomly generated, otherwise the proof wouldn't work.
- (c) The key couldn't be reused, which is why the **one-time pad** is *one time*. For instance, if we have key  $k$ , message  $m_1$  and  $m_2$ , let

$$c_1 = \text{Enc}_k(m_1) = m_1 \oplus k, \quad c_2 = \text{Enc}_k(m_2) = m_2 \oplus k,$$

then if we calculate  $c_1 \oplus c_2$ , the  $k$ 's cancel, and we'll figure out  $m_1 \oplus m_2$ , which is bad. The figure below illustrates this scenario:

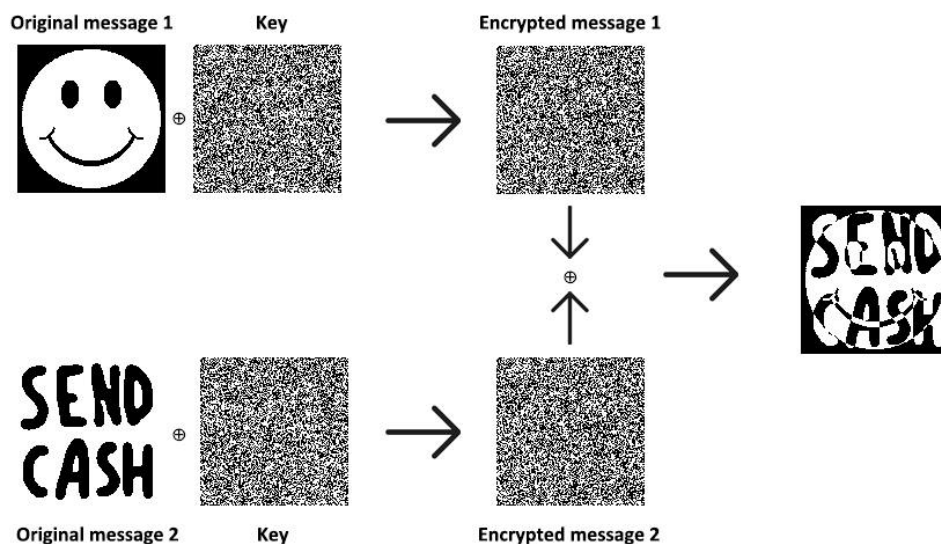


Figure 1.1: Recover  $m_1 \oplus m_2$ .

Can we do better than **one-time pad**, i.e., satisfy the **perfectly secret** requirement without the above limitations? For the first bullet point, sadly, the answer is no.

**Theorem 1.3.2.** For any [perfect encoding scheme](#), we must have  $|\mathcal{K}| \geq |\mathcal{M}|$ , i.e., to achieve [perfect secrecy](#), we have to have at least as many potential keys as messages.

**Proof.** For any fixed message  $m_0 \in M$  and any fixed key  $k_0$ , consider the set of all possible decryption outcomes of  $m_0$ 's encoding under  $k_0$ , i.e.,

$$D := \{\text{Dec}_k(\text{Enc}_{k_0}(m_0)) : k \in \mathcal{K}\}.$$

Since we could have at most  $k$  outcomes,  $|D| \leq |\mathcal{K}|$ . Suppose  $|\mathcal{K}| < |\mathcal{M}|$ , then  $|D| \leq |\mathcal{K}| < |\mathcal{M}|$ , which implies that  $\exists m_1 \in \mathcal{M} \setminus D$ . But then  $\Pr_k(\text{Enc}_k(m_0) = \bar{c}) > 0$ , which is not equal to  $\Pr_k(\text{Enc}_k(m_1) = \bar{c}) = 0$  by definition of  $m_1$  and [correctness](#). ■

## 1.4 Computational Security

Rather than requiring the [encryption scheme](#) to be [perfectly secret](#), we wish to look for some kind of [encryption schemes](#) that are secure enough – more specifically, secure against eavesdroppers that are efficient.

**Intuition.** Recall problem (a), there we said that the key is too long. The upside is that it gives us more possible keys when the eavesdropper is making a random guess – but since computational power is limited, we wouldn't need that big a key space for it to be infeasible for an eavesdropper.

Let's first have a look at the sizes of key spaces and their corresponding computing time to give us an idea about which attacks are *feasible*:

- $2^{30}$  (1 billion): totally feasible;
- $2^{40}$ : a few minutes on a PC;
- $2^{60}$ : a few minutes on a supercomputer;
- $2^{80}$ : 1 to 2 years on a supercomputer;
- $2^{100}$ : 1 million years on a supercomputer;
- $2^{128}$ : 1 trillion years after 200 more years of [Moore's law](#). This is really infeasible;
- $2^{256}$ : more than the number of atoms ( $\approx 2^{240}$ ) in the universe.

## Lecture 5: Computational Security and PRGs

### 1.4.1 Concrete Security

23 Jan. 10:30

**As previously seen.** The [perfect secrecy](#) is stronger than what we actually need. What is sufficient security in practice, and how to formally define it?

One approach is to use *concrete security*, which deals with the exact setup of the notion of feasibility (number of operations, architecture, etc.).

**Intuition.** An adversary can have a *tiny* chance of security violation, where this tiny chance is much smaller than the chance that something goes wrong. And since the adversary can boost chance of success by repeating or doing more work, so we really measure the runtime over the probability of success as the *cost* of the attack, i.e.,

$$\text{cost} = \frac{\text{run time}}{\Pr(\text{success})}.$$

In other words, we wish to maximize the cost of an adversary's attack.

**Remark.** These ideas of thinking about feasibility and costs of an attacker diverges from Shannon's in that he did not concern himself with understanding the computational power of attackers, rather his theory only looked at mathematical foundation.

### 1.4.2 Asymptotic Security

However, defining the equation above requires us to quantify the duration of an attack algorithm. Since numbers can get messy and involve knowledge of OS/computer architecture, we abstract it away in this course and focus on asymptotic behavior instead.

**As previously seen.** An algorithm efficient if it runs in polynomial time rather than exponential.

We introduce a *security parameter*  $n$  which quantifies our level of security. This parameter is selected by the user for different use cases.

**Intuition.** Logically, it wouldn't make sense to use the same security parameter for TV streaming services as nuclear launch codes.

**Example.** Selecting 128 vs 256 bits for the key length of [OTP](#).

However, with higher security we also increase the computation time of the legitimate communicating parties. Thus, the problem becomes finding a security level which makes legitimate communication reasonable while making adversary attacks difficult.

**Example.** Legitimate entity runs fast, so rapid communication is possible, efficient in terms of  $n$

- $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ , etc.

Attacker can afford to take longer, but still feasible:

- $O(n^c)$  where  $c$  is some large constant.

In essence, we want the scenario where the attack spends a feasible amount of time but can only gain a *tiny advantage*.

# Chapter 2

## Message Security

In this chapter, we're going to focus on [schemes](#) that consider asymptotic security, where we naturally have some kinds of *adversaries* who is trying to attack the system.

### 2.1 Pseudorandom Generators

[Pseudorandom generators](#), noted as *the most important notion in cryptography*, are used in virtually every cryptosystem either explicitly or under the hood. The motivation is that, recalling that a major headache of [OTP](#) requires [the key being the same length with messages](#), and we don't have access to this kind of random source, so we want to generate a “random” key with small true randomness.

Moreover, the interesting thing is that, the “advantage” we care about relates directly to the quality of the randomness we use!

#### 2.1.1 Negligible Functions

We keep mentioning the word *tiny advantage*, and we now give a formal way to model it by using the notion of [negligible functions](#).

**Definition 2.1.1** (Negligible). A function  $\epsilon(n)$  is *negligible*, written as  $\epsilon(n) = \text{negl}(n)$ , if

$$\epsilon(n) = o(n^{-c})$$

for all constant  $c > 0$ .

Equivalently,

$$\lim_{n \rightarrow \infty} n^c \cdot \epsilon(n) = 0.$$

**As previously seen** (Little-*o*). Recall that  $f(n) = o(g(n))$  if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

**Example.**  $\epsilon(n) = \frac{1}{2^{n/4}}$  is [negligible](#).

**Proof.** Since  $\lim_{n \rightarrow \infty} n^c \times \epsilon(n) = 0$  for any positive constant  $c$  by [L'Hôpital's rule](#). ⊛

**Example.**  $\epsilon(n) = \frac{1}{n^5}$  is not [negligible](#).

**Proof.** Since  $\lim_{n \rightarrow \infty} n^c \times \epsilon(n) = \infty$  for  $c > 5$ . ⊛

We then use the notion of [negligible](#) to model *tiny advantage* and construct a generalized template for computational security.

**Definition 2.1.2 (Computational security).** Every randomized polynomial-time attacker has only negligible “advantage” when attacking our cryptographic system.

Now, there’s one fundamental question to ask: how should we define “advantage” formally? This leads to the next big topic, [pseudorandom generators](#).

## Lecture 6: PRGs, Distinguishing Games, and Stream Ciphers

### 2.1.2 Pseudorandom Generators

25 Jan. 10:30

We’re now ready to define randomness and [pseudorandomness](#), and dive deep into the difference between these two concepts. We must also quantify [pseudorandomness](#):

- How random is [pseudorandom](#)?
- Is there a way to predict a [pseudorandom](#) outcome better than a blind guess?
- Is there a feasible algorithm that can improve our prediction of the outcome?

To answer these questions, we considered methods of “differentiating” [pseudorandom](#) and random, as well as the efficiency of this operation. A [pseudorandom generator](#) is an efficient, deterministic algorithm  $G$  that takes in a seed  $s \in \{0, 1\}^n$  for some seed length  $n$  and generates a [pseudorandom](#) output  $G(s) \in \{0, 1\}^{\ell(n)}$ , where  $\ell(n) > n$  is some extension of the initial seed.

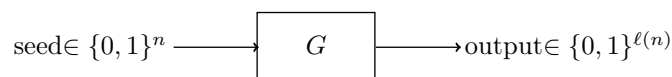


Figure 2.1:  $G$  generates a [pseudorandom](#) bit-string of length  $\ell(n)$  given a truly random seed of length  $n$ .

Let’s first give the definition of [PRGs](#).

**Definition 2.1.3 (Pseudorandom generator).** A *pseudorandom generator (PRG)*  $G$  with [expansion](#)  $\ell(n)$  is a deterministic, polynomial time algorithm satisfying [pseudorandom](#) property.

**Notation (Expansion).** The *expansion* of a [PRG](#)  $G$  is the length of its output, i.e.,  $|G(s)| = \ell(n) > n$  for all  $s \in \{0, 1\}^n$ .

We see that we haven’t defined the notion of [pseudorandom](#), since to define it we need some additional machinery. Before doing so, we see the following.

**Intuition.** Informally, [pseudorandom](#) means that for uniform random  $s \in \{0, 1\}^n$ ,  $G(s)$  looks random (like a uniform  $\ell(n)$ -bit string) to all feasible attackers.

From [Definition 2.1.3](#), new questions arise: What does it mean to look random? What defines a feasible attacker? We can answer these questions by defining a distinguisher that attempts to distinguish between a pseudorandom and random input, and the resulting “advantage” from this distinguisher’s output.

### 2.1.3 Distinguishers

A [distinguisher](#) takes in  $y \in \{0, 1\}^{\ell(n)}$  and outputs a decision bit that determines whether  $y$  was randomly or pseudorandomly generated.

**Definition 2.1.4 (Distinguisher).** A *distinguisher*  $D$  is a polynomial time (potentially probabilistic) algorithm such that

- takes in inputs  $1^n$  and  $y = G(s)$  for some uniformly random  $s \leftarrow \{0, 1\}^n$  (“real world”), or

some uniformly random  $y \leftarrow \{0, 1\}^{\ell(n)}$  (“ideal world”);

- outputs a decision bit (0/1) for whether  $y$  was pseudorandom or random.

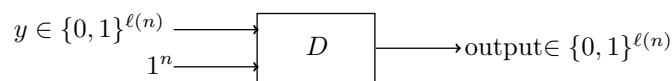


Figure 2.2:  $D$  distinguishes a bit-string of length  $\ell(n)$  given the length  $n$  (by  $1^n$ ).

**Note.** The  $1^n$  input is often also implicit, since it tells  $D$  the length of the seed (which is publicly known).  $n$  is also known as the security parameter because it determines the length of the random seed, and therefore, also the randomness of the generated result.

A **distinguisher** can potentially be probabilistic, meaning it can make random choices. In this way, the output of the distinguisher has a distribution based on the randomness of the inputs, as well as the internal randomness of the algorithm. We want to know the probability that the distinguisher outputs 1 or accepts when given random and pseudorandom inputs. Considering these probabilities, we can then define **advantage**, a way to quantify **pseudorandomness**.

**Definition 2.1.5 (Advantage).** Given a **distinguisher**  $D$ , its *advantage*  $\text{Adv}_G(D)$  in distinguishing the “real world” ( $y \leftarrow G(s)$  for uniformly random  $s$ ) and the “ideal world” ( $y \leftarrow \{0, 1\}^{\ell(n)}$  is uniformly random) is given by

$$\text{Adv}_G(D) := \left| \Pr_{s \leftarrow \{0, 1\}^n} (D(1^n, G(s)) = 1) - \Pr_{y \leftarrow \{0, 1\}^{\ell(n)}} (D(1^n, y) = 1) \right|.$$

**Intuition.** The first probability is over the “real world” and the second probability is over the ideal world,” and both probabilities are also over the randomness of the distinguisher  $D$ , since  $D$  can be probabilistic algorithm.

For a **pseudorandom generator**, we want our **advantage** to be small and as close to 0 as possible, which can be characterize by the notion of **negligible**.

**Definition 2.1.6 (Fool).** If the **advantage** of a **distinguisher**  $D$  against a **PRG**  $G$  is **negligible**, then  $D$  is *fooled* by  $G$ , or  $G$  *fools*  $D$ .

When we have a **generator**  $G$ , we want it to **fool** all **distinguishers**  $D$ . In other words, to be **secure**, the **generator**  $G$  must **fool** all **distinguishers**  $D$ .

**Definition 2.1.7 (Pseudorandom).** A **PRG**  $G$  is *pseudorandom* (or *secure*) if for all (randomized) polynomial time<sup>a</sup> **distinguishers**  $D$ , we have  $\text{Adv}_G(D) = \text{negl}(n)$ .

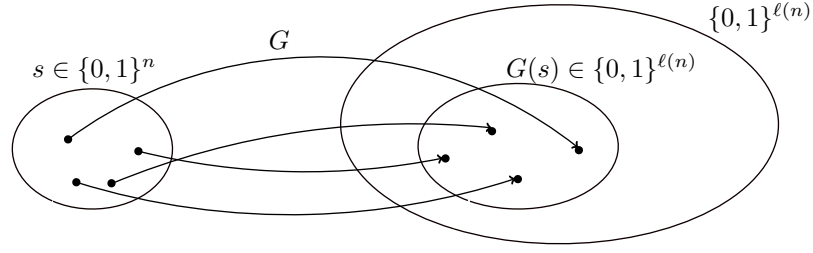
<sup>a</sup>Or probabilistic polynomial time.

A natural question from **Definition 2.1.7** arises.

**Problem.** Why must  $D$  be efficient?

**Answer.** Otherwise, we can construct an attack that essentially checks if  $y$  is the same as  $G(s)$  for every possible seed  $s$ . Note that according to the **Kerckhoff principle** (open source assumption), the length of the seed  $n$  and the **generator** algorithm  $G$  are both public knowledge. ⊛

Consider any **generator**  $G$  that maps  $s \in \{0, 1\}^n$  to some subset of  $\{0, 1\}^{\ell(n)}$ :

Figure 2.3: Venn diagram of the domain and the image of  $G$ .

Then, let's define a **distinguisher**  $D$  that accepts  $y \in \{0,1\}^{\ell(n)}$  if and only if there exists  $s \in \{0,1\}^n$  such that  $G(s) = y$ , i.e., the image of  $G$ ,  $\text{Im } G$ .

We know that  $\ell(n) > n$ , so  $\ell(n) \geq (n+1)$  and therefore  $2^{\ell(n)} \geq 2^{n+1} > 2^n$ . So, the size of  $\{0,1\}^{\ell(n)}$  is at least two times the size of what's mapped to by  $s \in \{0,1\}^n$ . From here, we see that

$$\Pr_{s \leftarrow \{0,1\}^n} (D(1^n, G(s)) = 1) = 1$$

since  $D$  accepts if  $G$  maps some  $s$  to  $G(s)$ , and also

$$\Pr_{y \leftarrow \{0,1\}^{\ell(n)}} (D(1^n, y) = 1) = \frac{|\text{Im } G|}{|\{0,1\}^{\ell(n)}|} = \frac{2^n}{2^{\ell(n)}} \leq \frac{2^n}{2^{n+1}} = \frac{1}{2}.$$

So, we can determine a lower bound for the **advantage** of  $D$ , which is

$$\text{Adv}_G(D) = \left| \Pr_{s \leftarrow \{0,1\}^n} (D(1^n, G(s)) = 1) - \Pr_{y \leftarrow \{0,1\}^{\ell(n)}} (D(1^n, y) = 1) \right| \geq \left| 1 - \frac{1}{2} \right| = \frac{1}{2},$$

which is non-negligible. However, the problem here is that  $D$  is not efficient. For this procedure,  $D$  must check every seed  $s \in \{0,1\}^n$ , which includes  $2^n$  possible bit strings. This wouldn't be efficient since it's exponential with respect to the bit string length. So, for practicality, we want to consider feasible algorithms since this would be a trivial, infeasible algorithm that produces a non-negligible advantage for any generator  $G$ .

### 2.1.4 Existence of PRGs

Continuing the discussion of complexity, another question is whether **PRGs** exist. Although it seems trivial, we actually don't know whether **PRGs** exist! This boils down to whether  $P$  equals  $NP$ .

**Proposition 2.1.1.** If **PRGs** exist, then  $P \neq NP$ .

**Proof.** Firstly, observe the following.

**Claim.** **PRGs** are efficiently verifiable, i.e., in  $NP$ .

**Proof.** Since  $G$  is open source, we can determine whether  $y \in \{0,1\}^{\ell(n)}$  was generated by a seed  $s \in \{0,1\}^n$  (the certificate) by simply running  $G(s)$ , hence **PRGs** must be in  $NP$ .  $\otimes$

Therefore, if **PRGs** exist, i.e., there is no efficient algorithm to find  $s$  given  $y$ , so  $P \neq NP$ .  $\blacksquare$

Solving  $P = NP$  or not is out of our reach currently, and this problem is hunting us for centuries! However, even though we don't know whether **PRGs** really exist or not, there are heuristics for **pseudorandom generators**. There are high quality implementations of **pseudorandom generators**.

**Example.** `/dev/random` in Unix, `CryptGenRandom()` in Windows, and `SecureRandom()` in Java are all high quality implementations of **PRGs**.



There is a distinction between the above implementation and one like `rand()`, since `rand()` is easy to break even though it is sufficiently random for algorithms. The repetition in `rand()` is illustrated below.

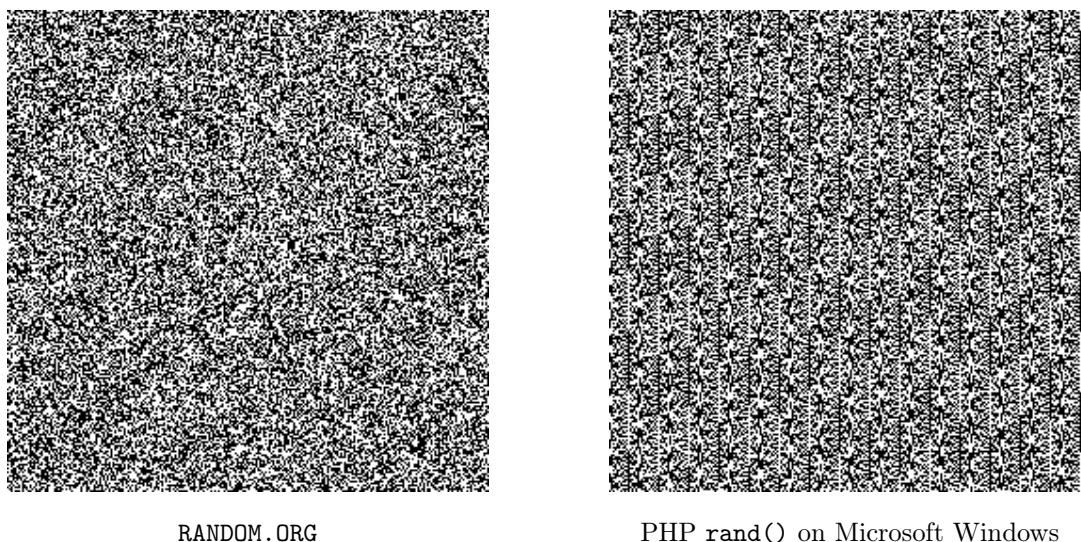


Figure 2.4: Comparison between the patterns of generated pseudo-randomness.<sup>1</sup>

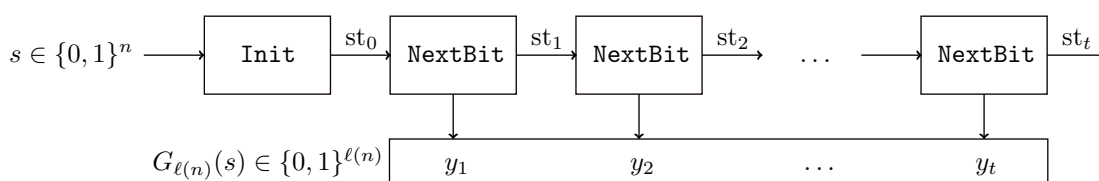
### 2.1.5 Stream Cipher

**Stream ciphers** are equivalent to **PRGs**, and can essentially be seen as a **PRG** on demand. Suppose we have a **PRG** that takes  $n$  bits and produces  $\ell(n)$  bits, what if we don't know what  $\ell(n)$  is? **Stream cipher** can be abstracted as a button you push, where each time you push the button, you get one more **pseudorandom** bit. You can then continue to push the button until you have the length you want.

**Definition 2.1.8 (Stream cipher).** A *stream cipher* is **PRG**-like procedure that generates **pseudorandom** bits with two functions, **Init** and **NextBit**.

- **Init**( $s$ ): takes seed  $s$  and outputs some state data  $st_0$ .
- **NextBit**( $st_i$ ): takes in state data and outputs the next state  $st_{i+1}$  and a bit  $y_{i+1}$ .

Compiling all bits will be a **pseudorandom** bit-string of length  $\ell(n)$  (essentially  $G_{\ell(n)}(s)$ ).



## Lecture 7: Stream Ciphers and Eavesdropping Security

**Intuition.** The purpose of having a **stream cipher** is to have a **PRG**-like procedure that generates **pseudorandom** bits as needed, or a **pseudorandom generator on demand**.

30 Jan. 10:30

If implemented in C++, the  $st_i$  would be set as a global variable to keep track of what the seed has evolved to and additionally use this variable in our changing functions. Note this process is stateful.<sup>2</sup> The **stream cipher** is **pseudorandom** if for every  $\ell(n) = \text{poly}(n)$ , the function  $G_\ell$  defined above is **pseudorandom**.

<sup>1</sup>Source: [random.org](http://random.org)

<sup>2</sup>One that uses the last updated state to produce a new state, i.e., one that remembers what was done last



**Note.** To get a **stream cipher** from a **PRG**, take **PRG** and compose with itself for as many times as needed.

## 2.2 Eavesdropping Security

In this section, we aim to address the key length problem in **perfect secrecy**, as depicts by **Theorem 1.3.2**.

**As previously seen.** **Perfect secrecy** dictates that

- (a) Given a ciphertext, you cannot determine the **plaintext**.
- (b)  $c_0$  and  $c_1$  will have the exact same statistical distribution.
- (c) If a shorter key than the message length is used, **perfect secrecy** is impossible.

Now, we try to relax to only require that given  $c_0$  or  $c_1$ , it's hard to distinguish between the two cases.

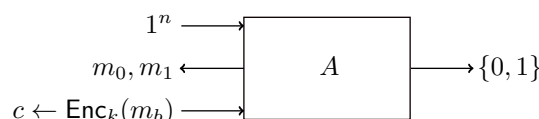
### 2.2.1 Eavesdropping Game

To formulate the above idea, we design the so-called **EAV game**, which requires that any adversary  $A$  is unable to distinguish between  $\text{Enc}_k(m_0)$  and  $\text{Enc}_k(m_1)$  given two plaintexts  $m_0$  and  $m_1$ .

**Definition 2.2.1 (Eavesdropping game).** The *eavesdropping game* for an adversary  $A$  against an **encryption scheme**  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  proceeds as follows:

1.  $A$  is given the security parameter;
2.  $A$  outputs two messages,  $m_0, m_1$  with  $|m_0| = |m_1|$ ;
3. the game generates  $c \leftarrow \text{Enc}_k(m_b)$  for  $b \in \{0, 1\}^a$  with  $k \leftarrow \text{Gen}$ ;
4.  $A$  is given  $c$ ;
5.  $A$  outputs a decision bit.

<sup>a</sup>Indicating which “world”  $A$  is in.



### 2.2.2 Eavesdropping Secrecy

Consider the new notion of **advantage** against the **EAV game**.

**Definition 2.2.2 (Advantage).** Given an adversary  $A$  in an **EAV game**, the *advantage*  $\text{Adv}_{\Pi}^{\text{EAV}}(A)$  in distinguishing “world 0” and the “world 1” is given by

$$\text{Adv}_{\Pi}^{\text{EAV}}(A) := |\Pr(A \text{ in “world 1” outputs 1}) - \Pr(A \text{ in “world 0” outputs 1})|.$$

Naturally, we have the following.

**Definition 2.2.3 (Eavesdropping secrecy).** An **encryption scheme**  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *eavesdropping secure* if for every probabilistic polynomial time adversary  $A$ , the **advantage** is **negligible**.

**Intuition.** Probabilities of adversary accepting on both  $c_0 = \text{Enc}_k(m_0)$  and  $c_1 = \text{Enc}_k(m_1)$  should

be similar or the same (the adversary should not know what is going on).<sup>a</sup>

<sup>a</sup>We have seen the similar definition in the assignment:  $A$  is trying to get some information about  $m_b$  from  $c_b$ !

**Remark.** We assume the length of both messages are the same, i.e.,  $|m_0| = |m_1|$  because otherwise  $A$  can distinguish messages by length.

**Note.** If we change  $\text{negl}(n)$  to 0 and remove the polynomial time requirement from Definition 2.2.3, then we get back the perfect secrecy.

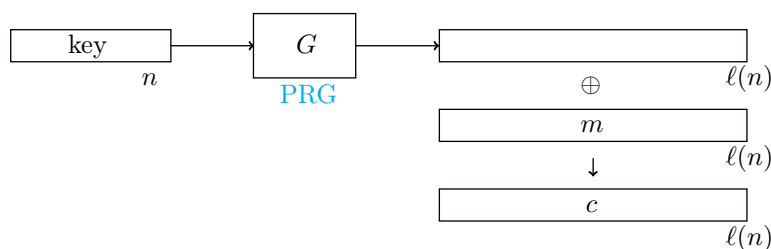
### 2.2.3 Eavesdropping Secure Schemes

We are now interested in constructing an **EAV-secure scheme**. The idea is simple, we start from **one-time pad**:

- use **pseudorandom generators** to create a short key;
- stretch the short key to be an effective key and run **one-time pad**.

With  $|m| = \ell(n)$ , and a **PRG**  $G$ , we define our **scheme**  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  by

- $\text{Gen}(1^n)$ : outputs a uniformly random  $k \leftarrow \{0, 1\}^n$ ;
- $\text{Enc}_k(m)$ : given  $m \in \{0, 1\}^{\ell(n)}$ , outputs  $c = m \oplus G(k)$ ;
- $\text{Dec}_k(c)$ : given  $c \in \{0, 1\}^{\ell(n)}$ , outputs  $c \oplus G(k)$ .



**Claim.**  $\Pi$  is **correct**.

**Proof.** On a broad scale, this is just **one-time pad** ⊗

As for **EAV secrecy**, consider proving the contrapositive, i.e., if  $\Pi$  is not **EAV secure**, an adversary can win the **eavesdropping game**. By using this adversary to break the security of the **PRG** since the **security** of  $\Pi$  relies on the security of the **PRG**.

**Intuition.** We recognize that this is just a proof by reduction! Assume some adversary can win this **game**. Incorporate this adversary into a **game** that can break the **PRG**.

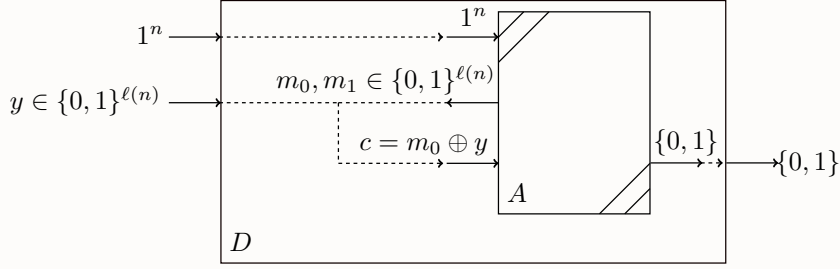
## Lecture 8: EAV Construction Analysis and CPA Security

**Theorem 2.2.1.** If  $G$  is a (secure) **PRG**, then  $\Pi$  is **EAV secure**.

**Proof.** Let  $A$  be any probabilistic polynomial time **EAV-attacker** against  $\Pi$ . We use  $A$  to build a probabilistic polynomial time distinguisher  $D(1^n, y)$  for  $y \in \{0, 1\}^{\ell(n)}$  against  $G$  by

- run  $A(1^n)$  and receive messages  $m_0, m_1 \in \{0, 1\}^{\ell(n)}$ ;
- give  $c = m_0 \oplus y$  to  $A$ ;<sup>a</sup>
- output  $A$ 's verdict.

1 Feb. 10:30



We see that  $D$ 's **advantage** is

$$\text{Adv}_G^{\text{PRG}}(D) = \left| \Pr_{k \leftarrow \{0,1\}^n} (D(1^n, G(k)) = 1) - \Pr_{y \leftarrow \{0,1\}^{\ell(n)}} (D(1^n, y) = 1) \right|,$$

which is just

$$\text{Adv}_G^{\text{PRG}}(D) = \left| \Pr_k (A \text{ in "world 0" accepts}) - \Pr(A \text{ in random } c \text{ accepts}) \right|.$$

Recall that we want to bound the **advantage** of  $A$  in the **EAV game**, which is

$$\text{Adv}_{\Pi}^{\text{EAV}}(A) = |\Pr(A \text{ in "world 0" accepts}) - \Pr(A \text{ in "world 1" accepts})|.$$

By adding 0, we have

$$\begin{aligned} \text{Adv}_{\Pi}^{\text{EAV}}(A) &= |\Pr(A \text{ in "world 0" accepts}) - \Pr(A \text{ in "world 1" accepts}) \\ &\quad + \Pr(A \text{ in random } c \text{ accepts}) - \Pr(A \text{ in random } c \text{ accepts})|. \end{aligned}$$

By using the hybrid argument, i.e., the triangle inequality,<sup>b</sup> we have

$$\begin{aligned} \text{Adv}_{\Pi}^{\text{EAV}}(A) &= |\Pr(A \text{ in "world 0" accepts}) - \Pr(A \text{ in "world 1" accepts}) \\ &\quad + \Pr(A \text{ in random } c \text{ accepts}) - \Pr(A \text{ in random } c \text{ accepts})| \\ &\leq |\Pr(A \text{ in "world 0" accepts}) - \Pr(A \text{ in random } c \text{ accepts})| \\ &\quad + |\Pr(A \text{ in "world 1" accepts}) - \Pr(A \text{ in random } c \text{ accepts})|. \end{aligned}$$

We can replace step two of  $D$  to use  $c = m_1 \oplus y$ , which we name it  $D'$ . And now we can substitute  $\text{Adv}_G^{\text{PRG}}(D)$  and  $\text{Adv}_G^{\text{PRG}}(D')$  into the above inequality and get

$$\text{Adv}_{\Pi}^{\text{EAV}}(A) \leq \text{Adv}_G^{\text{PRG}}(D) + \text{Adv}_G^{\text{PRG}}(D') = \text{negl}(n)$$

since we know that  $G$  is **PRG**, hence both  $\text{Adv}_G^{\text{PRG}}(D)$  and  $\text{Adv}_G^{\text{PRG}}(D')$  are **negligible**. ■

<sup>a</sup>We will deal with  $m_1$  later.

<sup>b</sup> $|a + b| \leq |a| + |b|$ , or  $|a - b| = |a - b + c - c| \leq |a - c| + |c - b|$ .

## 2.3 Chosen Plaintext Attack Security

Although the constructed **EAV secure scheme** sounds promising, it still suffers from the key-reuse problem: suppose we have  $c_1 = m_1 \oplus G(k)$  and  $c_2 = m_2 \oplus G(k)$ , then when an adversary gets  $c_1$  and  $c_2$ ,

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

then we fall into the same situation as depicted in **Figure 1.1**.

**Note (Potential solution).** We might use a **stream cipher** to generate new keys, but this is inconvenient for the decrypting user if the ciphertext are not ordered.

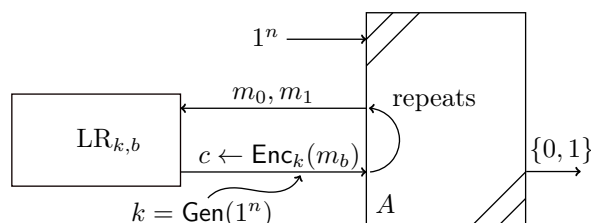
### 2.3.1 Chosen Plaintext Attack Game

To enhance the notion of [EAV security](#), we can address the key-reuse issue, i.e., consider a repeated [EAV game](#). In order to simplify things a bit, we abstract the “game mechanism” in [EAV games](#), i.e., the generation of  $c$ , into a call-able function, called [left-right oracle](#).

**Definition 2.3.1** (Left-right oracle). The *left-right oracle*  $\text{LR}_{k,b}(\cdot, \cdot)$  with parameters  $k, b$  on input  $(m_0, m_1)$  is defined as

$$\text{LR}_{k,b}(m_0, m_1) = \begin{cases} \text{Enc}_k(m_b), & \text{if } |m_0| = |m_1|; \\ \emptyset, & \text{if } |m_0| \neq |m_1|. \end{cases}$$

**Definition 2.3.2** (Chosen plaintext attack game). The *chosen plaintext attack game* is a model of a repeated [EAV game](#) for a probabilistic polynomial time adversary  $A$  against  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$ .  $A$  can only make  $\text{poly}(n)$  number of queries  $(m_0, m_1)$  to a [Left-right oracle](#)  $\text{LR}_{k,b}(\cdot, \cdot)$  to get a ciphertext for each query, and output a decision bit in the end.



**Intuition.** The [CPA game](#) is a model for [EAV security](#) that supports key-reuse. That way, if a key-reuse problem exists, the adversary will be able to use it to their [advantage](#).

**Remark.** The decision bit  $b$  and the encryption key  $k$  are only chosen *once* at the start. For example, if the key is 7 and  $b$  is 1, then the [LR oracle](#) will return  $\text{Enc}_7(m_1)$  for every query.

## Lecture 9: CPA Secure Scheme and the Construction

To decipher [Definition 2.3.2](#), a [CPA game](#) is conducted as follows.

6 Feb. 10:30

1. Adversary  $A$  takes the security parameter  $1^n$  and a key  $k$  will be generated according to the protocol.
2. Adversary  $A$  will then make adaptive queries by inputting two messages  $m_0, m_1$  into the [left-right oracle](#)  $\text{LR}_{k,b}$ . The output of the [left-right oracle](#) will be sent to the adversary  $A$ . These processes of the oracle will be repeated polynomial (in  $n$ ) many times.
3. Finally, adversary  $A$  will output the final decision.

**Example.** In WWII, American forces managed to decipher part of Japanese communication codes and found out their next target was a location called “AF.” American forces suspected that “AF” was Midway, so they sent a false message saying “Midway is running out of water.” Another piece of codes was deciphered afterwards, saying “AF is running out of water” and it was clear that “AF” was Midway. In this example, the adversary is American forces and the false message is the chosen plaintext.

### 2.3.2 Chosen Plaintext Attack Secrecy

Consider the new notion of [advantage](#) against the [CPA game](#).

**Definition 2.3.3 (Advantage).** Given an adversary  $A$  in a CPA game, the *advantage*  $\text{Adv}_{\Pi}^{\text{CPA}}(A)$  in distinguishing “world 0” and “world 1” is given by

$$\text{Adv}_{\Pi}^{\text{CPA}}(A) := \left| \Pr\left(A^{\text{LR}_{k,0}(\cdot, \cdot)} \text{ accepts} \right) - \Pr\left(A^{\text{LR}_{k,1}(\cdot, \cdot)} \text{ accepts} \right) \right|.$$

**Notation.**  $A^{\text{LR}_{k,b}(\cdot, \cdot)}$  means the adversary  $A$  can interact with the oracle.

Then, we have the following.

**Definition 2.3.4 (Chosen Plaintext attack secrecy).** An encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *chosen plaintext attack secure* if for every probabilistic polynomial time adversary  $A$ , the advantage is negligible.

**Remark.** Definition 2.3.2 is equivalent as saying that  $A$  have access to  $\text{Enc}$  and a single call to  $\text{LR}$ .

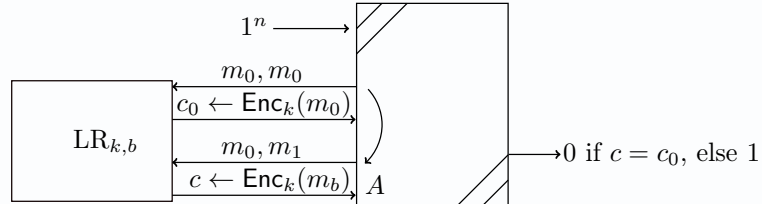
### 2.3.3 Chosen Plaintext Attack Secure Schemes

Now we ask that whether a CPA secure scheme exists. Unfortunately, we see the following.

**Claim.** A CPA secure encryption scheme does not exist.

**Proof.** Let  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  be CPA secure and  $A$  be an efficient attacker and consider the following.

- Firstly, take any  $m_0$  and query  $\text{LR}_{k,b}(m_0, m_0)$  to receive  $c_0 = \text{Enc}_k(m_0)$ .
- Then take  $\text{LR}_{k,b}(m_0, m_1)$  where  $m_1 \neq m_0$  but  $|m_0| = |m_1|$  and receive  $c = \text{LR}_{k,1}(m_0, m_1)$ .
- If  $c = c_0$ , outputs 0, else output 1.



If  $A$  is in the “left world” ( $b = 0$ ), then it will always output 0; if  $A$  is in the “right world” ( $b = 1$ ),  $A$  will always output 1 by correctness. Therefore,

$$\text{Adv}_{\Pi}^{\text{CPA}}(A) = \left| \Pr\left(A^{\text{LR}_{k,0}(\cdot, \cdot)} \text{ accepts} \right) - \Pr\left(A^{\text{LR}_{k,1}(\cdot, \cdot)} \text{ accepts} \right) \right| = 1 \neq \text{negl}(n).$$

⊗

**Problem.** There’s a bug in the above explanation!

**Answer.** The attacker  $A$  assumes that each time the ciphertext for  $m_0$  is the same, which is not the case in the reality since it is dangerous to do so (and there’s no reason to do so since  $\text{Enc}$  can be probabilistic).

⊗

**Intuition.** From the encryption side, the information of whether the same message is encrypted should be protected, so the ciphertext space should be much larger than the message space and cannot be solved efficiently by brute-force.

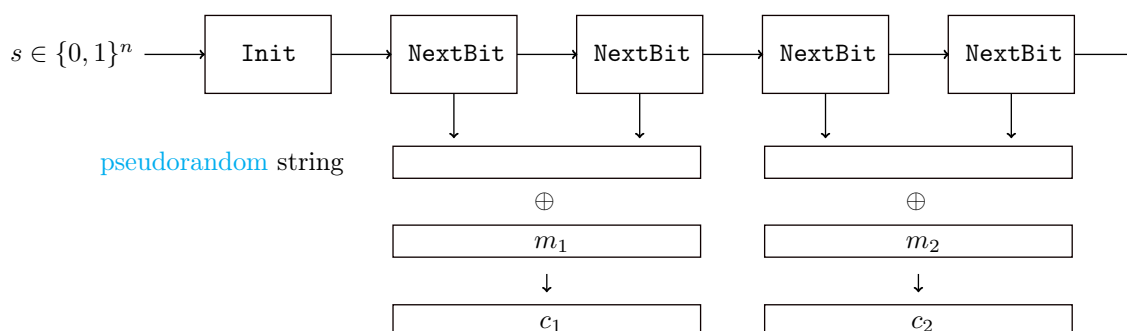
Hence, the correct way to put this is the following.

**Theorem 2.3.1.** There does not exist a CPA secure encryption scheme with deterministic and stateless<sup>a</sup>  $\text{Enc}_k(\cdot)$ .

<sup>a</sup>This means the ciphertext for the same message in different times should be the same.

This suggests that we still have hope! Theorem 2.3.1 suggests that we can try to construct a CPA secure scheme by using stream cipher.

- Firstly, initialize the stream cipher with the key.<sup>3</sup>
- Then encrypt the message by XOR-ing it with part of the string, and we get ciphertext  $c_1$ .
- Each time we encrypt the message, the string used in last time will be discarded and next bits of string will be used.
- Note that the sequence of the message is also sent in the ciphertexts.



**Remark.** This encryption scheme will survive in the chosen plaintext attack.

**Proof.** Because each time a same message  $m_0$  will be encrypted into different ciphertexts. The sequence does not affect the security because even if the attacker knew it, it cannot know where to start deciphering the text. \*

**Note.** However, if the number of messages is very large, the decryption will take a great deal of time since we have to look through the history and locate the piece of message we want.

## Lecture 10: Pseudorandom Functions

**As previously seen.** For a deterministic Enc in a CPA game, adversary could send in two identical messages ( $m_0, m_0$ ) to find the encryption of an individual message. Therefore, we need an encryption strategy that appears randoms and provides a different output for the same message if ran multiple times.

8 Feb. 10:30

Stream cipher seemed promising, but fell short in being tedious and inefficient in ensuring that the users who are supposed to know the message are in the same state. The following problem illustrates this issue.

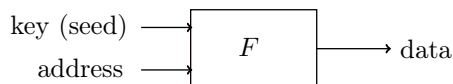
**Example.** If a user wants to know the 20-th bit they need to use the key to get the initial state and go through each of the previous bits.

So, our next goal is to find a methodology that allows stateless access.

### 2.4 Pseudorandom Functions

To address the aforementioned issue, we want something like this:

<sup>3</sup>Note that the string generated is pseudorandom.



This suggests the notion of **keyed function**.

**Definition 2.4.1 (Keyed function).** A *keyed function*  $F: \mathcal{K} \times X \rightarrow Y$  is a function with domain being the product between the key space and the input space (address).

We see that a **keyed function** is just any kind of function with domain in the form of  $\mathcal{K} \times X$  and with an appropriate interpretation.

**Example.**  $\mathcal{K} = X = Y = \{0, 1\}^n$ .

**Notation.** We usually write  $F(k, x) =: F_k(x)$  for a **keyed function**  $F$ .

Now, we simply want that given a key  $k \in \mathcal{K}$ , we can access an arbitrary address  $x \in X$  such that the output  $y$  looks random, i.e., we want a **random function**.

**Notation (Random function).** The *random function*  $\mathcal{U}: X \rightarrow Y$  is a deterministic (consistent) function such that the lookup table is a uniformly random.

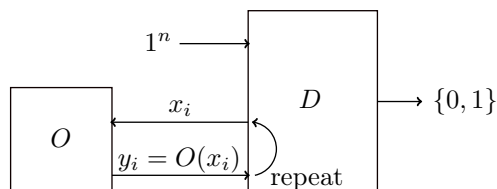
Input	Output
00	100
01	111
10	010
11	100

Table 2.1: A **random function** with  $X = \{0, 1\}^2, Y = \{0, 1\}^3$ .

### 2.4.1 Pseudorandom Functions Game

And naturally, we want a *pseudo* version of a **random functions**, which we called **pseudorandom functions**. Before we formally introduce the definition, as always, we should define the **advantage** under some adversary-style games set-up. To do this, we design a **PRF game** similar to the game of **PRG**.

**Definition 2.4.2 (Pseudorandom function game).** The *pseudorandom function game* contains a **distinguisher**  $D$  and an oracle (black box)  $O$  which is either  $O(\cdot) = F_k(\cdot)$  for secret random key  $k \leftarrow \{0, 1\}^n$  (the “real world”) or  $O(\cdot) = \mathcal{U}(\cdot)$  (the “ideal world”), where  $D$  can query  $O$  with various inputs and get the corresponding outputs, then output a decision bit.



Then, we naturally define the **advantage** in this context as follows.

**Definition 2.4.3 (Advantage).** Given a distinguisher  $D$  in a **PRF game**, the *advantage*  $\text{Adv}_F(D)$  in distinguishing the “real world” ( $O = F_k$ ) and the “ideal world” ( $O = \mathcal{U}$ ) is given by

$$\text{Adv}_F(D) := \left| \Pr_{k \leftarrow \mathcal{K}} (D^{F_k(\cdot)}(1^n) = 1) - \Pr_{\mathcal{U} \leftarrow \text{RF}} (D^{\mathcal{U}(\cdot)}(1^n) = 1) \right|.$$

**Intuition.** The first probability represents the “real world”  $O = F_k$  for a uniform  $k \in \mathcal{K}$ , and the second probability represents the “ideal world”  $O = \mathcal{U}$ .

## 2.4.2 Pseudorandom Functions

Finally, we have the following definition.

**Definition 2.4.4 (Pseudorandom function).** A **keyed function**  $F$  is a *pseudorandom function* if every probabilistic polynomial time distinguisher  $D$  has **negligible advantage** in the **PRF game** against  $F$ .

**Intuition.** You can think of **stream ciphers** as cassette tapes, and **PRFs** as CDs.

**Remark.** **PRFs** exist if and only if **PRGs** exist. You can think of the key as similar to the seed in the **PRG**.

**Example (Poor example).** Let  $F_k(x) = k \oplus x$ , given  $O$ .

**Proof.** Query  $m_0 = 0 \dots 0, m_1 = 1 \dots 1$  to get  $y_0 = O(0 \dots 0)$  and  $y_1 = O(1 \dots 1)$ . If  $y_0 \oplus y_1 = 1 \dots 1$  accept, else reject. We see that in the real world, we have

$$(k \oplus x_0) \oplus (k \oplus x_1) = x_0 \oplus x_1 = 1 \dots 1,$$

hence  $\Pr(\text{accept}) = 1$ . While in the ideal world,  $\Pr(\text{accept}) = 2^{-n}$ , hence

$$\text{Adv}_F(D) = |\Pr(\text{accept in real world}) - \Pr(\text{accept in ideal world})| = 1 - 2^{-n} \approx 1.$$

⊛

## 2.4.3 PRF-Based CPA-Secure Schemes

Now we’re interested in utilizing a **PRF**  $F_k(\cdot): \mathcal{K} \times X \rightarrow Y$  to build a **CPA secure scheme**, where we let  $\mathcal{K} = X = Y = \{0, 1\}^n$ . Consider the following **scheme**  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  by

- $\text{Gen}(1^n)$ : outputs a uniformly random  $k \leftarrow \{0, 1\}^n$ ;
- $\text{Enc}_k(m)$  given  $m \in \{0, 1\}^n$ : choose a random  $x \leftarrow \{0, 1\}^n$ , outputs  $(x, m \oplus F_k(x))$ ;
- $\text{Dec}_k(c)$  given  $c = (x, c')$ : outputs  $c' \oplus F_k(x)$ .

In real world applications, it is important to construct it with a key such that the key space  $\mathcal{K}$  is fairly large since if  $\mathcal{K}$  is small, **PRF** is broken because with brute force you can guess a key or try all the keys.

**Example.** DES (data encryption standard, 1976-1977) fell fault to this for  $\mathcal{K} = \{0, 1\}^{56}$ ,  $X = Y = \{0, 1\}^{64}$ .

**Example.** Currently AES (2001-2002) is using  $X = Y = \{0, 1\}^{128}$ , while  $\mathcal{K} = \{0, 1\}^n$  with  $n = 128, 192, 256$ . Even for  $n = 128$  is considered **secure**.

# Lecture 11: PRF-Based and Arbitrary Length Encryption

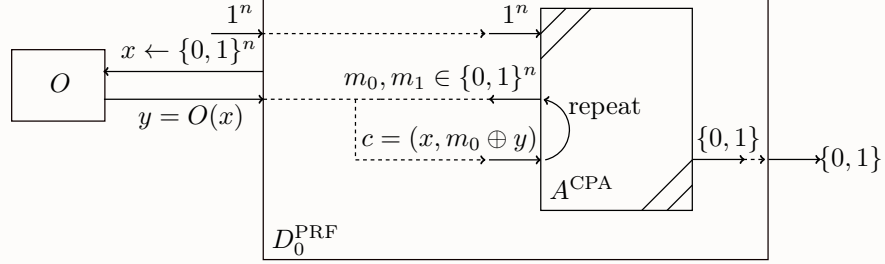
Now, we show that  $\Pi$  is actually **CPA-secure**.

13 Feb. 10:30



**Theorem 2.4.1.** If  $F$  is a (secure) PRF, then the above scheme  $\Pi$  is CPA-secure for messages of length  $n$ .

**Proof.** We will conduct a proof by reduction. Let  $A$  be any CPA adversary against the system. If  $A$  has non-negligible advantage over the system, then we can write a distinguisher  $D$  for the PRF backing the system. In diagram, we have the following.



In this diagram, we are utilizing the adversary  $A$  within distinguisher  $D_0$  to help output a decision bit.  $A$  outputs some messages  $m_0$  and  $m_1 \leftarrow \{0, 1\}^n$ , out of which we arbitrarily choose  $m_0$ .  $D_0$  can then be defined as such:

- choose random  $x$ ;
- query the oracle  $O$  to obtain  $y$ , which is either  $F_k(x)$  or a uniform random string;
- encode the previously chosen  $m_0$  and pass  $c = (x, m_0 \oplus y)$  to the adversary  $A$ ;
- output  $A$ 's decision bit.

There are two “worlds” for  $D_0$ :

- real world:  $O = F_k(x)$  for random  $k$ .  $D_0$  perfectly simulates the “left world” in the CPA game for adversary  $A$  because it evaluates  $c = (x, m_0 \oplus y) = (x, m_0 \oplus F_k(x))$ . This is exactly the input  $c$  that  $A$  is expecting as the encryption of  $m_0$ ;
- ideal world:  $O = \mathcal{U}$  (a random function).  $D_0$  perfectly simulates the following “hybrid,” “random ciphertext” world. In this world, our adversary receives nonsense because  $(x, m_0 \oplus y)$  is complete randomness, and  $A$  does not expect this!  $A$  is expecting either an encryption of  $m_0$  or  $m_1$ , which are the true “left” and “right worlds” for  $A$ .

**Problem.** We note that  $O$  is **consistent**, i.e., for a fixed  $a$ ,  $O(a)$  will always be the same. Specifically, when  $O = \mathcal{U}$ , we can't treat it as truly random, since independence of  $c'$  holds if and only if all queries  $x$  to the oracle  $O$  are distinct.

**Answer.** This is not a concern because of the birthday paradox.

**Note (Birthday paradox).** If we pick  $q$  random objects from a population of a size  $N$ , then  $\Pr(\text{two or more are the same}) \approx q^2/N$ , where in our scenario  $q$  is the number of queries to the oracle and  $N$  is the size of the sample space  $X = \{0, 1\}^n$ .

From here we can show that the probability of the “ideal world” accepting for  $D_0$  in the PRF game is negligibly different from the probability of the “hybrid world” accepting in the CPA game, specifically,

$$\begin{aligned} \Pr(D_0^{\mathcal{U}(\cdot)} = 1) &= \Pr(A = 1 \text{ in “hybrid”}) \pm \text{negl}(n) \\ &\leq \frac{\# \text{ queries}^2}{2^{n+1}} \pm \text{negl}(n) = \frac{\text{poly}(n)}{2^{n+1}} \pm \text{negl}(n) = \text{negl}(n). \end{aligned} \quad (2.1)$$

⊛

Now we can write the **advantage** of  $A$  in the **CPA game** against  $\Pi$  as

$$\begin{aligned} \text{Adv}_{\Pi}^{\text{CPA}}(A) &= \left| \Pr(A = 1 \text{ in "left"}) - \Pr(A = 1 \text{ in "right"}) \right. \\ &\quad \left. + \Pr(A = 1 \text{ in "hybrid"}) - \Pr(A = 1 \text{ in "hybrid"}) \right| \\ &\leq |\Pr(A = 1 \text{ in "left"}) - \Pr(A = 1 \text{ in "hybrid"})| \\ &\quad + |\Pr(A = 1 \text{ in "right"}) - \Pr(A = 1 \text{ in "hybrid"})| \\ &\leq (\text{Adv}_F^{\text{PRF}}(D_0) + \text{negl}(n)) + (\text{Adv}_F^{\text{PRF}}(D_1) + \text{negl}(n)), \end{aligned}$$

where the last line follows from [Equation 2.1](#).<sup>a</sup> Finally, notice that the **advantage** on both  $D_0$  and  $D_1$  is **negligible** because we are assuming that  $F$  is a **PRF** for which all polynomial time **distinguishers** have a **negligible advantage** in the **PRF game** against  $F$ .

Hence, in all, the overall **advantage** of  $\text{Adv}_{\Pi}^{\text{CPA}}(A)$  is

$$\text{Adv}_{\Pi}^{\text{CPA}}(A) \leq \text{negl}(n) + \text{negl}(n) + \text{negl}(n) + \text{negl}(n) = \text{negl}(n),$$

which is **negligible** for all possible  $A$ , hence  $\Pi$  is **CPA-secure**. ■

<sup>a</sup>Note that we implicitly use the **advantage** where we build  $D_1$  as  $D_0$  by passing the adversary  $m_1$  instead of  $m_0$ .

[Theorem 2.4.1](#) shows that given a valid **PRF**, we can build a valid **CPA-secure scheme** that prevents the original drawbacks such as key reuse.

## 2.4.4 Remaining Problems

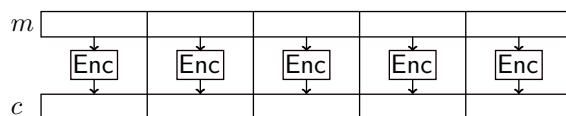
However, new issues arise:

- (a) Messages must be of fixed-length by the definition of  $\Pi$  at hand (length  $n$  and **only** length  $n$ ).
- (b) **CPA security** does not address **active adversaries**, i.e., every adversary so far has been a passive observer of behaviors of these **encryption schemes**, whereas an active adversary would be able to affect what is decrypted as well. This type of attack is called *chosen ciphertext attack*.

**Example.** Enigma machine was also only for passive adversaries.

**Example.** Suppose an adversary wants to mess with your salary. They may not know what it is, but they want to mess with the ciphertext of your salary to make it higher or lower by some factor, so that the decryption is incorrect.

Let's first focus on the long message issue. To address it, a naive approach will be reusing the key! We could chop up the message into parts that our **CPA secure scheme** can take.



This is **CPA secure**, but we have an efficiency issue because our ciphertext  $c = (x, c')$  where  $x \in \{0, 1\}^n$  and  $c' \in \{0, 1\}^n$  is with size  $2 \times n$ . This is too large, and we need to strive for smaller overhead.

## 2.5 Modes of Operations and Encryption in Practice

To address the problems mentioned, we discuss modes of operation for encrypting arbitrary-length messages using a **stream cipher**.

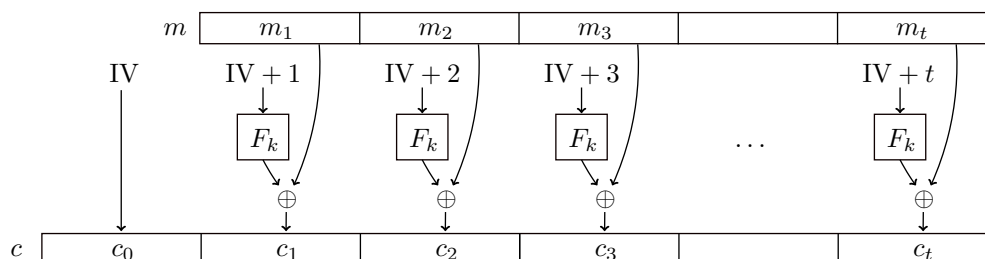
### 2.5.1 Counter Mode

Let  $F$  be a **PRF**. To encrypt  $m \in \{0, 1\}^*$ , we first break up  $m$  into blocks of length  $n$  each as

$$m = m_1 \| m_2 \| \dots \| m_t$$

for  $|m_i| = n$  and  $|m_t| \leq n$ . Then, we proceed as follows.

- Choose a random address as an initialization vector  $IV \leftarrow \{0, 1\}^n$ .
- Initiate the ciphertext  $c$  with  $c_0 \leftarrow IV$ .
- For  $m_i$ , encode with  $F_k(IV + i)$  such that  $c_i \leftarrow m_i \oplus F_k(IV + i)$ .
- Concatenate them together to get  $c \leftarrow c_0 \| c_1 \| \dots \| c_t$ .



**Notation** (Counter mode). We call such a [scheme](#) *counter mode*, or *CTR*.

## Lecture 12: Modes of Operations

By using [CTR](#), we can now encrypt messages of various lengths. Specifically, [counter mode](#) gets around the issue by having a single initialization vector (IV) that gets incremented with each block. Subsequent blocks of the message will be padded with the result of the [PRF](#) given the input of  $IV + i$ , where  $i$  is the block number.

15 Feb. 10:30

**Proposition 2.5.1.** If  $F$  is [PRF](#), then [CTR](#) is [CPA secure](#).

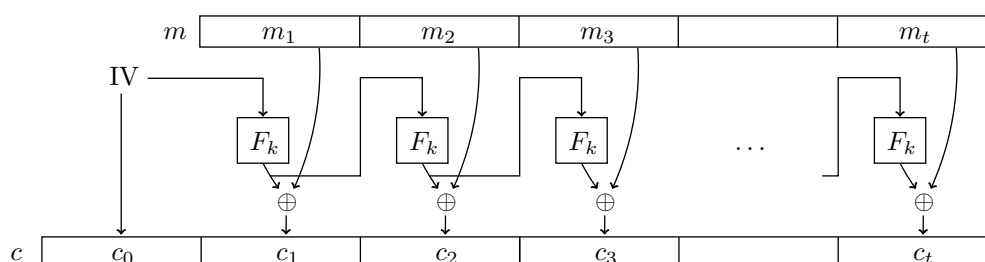
**Proof.** Every Input to  $F$  across the entire [CPA game](#) is distinct, with a very [negligible](#) probability to be the same. Therefore, all output of  $F$  will “look like” truly random and independent. ■

**Remark.** Advantages of using [CTR](#):

- simple and satisfies [CPA secure](#);
- does not require a message length that is a multiple of  $n$ ;
- fast and efficient because it can be computed in parallel;
- no need for padding (we can just trim the output of  $F$  to fit the last message block size  $|m_t|$ ).

### 2.5.2 Output Feedback Mode

One potential issue with [counter mode](#) is that the input for each [PRF](#) is easy for an adversary to calculate for each block if they are able to guess the initialization vector. This is a concern because if the function that generates the IV does not have a good random distribution, the system will not be secure. Consider the following [scheme](#).



**Notation** (Output feedback mode). We call such a [scheme](#) *output feedback mode*, or *OFB*.

**OFB** mode gets around this issue by feeding the output from the previous block as an input to each block, starting with the IV for the first block. This makes the index of each block practically random which is much closer to the original [CPA game](#) model.

**Proposition 2.5.2.** If  $F$  is [PRF](#), then **OFB** is [CPA secure](#).

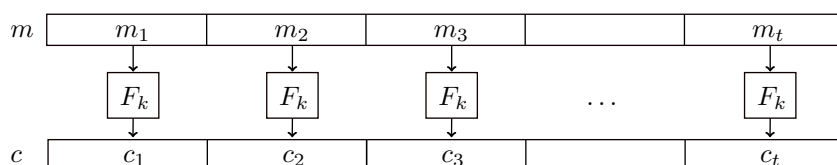
**Proof.** Since **OFB** does not have repeated input to  $F$  (happen with [negligible](#) probability). ■

**Remark.** Some remarks of using **OFB**:

- **OFB** is random by the chain, which is independent of message, not by counter;
- **OFB** can increase security: because IV is not random, if IV is attacked, every block in **CTR** can in danger. In contrast, in **OFB**, because IV is executed with  $F$  before XOR with message block, **OFB** can still be secure even if IV is guessed.
- **OFB cannot** be computed in parallel.

### 2.5.3 Electronic Codebook Mode

Consider the following [scheme](#).



**Notation** (Electronic Codebook mode). We call such a [scheme](#) *electronic codebook mode*, or *ECB*.

We see that this is not similar to **CTR** or **OFB**. Since for **ECB**, we use the message as input to generate ciphertext.

**Proposition 2.5.3.** Even if  $F$  is [PRF](#), **ECB** is **not** [CPA secure](#).

**Proof.** Since same message block is encrypted to the same ciphertext block, implying that **ECB** is not [CPA secure](#) because it is stateless and deterministic from [Theorem 2.3.1](#). ■

Clearly, we have a clear problem when trying to decrypt.

**Remark (Decryption).** We see the following.

- If  $F$  is [PRF](#), we are not able to decrypt the ciphertext because it is not guaranteed that the inverse function of  $F$  exists.
- If  $F$  is [PRP](#) (will be introduced soon!), we are able to decrypt the ciphertext, but it is still not [CPA secure](#) because the same message still shares the same ciphertext.

### 2.5.4 Pseudorandom Permutation

To introduce the next mode, we need some preliminaries.

**Definition 2.5.1 (Block cipher).** A bijective [keyed function](#)  $F: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  such that both  $F_k$  and  $F_k^{-1}$  can be computed efficiently given the key  $k$  is called a *block cipher*.

**Intuition.** Block cipher is an invertible version of a PRF.

We see that we need invertibility, which proposes the following notion.

**Definition 2.5.2** (Pseudorandom permutation). A keyed function  $F$  is a *pseudorandom permutation* if every probabilistic polynomial time adversary  $A$  has negligible advantage in distinguishing between  $F_k$  and a random bijection with a random key  $k \leftarrow \mathcal{K}$ .

Where here, the advantage is defined as follows.<sup>4</sup>

**Note.** Notice that in  $A$  can call the oracle multiple times.

**Definition 2.5.3** (Advantage). Given a adversary  $A$  which tries to distinguish a PRP from a random bijection, the *advantage*  $\text{Adv}_F(A)$  in distinguishing the “real world” ( $F_k$ ) and the “ideal world” ( $P \leftarrow P_n$ )<sup>a</sup> is given by

$$\text{Adv}_F(A) := \left| \Pr_{k \leftarrow \mathcal{K}}(A^{F_k(\cdot)} = 1) - \Pr_{P \leftarrow P_n}(A^{P(\cdot)} = 1) \right|.$$

<sup>a</sup>We use  $P_n$  to denote the set of all bijection on  $\{0, 1\}^n$ .

**Note.** If we can give  $A$  access to  $F_k^{-1}(\cdot)/P^{-1}(\cdot)$  as well, then  $F_k$  is a strong PRP.

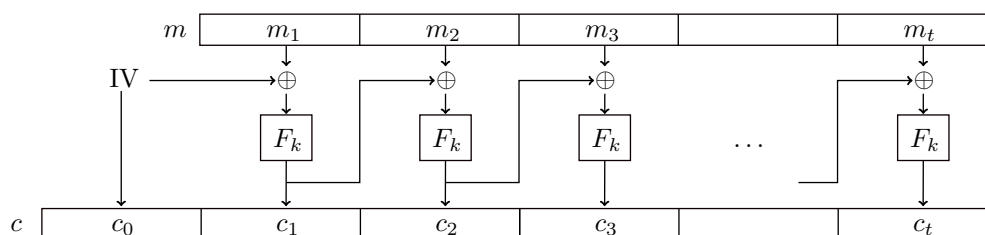
**Theorem 2.5.1.** If  $F$  is a PRP,  $F$  is also a PRF.

**Proof idea.** Given oracle access, a random permutation is identical to a random function as long as distinct input queries to the random function don't return the same value (because if  $c_1 = c_2$ , function  $F$  is not invertible), which implies "birthday collision" on outputs. However, collision happens with negligible probability:  $\frac{\text{poly}(n)}{2^n}$ . Thus, under the efficient setting, if  $F$  is a PRP, it is also a PRF. ■

With all these, we now introduce the final mode of operation combines ideas from OFB and ECB.

## 2.5.5 Cipher Block Chaining

Like OFB, we now use the result of the previous block to help encrypt each block, starting with IV for the first block. However, instead of feeding a pseudorandom string into a PRF, the pad of the message and the original block is used as an input. To decrypt, the operations are reversed for each block using the inverse of  $F$  when necessary. In diagram, we see that for encryption, we have



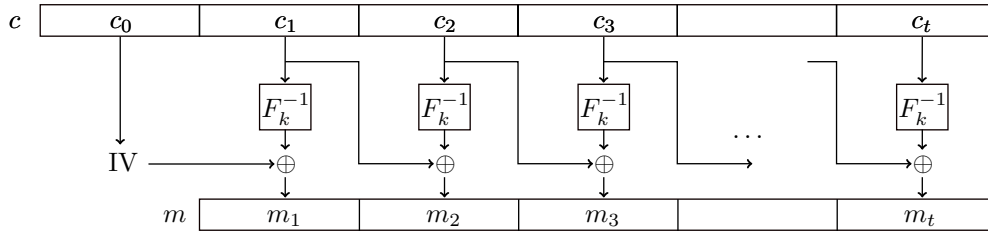
In other words, we have

$$\begin{cases} c_0 = \text{IV} \\ c_i = F_k(m_i \oplus c_{i-1}). \end{cases}$$

**Note.** We need to find a way to increase the length of  $m_t$  to  $n$ -bits.

And to decrypt, we have

<sup>4</sup>Formally, we should define the “PRP game,” which is just a variation of PRF game!



In other words, we have

$$m_i = c_{i-1} \oplus F_k^{-1}(c_i).$$

**Note.** We need to un-pad “ $m_t$ ” to the right size.

**Notation** (Cipher block chaining). We call such a [scheme](#) *cipher block chaining*, or *CBC*.

**Theorem 2.5.2.** If  $F$  is a [PRP](#) (which is also a [PRF](#)), then [CBC](#) is [CPA secure](#).

**Proof idea.** All ciphertexts look like random independent strings as long as no input to  $F_k(\cdot)$  is ever repeated. Based on the [birthday paradox](#), repetitions happen with only [negligible](#) ( $\text{poly}(n)/2^n$ ) probability by the choice of IV and [pseudorandom](#) outputs of prior blocks. ■

**Remark.** Some remarks of using [CBC](#):

- encryption is sequential, i.e., we cannot compute ciphertext without computing all prior blocks;<sup>[a](#)</sup>
- provides more security than [CTR](#) if IV is predictable;
- requires padding the last message block since we cannot trim the output of  $F$  to fit the last message block.<sup>[b](#)</sup>

<sup>a</sup>Although decryption can be done in parallel.

<sup>b</sup>Which will also increase the execution time.

The last point means that the plaintext must be extended or padded to the correct length, i.e., multiple of  $n$ . This leads to a potential vulnerability we next discuss.

# Chapter 3

## Message Authentication

### Lecture 13: Integrity and Authentication

The fact that CBC requires padding the last block of the message leads to a potential vulnerability called “authenticity”: if attacker can change the ciphertexts and send it to the receiver in the CBC mode, the attacker can decrypt the entire message. More broadly, with the previous methodology, we can’t know whether the message is sent by a particular user, which is a huge problem. But firstly, let’s see what’s wrong if the authenticity can’t be ensured in the CBC mode.

20 Feb. 10:30

#### 3.1 Security vs. Authenticity

##### 3.1.1 Cipher Block Chaining Padding

This padding must be done in an unambiguous way, and one possible standard to accomplish this is the PKCS standard, and we are interested in the following specific version.

**Definition 3.1.1 (PKCS #5).** Let  $L$  be the length of a message block in bytes. On message  $m$ , let  $b \in \{1, \dots, L\}$ <sup>a</sup> be the number of bytes that when added to the message  $m$ , makes the total message length equal to a multiple of  $L$  as required in CBC. The PKCS #5 standard requires that the new message  $\hat{m}$  after the padding to be

$$\hat{m} := m \parallel \underbrace{bb \dots b}_{b \text{ times}}$$

i.e., we add exactly  $b$   $b$ ’s at the end of  $m$ .

<sup>a</sup> $b$  can’t be 0 implies that if the length of  $m$  is equal to a multiple of  $L$  already, we must add another  $L$  bytes.

**Note.** We’re now thinking about characters, not bits (otherwise the above doesn’t make sense).

**Example.** If  $b = 5$ , we have  $\hat{m} = m \parallel 55555$ .

However, this can create some vulnerabilities in CBC mode. This is based on the following fact.

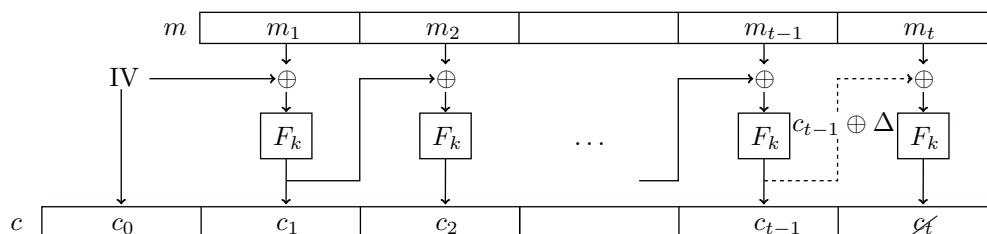
**Remark (Practical implementation).** If the last message block is formatted incorrectly w.r.t. PKCS #5, it’s common that a public error will be announced.

**Example.** Cat55555 is a well formatted message block, which is easy to decrypt.

**Example.** Cat54555 is a poorly formatted message block is, which will error.

### 3.1.2 Padded Oracle Attacks

All these may seem irrelevant: what is this all about? Don't we already prove that CBC is CPA secure in Theorem 2.5.2? In reality is that, whenever we do something outside the scheme (in this case, using PKCS #5 for padding and allowing other things as we will see) that is not considered in the standard CPA game, there's no guarantee anymore. To be more specific, there's a potential "integrity" issue: what if the adversary is able to change  $c_{t-1}$  and throw it back to the sender?



Formally, if the adversary is able to do this, then the output of the last block, i.e.,  $c_t$ , becomes

$$c'_t = F_k(m_t \oplus (c_{t-1} \oplus \Delta)) = F_k((m_t \oplus \Delta) \oplus c_{t-1}).$$

**Problem.** What does this suggest?

**Answer.** By letting  $c'_{t-1} = c_{t-1} \oplus \Delta$ , we're able to deceive the user (or the "format checker") that " $m_t$  is  $m_t \oplus \Delta$ "!  $\otimes$

We see that the same thing can be applied in any block, hence, we are able to "shift" the message any way we want.

**Intuition.** An active adversary can recover the plaintext by shifting each bit in the ciphertext and observing whether an error gets thrown.

Now, we claim the following.

**Claim.** Just knowing that an error occurs is enough for the adversary to recover the entire plaintext!

**Proof.** Firstly, we claim that we can recover  $b$ . It's simple: we start by checking whether  $b = L$ . If it is, then the last block of the message should be  $m_t = L \ L \ \dots \ L$ . By "shifting" the left most character<sup>a</sup> by  $\delta$ , i.e., we make the "format checker" to think that  $m_t$  looks like

$$m'_t = (L + \delta) \ L \ \dots \ L.$$

Now the formatter thinks there's an error since by looking at the last  $L - 1$   $L$ 's, it thinks that the final (left most) character should also be an  $L$ , not  $L + \delta$ ! So we got an error prompt. This only happens when  $b = L$ , hence we can determine whether  $b = L$ . In the same way, we can keep ruling out the possibility by doing the same thing from left to right, until we found out  $b$ .

Once the adversary recovers  $b$ , the message is basically exposed in the same way: Suppose

$$m_t = \dots \ x \ \underbrace{b \ b \ \dots \ b}_{b \text{ times}},$$

where  $x$  is the last character of the data. Observe that if we replace  $b$  by  $b + 1$ , the formatter now will complain if  $x$  is not  $b + 1$ ! This implies that we can use the formatter to determine  $x$  is  $b + 1$  or not. Since we also shift  $x$  by  $\delta'$ , so we can actually determine whether  $x + \delta'$  is  $b + 1$  or not:

$$m'_t = \dots \ (x + \delta') \ \underbrace{(b + 1) \ (b + 1) \ \dots \ (b + 1)}_{b \text{ times}},$$

and if  $x + \delta' \neq b + 1$ , then we get an error prompt. We can simply try out different  $\delta'$  until there's no error. In that case,  $x = b + 1 - \delta'$ ! By recovering the last  $b + 1$  characters, we can replace them



by  $b + 2$ , and keep doing the same thing until every character in this block is decrypted. ⊛

<sup>a</sup>Notice that I didn't use the word "bit" due to the string representation.

**Notation** (Padded oracle attack). The above attack is called the *padded oracle attack*.

We learn that we need to be very careful with padding and consider attacks outside the [CPA model](#). In this case, the integrity of the cyphertexts used in the next block.

## 3.2 Message Authentication Codes

The [padded oracle attack](#) raises the problem of authenticity of the message, i.e., does this message really comes from a particular user? Obviously, this is very important from the above example. But there are many others obvious reasons suggesting that this is important.

**Example.** Alice is trying to send the message "Please give Eve \$20" to Bob. If Eve is able to read it and change the message to "Please give Eve \$200", then that would clearly be advantageous to Eve.

So how do we ensure that an active adversary can't modify the data being transmitted? Following our previous approach, we should define something similar to the [encryption scheme](#), i.e., we formalize how to generate string that we're going to sent. Now, instead of considering ciphertext, what we want is authenticity, so naturally, we define something called "tag", which can be thought of to be used to verify the identity of a user. Then, we define the following.

**Definition 3.2.1** (Message authentication code). The *message authentication code* or *MAC* is a tuple  $\Pi = (\text{Gen}, \text{Tag}, \text{Ver})$  defined as follows.

- $\text{Gen}(1^n)$  outputs a key  $k \in \mathcal{K}$  given the security parameter  $n$ .
- $\text{Tag}_k(m)$  outputs some short string called a "tag" given key  $k$  and message  $m$ .
- $\text{Ver}_k(m', t')$  accepts or rejects depending on whether  $m'$  and  $t'$  is a valid message-tag pair.

Same as before, we want "correctness" and "security".

**Definition 3.2.2** (Correctness). A [MAC](#)  $\Pi = (\text{Gen}, \text{Tag}, \text{Ver})$  satisfies *correctness* if for all  $m \in \mathcal{M}$  and  $k \in \mathcal{K}$ ,

$$\text{Ver}_k(m, \text{Tag}_k(m)) = 1.$$

As for "security", we want that the adversary can't produce a valid tag for a different  $m' \neq m$ .

**Intuition.** Seeing  $(m, t)$ , the adversary can't produce  $(m', t')$  with  $m' \neq m$  and  $\text{Ver}_k(m', t') = 1$ .

But since the adversary can wait for a long time, i.e., observing lots of valid  $(m, t)$  pairs, hence we should not just consider a one-time game.

## 3.3 Chosen Message Attack Security

### 3.3.1 Chosen Message Attack

Consider the following game.

**Definition 3.3.1** (Chosen message attack). The *chosen message attack* or *CMA* game given an adversary (called a *forgery*)  $F^a$  and a [MAC](#)  $\Pi = (\text{Gen}, \text{Tag}, \text{Ver})$  is conducted as follows.

1.  $k \leftarrow \text{Gen}(1^n)$  is kept secret from  $F$ .
2.  $F$  receives polynomially many tags for messages of their choices by querying the  $\text{Tag}_k(\cdot)$  oracle.

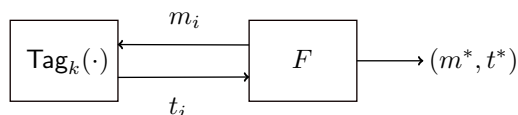
3.  $F$  outputs an attempted forgery  $(m^*, t^*)$ .

<sup>a</sup> $F$  gets access to the security parameter  $1^n$ .

**Definition 3.3.2 (Weak forgery).** In the **CMA**,  $F$  *weakly forges* if  $\text{Ver}_k(m^*, t^*) = 1$  and  $m^*$  is not a query from  $A$  to the tag oracle.

**Notation.** It's canonical to drop the word “weak” in **Definition 3.3.2**.

If you like a picture:



**Remark.** Notice that we didn't consider the case that  $F$  can resend a seen  $(m, t)$  pair. That is, **CMA game** doesn't consider the repeated case.

## Lecture 14: Message Authentication via PRFs

### 3.3.2 Chosen Message Attack Secrecy

22 Feb. 10:30

Then, we have the following security notion.

**Definition 3.3.3 (Advantage).** Given a forger  $F$  in an **CMA**, the *advantage*  $\text{Adv}_{\Pi}^{\text{CMA}}(F)$  in generating a fake message/tag pair is given by

$$\text{Adv}_{\Pi}^{\text{CMA}}(F) := \Pr_{k \leftarrow \text{Gen}} (F^{\text{Tag}_k(\cdot)} \text{ forges}).$$

**Definition 3.3.4 (Unforgeable).** A **MAC** is *unforgeable* under **CMA** if for every probabilistic polynomial time forger  $F$ , the **advantage** is **negligible**.

This has a more common name.

**Notation (UFCMA).** An **unforgeable MAC** under **CMA** is abbreviated as **UFCMA**.

**Remark (Canonical MAC).** If  $\text{Tag}_k(m)$  and  $\text{Ver}_k(m, t)$  are deterministic, then this **MAC** is called *canonical*. In this case,  $\text{Ver}_k(m, t)$  is automatic defined, i.e., it accepts if and only if  $t = \text{Tag}_k(m)$ .

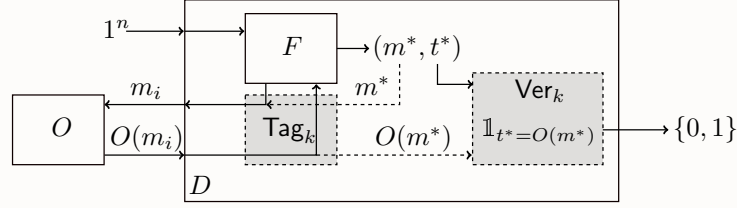
### 3.3.3 Chosen Message Attack Message Authentication Codes

We can use a **PRF** to construct a **MAC** that is **UFCMA**. For a fixed-length message  $m \in \{0, 1\}^\ell$ , suppose  $f$  is a **PRF** such that  $f: \mathcal{K} \times \{0, 1\}^\ell \rightarrow \{0, 1\}^n$ . We define a **MAC** as

- $\text{Gen}(1^n)$ : Choose a random  $k \leftarrow \mathcal{K}$ .
- $\text{Tag}_k(m)$ : Output  $\text{Tag}_k(m) = f_k(m)$ .
- $\text{Ver}_k(m, t)$ : Accept if and only if  $t = \text{Tag}_k(m)$ .

**Theorem 3.3.1.** This **MAC** is **UFCMA** if  $f$  is a **PRF**.

**Proof.** We prove the theorem by reduction. Let  $F$  be a probabilistic polynomial time forger, and we construct a **distinguisher**  $D$  against the **PRF**  $f$  as follows.



In words:

- $D$  runs a query  $m_i$  by  $F$  to  $O(\cdot)$  and relays the answer  $t_i = O(m_i)$  back to  $F$ .
- When  $F$  outputs  $(m^*, t^*)$ , query  $O$  again to get  $O(m^*)$ .
- Output 1 if and only if  $O(m^*) = t^*$  and  $m^* \notin \{m_i\}_i$ .

For  $F$ , the “real world” is when  $O = f_k$  and the “ideal world” is when  $O = \mathcal{U}$  where  $\mathcal{U}$  is a **random function**. The **advantage** of  $D$  is defined as

$$\text{Adv}^{\text{PRF}}(D) = |\Pr(D \text{ in “real world” accepts}) - \Pr(D \text{ in “ideal world” accepts})|.$$

Observe the following two cases:

- (a) If  $O = f_k$ : We are in the “real world” and  $D$  is a perfect simulation of the **MAC**. Thus,

$$\Pr(D \text{ in “real world” accepts}) = \Pr(F \text{ forges}) = \text{Adv}^{\text{CMA}}(F).$$

- (b) If  $O = \mathcal{U}$ : We are in the “ideal world” so we keep giving random values  $t$  to  $F$ . Since  $m^*$  is new,  $O(m^*)$  will be uniformly random and independent. Thus,

$$\Pr(D \text{ in “ideal world” accepts}) = 2^{-n}.$$

This suggests that  $\text{Adv}^{\text{PRF}}(D) = |\text{Adv}^{\text{CMA}}(F) - \text{negl}(n)|$ , which is also **negligible** since  $f$  is a **PRF** and  $D$  runs in polynomial time, we have

$$\text{Adv}^{\text{CMA}}(F) = \text{Adv}^{\text{PRF}}(D) \pm \text{negl}(n) = \text{negl}(n).$$

■

**Remark.** This **MAC** has a deterministic  $\text{Tag}_k(m)$  after specifying  $k$ .

### 3.3.4 Strong Unforgeability

You may wonder why we use the word “weak” in **Definition 3.3.2** but never actually mention it again. This is because while it’s satisfactory for many applications, **CMA** doesn’t necessarily rule out the ability to find a new tag  $t^*$ , for an old (queried) message  $m$ , but in **Definition 3.3.2** we rule this possibility out. In response, we define the following.

**Definition 3.3.5 (Strong forgery).** In the **CMA**,  $F$  *strongly forges* if  $\text{Ver}_k(m^*, t^*) = 1$  and  $(m^*, t^*)$  is different from all  $(m, t)$ ’s.

**Note.** For deterministic  $\text{Tag}_k(m)$ , **strong forgery** is equivalent to **weak forgery**. Also, for deterministic **MACs**, **strong forgery** is the same if we give the forger access to  $\text{Ver}_k(\cdot)$  in addition.

### 3.3.5 Domain Expansions

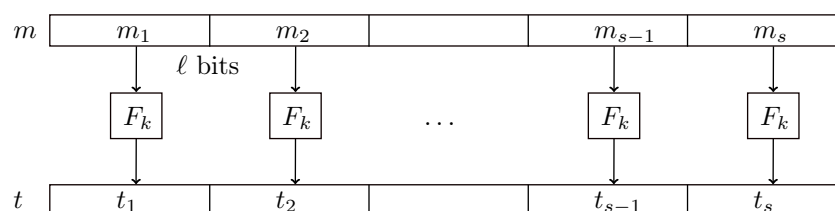
A natural question to ask next is whether there exists a **secure MAC** for messages of arbitrary length. Consider the previously constructed **secure MAC** for messages of length  $n$ , and let  $m$  be an arbitrary message of arbitrary length, we need to produce a method of tagging  $m$  without leaking any information to an adversary.

**Intuition.** An attractive naive implementation would be to simply parse the message into blocks of fixed-length and tag each block directly.

However, this implementation is not **secure**, and the final **MAC** for arbitrary length messages is not trivial.

#### Attempt 1

Firstly, we formalize the above idea. Suppose  $s$  is the number of blocks and  $m = m_1 \| \dots \| m_s$  where  $|m_i| = \ell$  and  $m \in \{0, 1\}^*$ . To tag  $m$ , we tag each part of the message.



**Claim.** This is **not** **UFCMA**.

**Proof.** Consider the “reordering attack”:

- Query  $m_1 \| m_2$  and get  $t_1 \| t_2$ .
- Output  $(m^*, t^*)$  where  $m^* = m_2 \| m_1$  and  $t^* = t_2 \| t_1$ .

This adversary wins the **CMA game** every time since this is a valid message/tag pair for messages with length  $2\ell$ . \*

#### Attempt 2

We now try to include indices when tagging  $m$  to fix the problem. First, break the message into  $m = m_1 \| m_2 \| \dots \| m_s$  in blocks of length  $\ell/2$ . Suppose  $t_i = F_k(m_i \| \langle i \rangle)$  with  $\langle i \rangle$  being length  $\ell/2$ .

**Notation.** Given an integer  $i \in \mathbb{Z}$ ,  $\langle i \rangle$  is its binary representation in bits.<sup>a</sup>

<sup>a</sup>The length can vary and should be specified.

Additionally, we demand that  $\text{Ver}_k(\cdot)$  to verify individual blocks and make sure indices are in the correct order.

**Claim.** This is still **not** **UFCMA**.

**Proof.** Consider the “truncation attack”:

- Query  $m = m_1 \| m_2$ , get  $t = t_1 \| t_2$ .
- Output  $m^* = m_1, t^* = t_1$ .

This adversary wins the **CMA game** again, obviously. \*

**Attempt 3**

To prevent “truncation attack”, we try to also include the length (number of blocks) when tagging the message  $m$ . Again, break the message into  $m = m_1 \| m_2 \| \dots \| m_s$  in blocks of length  $\ell/3$ , and set  $t_i = F_k(m_i \| \langle i \rangle \| \langle s \rangle)$ , where both  $\langle i \rangle$  and  $\langle s \rangle$  are in  $\ell/3$  bits.

Additionally, we demand that  $\text{Ver}_k(\cdot)$  to verify individual blocks, make sure indices are in the correct order, and also the number of the total number of blocks matches with the entire  $m$ .

**Claim.** This is still **not** **UFCMA**.

**Proof.** Consider the “mix-match attack”:

- (a) Query  $m = m_1 \| m_2$  and get  $t_1 \| t_2$ .
- (b) Query  $m' = m'_1 \| m'_2$  and get  $t'_1 \| t'_2$ .
- (c) Output  $m^* = m_1 \| m'_2$  and  $t^* = t_1 \| t'_2$ .

This adversary wins the **CMA game** again since not only the order is preserved, the length is fixed to be the same too (compared to the previous attempt).  $\otimes$

**Attempt 4**

Finally, we give each message a random identifier. Again, we break the message into  $m = m_1 \| m_2 \| \dots \| m_s$  in blocks of length  $\ell/4$ , and let  $r \leftarrow \{0, 1\}^{\ell/4}$  be uniform. Set  $t_i = F_k(m_i \| \langle i \rangle \| \langle s \rangle \| r)$  for each  $i$  and let  $t = r \| t_1 \| t_2 \| \dots \| t_s$ .

Additionally, we demand that  $\text{Ver}_k(\cdot)$  to verify  $i$  and  $s$  as before, and also verify  $r$  for all block.

**Theorem 3.3.2.** This is a **unforgeable MAC** if  $F$  is a **PRF**.

**Proof idea.** The extra information we include in each block prevents the previously described attacks, and no more attacks are possible, i.e., a forgery  $m^*, t^*$  must include a block  $m_i \| r \| \langle i \rangle \| \langle l \rangle$ . For the full proof, see [KL20, Page 117-120]  $\blacksquare$

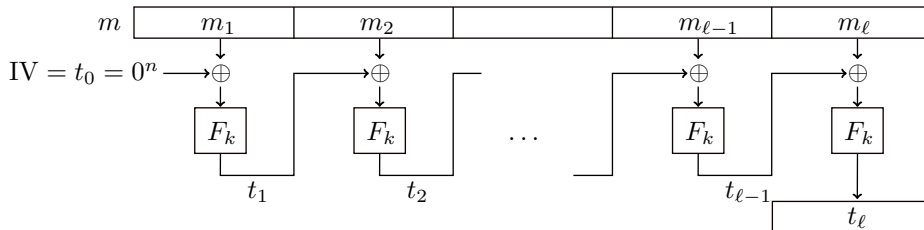
**Remark.** Although this is a solution, but now the tag is 4 times longer than the message, so we seek for a better solution.

**Lecture 15: CBC-MAC and Authenticated Encryption****3.3.6 Cipher Block Chaining-Message Authentication Codes**

6 Mar. 10:30

One potential solution is the following, which is used more in practice. Let  $m \in \{0, 1\}^{\ell \cdot n}$  such that  $m = m_1 \| m_2 \| \dots \| m_\ell$  with  $|m_i| = n$ , i.e., consider the block length being  $n$  and the message length being  $\ell \cdot n$ , and

- $F: \mathcal{K} \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  be a **PRF**;
- $\text{Gen}(1^n)$ : chooses  $k \leftarrow \mathcal{K}$  uniformly at random for the **PRF**;
- $\text{Tag}_k(m)$ : set  $t_0 = 0^n$  and  $t_i = F_k(t_{i-1} \oplus m_i)$  where  $i \in [\ell]$ , output at the end  $t_\ell$ ;
- $\text{Ver}_k(m, t)$  is **canonical** that it could just run the whole thing again and verify that we have the same tag since  $F_k$  is deterministic.



**Notation** (Cipher block chaining-message authentication codes). The above **MAC** is called *Cipher block chaining-message authentication codes*, or **CBC-MAC** for short.

**Remark.** It is necessary to have a fixed  $t_0$ , otherwise the system is totally insecure.

**Proof.** If we do allow a non-fixed  $t_0$ , then consider now our tag is defined as  $(t_0, t_\ell)$  for  $m$ . Now, if we change  $t_0$  to  $t_0 + \Delta$ , it would be valid for  $(m_1 + \Delta) \| m_2 \| \dots \| m_\ell$  since

$$t_1 = F_k(t_0 \oplus m_1) = F_k((t_0 \oplus \Delta) \oplus (m_1 \oplus \Delta)) = F_k(t'_0 \oplus m'_1),$$

and everything else in the chain would be the same, hence it's valid, i.e., the adversary wins.  $\otimes$

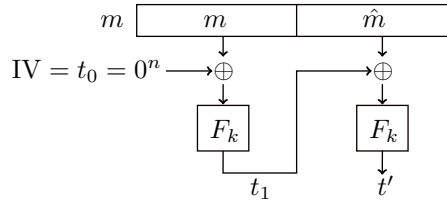
With this, we see that it is also necessary to **not** output all  $t_i$ 's, since we can consider replacing  $t_i$  and  $m_i$  to  $t_i + \Delta$  and  $m_i + \Delta$ .

So, with these caveats, we have the following.

**Theorem 3.3.3.** If  $F$  is a **PRF**, then **CBC-MAC** is **UFCMA** for message of length exactly  $\ell \cdot n$  for a fix  $\ell$ .

The proof is tedious, but it's quite obvious to see why this shouldn't work for arbitrary length by considering the "length extension attacks":

- query  $m$  (one block) and get tag  $t = F_k(m)$ ;
- query  $m' = \hat{m} \oplus t$  and get tag  $t' = F_k(t \oplus \hat{m})$ ;
- Now consider the forge  $m^* = m \| \hat{m}$ ,  $t^* = t'$ , which is valid since:



**Remark.** This is the only attack that an adversary can do for **CBC-MAC**.

Hence, if we can resolve the above issue, we can support arbitrary length message.

**Notation** (Prefix-free encoding). The *prefix-free encoding* is an encoding of a message such that no valid message is a prefix of another.

For example, given  $m$ , we can encode it into  $\langle |m| \rangle \| m \| 0 \dots 0$ , where  $\langle |m| \rangle$  denotes the length of  $m$  in binary and  $0 \dots 0$  is used for padding to block length.

**Example.** Suppose  $n = 4$  and  $m = 001$ , so  $\langle |m| \rangle = \langle 3 \rangle = 0011$  and the encoding of  $m$  is  $0011 \| 0010$ . To decode, we just read the first block to obtain the information of  $|m|$  and read the second block.

It is easy to see that the above encoding is **prefix-free**.

**Claim.** The encoding  $m \mapsto \langle |m| \rangle \| m \| 0 \dots 0$  is **prefix-free**.

**Proof.** Suppose  $\hat{m}$  (the encoding of  $m$ ) is a prefix of  $\hat{m}'$  (the encoding of  $m'$ ), then  $\hat{m}_1 = \hat{m}'_1$ ,<sup>a</sup> and hence  $|m| = |m'|$ . But  $\hat{m}, \hat{m}'$  have the same number of block, meaning  $\hat{m} = \hat{m}'$ .  $\otimes$

<sup>a</sup>This denotes the first block.

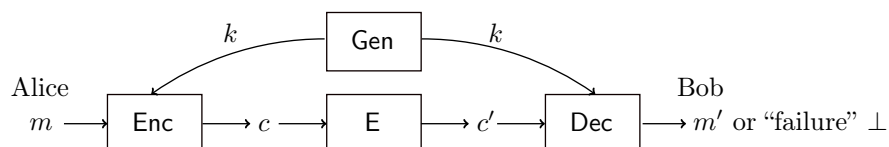
Now we see an improvement of **Theorem 3.3.3** based on the above finding.

**Theorem 3.3.4.** If we use a [prefix-free encoding](#) before tagging, and remove encoding after verifying, then [CBC-MAC](#) is [UFCMA](#) for arbitrary length message.

**Remark.** In reality, there are a better option, but theoretically this works and the above encoding is not that bad since we only need to add  $\log |m|$  bits.

### 3.4 Authenticated Encryption Scheme

After seeing [MAC](#), which is all about integrity, we now ask whether we can combine it with [encryption](#), i.e., we want both confidentiality and integrity. What we want is the following.



This leads to the so-called [cryptosystem](#).

**Definition 3.4.1 (Cryptosystem).** A *cryptosystem* is defined as a tuple  $\Pi = (\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$  where

- $\text{Gen}(1^n)$  output a secret key  $k$ ;
- $\text{Enc}_k(m)$  output a ciphertext  $c$ ;
- $\text{Dec}_k(c')$  output either a message  $m'$  or a special failure symbol  $\perp$ .<sup>a</sup>

<sup>a</sup>Indicating that  $c'$  is inauthentic.

For “correctness”, we can still use [Definition 1.2.1](#), i.e., we require that for all  $k$  and  $m$ ,

$$\text{Dec}_k(\text{Enc}_k(m)) = m.$$

**Note.** We don’t allow  $\text{Dec}_k(\text{Enc}_k(m)) = \perp$  to be specific, since  $\text{Enc}_k(m)$  should always be a valid ciphertext.

As for “security”, we now combine what we discussed so far:

- Confidentiality: Same as [CPA-security](#).
- Authenticity: Attacker shouldn’t be able to produce an “authentic-looking” ciphertext on its own, even after seeing many ciphertexts from the sender (on messages of its choice).

We see that there’s no “tag” here, i.e., we need to define a similar game as [CMA game](#) in this context.

#### 3.4.1 Ciphertext Forgery Game

Formally, we have the following.

**Definition 3.4.2 (Ciphertext forgery game).** The *ciphertext forgery game* for an adversary  $A$  let  $A$  has access to  $\text{Enc}_k(\cdot)$  oracle, and  $A$  wins if it can produce a new and authentic-looking ciphertext.

Then, naturally, we have the following definition.

**Definition 3.4.3 (Advantage).** Given an adversary in a [ciphertext forgery game](#), the *advantage*  $\text{Adv}_{\Pi}^{\text{UNF}}(A)$  is given by

$$\text{Adv}_{\Pi}^{\text{UNF}}(A) := \Pr_{k \leftarrow \text{Gen}(1^n)} (A^{\text{Enc}_k(\cdot)} \text{ “forges”}),$$

where “forges” means that the output  $c^*$  satisfies  $\text{Dec}_k(c^*) \neq \perp$  and  $c^*$  was not a reply for the  $\text{Enc}_k(\cdot)$  oracle.

**Definition 3.4.4 (Unforgeable).** A cryptosystem  $\Pi = (\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$  is *unforgeable* if for all probabilistic polynomial time adversary  $A$ , the advantage is negligible.

### 3.4.2 Authenticated Encryption Scheme

Bringing everything together, we have the following.

**Definition 3.4.5 (Authenticated encryption scheme).** A cryptosystem  $\Pi = (\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$  is an *authenticated encryption (AE) scheme* if it is CPA-secure and unforgeable.

Now we would like to construct AE schemes. We want to combine the existing construction of CPA-secure schemes and UFCMA MACs in a modular way.

**Remark.** This is not what people do in reality, since we might want everything in one-shot in a more efficient way.

## Lecture 16: AE Scheme Construction and Cryptographic Hashing

Now, we explore different approaches to AE constructions. Assume we have an encryption scheme  $\Pi^E = (\text{Gen}^E, \text{Enc}^E, \text{Dec}^E)$  and a MAC  $\Pi^M = (\text{Gen}^M, \text{Tag}^M, \text{Ver}^M)$ , we want to use these to construct  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  which is AE.

8 Mar. 10:30

**Note.** It is important to use different randomness for the encryption scheme and MAC.

### Candidate 1 – Encrypt and Tag

Consider the following.

- $\text{Gen}(1^n)$ :  $k_E \leftarrow \text{Gen}^E(1^n)$ ,  $k_M \leftarrow \text{Gen}^M(1^n)$ ,  $k = (k_E, k_M)$ .
- $\text{Enc}_k(m)$ : compute  $c = \text{Enc}_{k_E}^E(m)$  and  $t = \text{Tag}_{k_M}^M(m)$ , output the ciphertext  $(c, t)$ .
- $\text{Dec}_k(c, t)$ : compute  $m = \text{Dec}_{k_E}^E(c)$ . If  $\text{Ver}_{k_M}^M(m, t) = 1$ , output  $m$ , else output  $\perp$ .

**Claim.** This construction approach of AE is not CPA secure nor unforgeable.

**Proof.** We see the following.

- Although the encryption scheme we use is CPA secure, the MAC we use does not guarantee anything about confidentiality, e.g., it may contain a part of the message. If Tag is deterministic, then it also directly breaks CPA security from the same proof as Theorem 2.3.1.
- This construction is also not necessarily unforgeable since similarly, the encryption scheme is not guaranteed to be unforgeable. For example, ciphertext can have extra parts (junk) that one can “tweak” without causing failure, hence we get a new ciphertext for the same message.

⊛

**Remark.** If we can somehow enforce the tag to be the same length (as we did before), then we might be okay. But this rule out of the generic theorem we want, i.e., we want to allow any kind of UFCMA MAC to be used.

So, we might we to do things sequentially.



**Candidate 2 – Tag, then Encrypt**

This was a popular choice a few decades ago. Consider the following.

- $\text{Gen}(1^n)$ :  $k_E \leftarrow \text{Gen}^E(1^n)$ ,  $k_M \leftarrow \text{Gen}^M(1^n)$ ,  $k = (k_E, k_M)$ .
- $\text{Enc}_k(m)$ : compute  $t = \text{Tag}_{k_M}^M(m)$ , then output ciphertext  $c = \text{Enc}_{k_E}^E(m||t)$ .
- $\text{Dec}_k(c, t)$ : compute  $\text{Dec}_{k_E}^E(c)$ , parse as  $m||t$ . If it is well formatted, check if  $\text{Ver}_{k_M}^M(m, t) = 1$ . If so, output  $m$ , else output  $\perp$ .

**Claim.** This construction approach of AE is not CPA secure nor unforgeable.

**Proof.** We see the following.

- This AE construction is not necessarily CPA secure because the tag can be of different lengths for different messages. The concatenation of the message and tag then might be of different lengths. This could break CPA security. In addition, this AE construction might provide different types of error messages, which can be a potential vulnerability.
- This AE construction is not necessarily unforgeable for the same reason as candidate 1. The ciphertext can be tweaked, and MAC still might accept a tweaked ciphertext into a valid one.

⊛

Now, we try a different order of doing encryption and tagging.

**Candidate 3 – Encrypt, then Tag**

Consider the following.

- $\text{Gen}(1^n)$ :  $k_E \leftarrow \text{Gen}^E(1^n)$ ,  $k_M \leftarrow \text{Gen}^M(1^n)$ ,  $k = (k_E, k_M)$ .
- $\text{Enc}_k(m)$ : compute  $c = \text{Enc}_{k_E}^E(m)$ , and  $t = \text{Tag}_{k_M}^M(c)$ , output ciphertext and the tag pair  $(c, t)$ .
- $\text{Dec}_k(c, t)$ : if  $\text{Ver}_{k_M}^M(c, t) = 1$ , output  $\text{Dec}_{k_E}^E(c)$ , else output  $\perp$ .

**Theorem 3.4.1.** If  $\Pi^E$  is CPA secure and  $\Pi^M$  is strongly unforgeable, then this cryptosystem (encrypt then tag) is a valid AE scheme.

**Proof sketch.** We see that this AE construction is

- CPA secure because the MAC function is only dependent on  $c$  and independent of the encryption key  $k_E$ .<sup>a</sup> Hence, confidentiality guarantee by the encryption scheme is preserved;
- unforgeable because is indicated directly from the strong unforgeability of MAC.

■

<sup>a</sup>This is why independent is important.

**Remark.** Some disadvantages of the "encrypt, then tag" AE scheme are

- the two encryption keys must be independent, which is tricky in practice;
- the tag and encryption have to be done sequentially, i.e., the AE needs two passes on data which is inefficient.

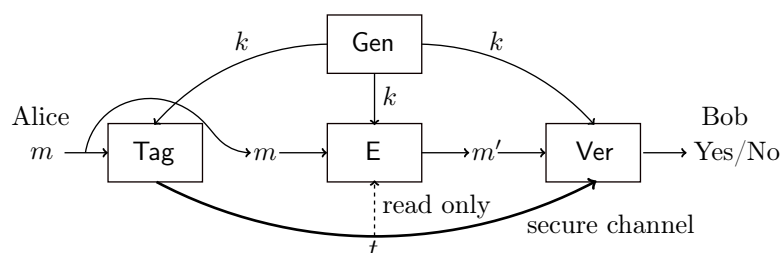
**Note.** There are other AE proposals, we name a few here.

- **Galois counter mode** (GCM).
- **Offset codebook mode** (OCB).
- **Integrity aware parallelizable mode** (IAPM).

## Chapter 4

# Symmetric Public Key Message Authentication

The [encryption scheme](#) we have talked about relies heavily on secret keys. However, key exchanges can be difficult in real life, and we want to construct an [AE scheme](#) that relies less on secret keys. To do this, we first focus on the integrity, i.e., we want a [UFCMA MAC](#) that does not rely on secret key. Consider the following new public key model that shares tags through a secure channel that is read only to the adversaries:



Getting integrity first already has some implication.

**Example.** Downloading a free software over P2P networks.

**Proof.** Everyone can see the data (message, ciphertext, whatever), but it's crucial to know that the data we're downloading is from the authority. \*

In this new public key [MAC](#), instead of using [PRF](#) to generate a tag for the message, we use the so-called [cryptographic Hash functions](#). These Hash functions are different from Hash functions (tables) used in data structures and algorithms: while collisions happened in both cases, sometimes we're able to compute what data will collide with each other in a Hash table, this is prohibited in the former case.<sup>1</sup>

**Intuition.** The adversary can't generate a different message such that it with the original tag is a valid pair.

### 4.1 Cryptographic Hash Family

We now define the main object we are going to focus on in this chapter formally.

**Definition 4.1.1** (Cryptographic Hash function). A *cryptographic hash function* is a deterministic polynomial time function  $H: \mathcal{K} \times X \rightarrow Y$  with  $|X| > |Y|$  such that  $H_k$  satisfies [collision resistance](#) for every  $k$ .

<sup>1</sup>We see that this is another hardness assumption!

**Intuition (Collision resistance).** Given any probabilistic polynomial time algorithm  $A$ , a function  $f$  is *collision resistance* if  $A$  can't find  $x \neq x'$  such that  $f(x) = f(x')$ .

Later, we will formally define *collision resistance* in Definition 4.1.2.

**Notation** (Cryptographic Hash family). We sometimes call Definition 4.1.1 a *Hash family* since it's actually a family of functions we're going to use over keys  $k \in \mathcal{K}$ .

Notice that  $H_k$  are compression functions since  $|X| > |Y|$ , so by the *pigeon-hole principle*, collisions will happen after the mapping, and hence it's reasonable to ask for *collision resistance*.

**Intuition.** Our goal is to make the adversary can't generate the same tag with a different message, and this is done by *collision resistance*.

**Note.**  $k, X, Y$  can depend on the security parameter.

The key generation algorithm  $\text{Gen}(1^n)$  outputs  $k \in \mathcal{K}$  where  $k$  is not necessarily uniformly random. Usually we have  $Y = \{0, 1\}^n$  and there are two common choices of  $X$  spaces:

- $X = \{0, 1\}^\infty$ :  $|X| = \infty, |Y| = 2^n$ ;
- $X = \{0, 1\}^{2n}$ :  $|X| = 4^n, |Y| = 2^n$ .

## Lecture 17: Cryptographic Hash Functions & Merkle-Damgård Construction

### 4.1.1 Collision Resistance and Second Pre-Image Resistance

13 Mar. 10:30

We now discuss the effectiveness of *hash families* through the concepts of *collision resistance* and *second pre-image resistance*.

**Definition 4.1.2 (Collision resistance).** A *hash family*  $(\text{Gen}, H)$  is said to be *collision resistant* if for all probabilistic polynomial time adversaries  $A$ ,

$$\text{Adv}_H^{\text{CR}}(A) = \Pr_{k \leftarrow \text{Gen}(1^n)}(A(k) \text{ outputs a collision}) = \text{negl}(n),$$

where a collision is some  $x$  and  $x'$  where  $H_k(x) = H_k(x')$  but  $x \neq x'$ .

**Remark.** It's critical to remember here that  $k$  is public. This means that the adversary  $A$  has access to  $k$ .

**Definition 4.1.3 (Second pre-image resistance).** A *hash family*  $(\text{Gen}, H)$  is said to be *second pre-image resistant* (or target collision resistant) if for all probabilistic polynomial time adversaries  $A$ ,

$$\text{Adv}_H^{\text{SPR}}(A) = \Pr_{k \leftarrow \text{Gen}(1^n), x \leftarrow X}(A(x, k) \text{ outputs a collision involving } x) = \text{negl}(n),$$

where a collision involving  $x$  is some  $x'$  such that  $H_k(x) = H_k(x')$  but  $x \neq x'$ .<sup>a</sup>

<sup>a</sup>Note that  $x$  is given, and is chosen at random from some distribution  $X$ .

**Remark.** Once again,  $k$  is public.

**Problem.** What's the difference between these two definitions?

**Answer.** Against [collision resistance](#), an adversary has to produce a collision of any two values in the domain. Against [second pre-image resistance](#) however, an adversary has to find a collision for some  $x$  chosen at random.

In fact, it's not hard to see that any [hash family](#) that is [collision resistant](#) is also [second pre-image resistant](#), i.e., [second pre-image resistance](#) has a weaker level of security.  $\otimes$

## 4.2 Forming Attacks

Now that we have definitions in place, we want to ask the following question: what sort of attacks could break [hash families](#) which are [collision resistant](#) and/or [second pre-image resistant](#)? Given a [hash function](#), how long might it take to discover a collision?

### 4.2.1 Pigeon-Hole Attack on Collision Resistant Families

As we have seen before, suppose we have some [hash functions](#)  $H_k: X \rightarrow \{0,1\}^\ell$ . In this attack, an adversary tries  $2^\ell + 1$  possible inputs where every input is a member of the domain space  $X$ .<sup>2</sup> By the [pigeon-hole principle](#), we know that we must have at least one collision.

**Remark.** This attack takes  $O(2^\ell)$  attempts. Is it possible to generate a more efficient attack?

### 4.2.2 Birthday Paradox Attack on Collision Resistant Families

Once again, suppose that we have some [hash functions](#)  $H_k: X \rightarrow \{0,1\}^\ell$ . If we choose  $q$  inputs from the domain space  $X$ , then we'll have  $\binom{q}{2}$  possible pairs, which is approximately  $\frac{q^2}{2}$  from the [birthday paradox](#). Each one of these pairs represents a chance of colliding with a probability of  $\frac{1}{2^\ell}$ .

Suppose that we choose  $q = \sqrt{2^{\ell+1}} \approx 2^{\ell/2}$ , then the probability of a collision is now approximately

$$\frac{q^2}{2} \cdot \frac{1}{2^\ell} = \frac{(\sqrt{2^{\ell+1}})^2}{2} \cdot \frac{1}{2^\ell} = \frac{2^\ell}{2^\ell} = 1,$$

hence when running a [birthday paradox](#) attack against a [collision resistant hash family](#), we know that we're likely to have a collision when the attacker attempts  $\sqrt{2^{\ell+1}}$  queries.

**Example.** If  $\ell = 128$ , an attacker would need  $2^{64}$  queries. This number of queries is computationally feasible, which means that [collision resistant hash functions](#) should not use key spaces this small.

**Example (SHA-1).** If we have  $\ell = 166$ , then we need approximately  $2^{80}$  queries. This key space size is used by SHA-1 and is considered to be on the edge of feasible.

**Example (SHA-256).**  $\ell = 256$  is computationally infeasible and is used in systems like SHA-256.

### 4.2.3 Attacks on Second Pre-Image Resistant Families

Both of the attacks described above discuss [collision resistant](#) families.

**Problem.** What about attacks on [second pre-image resistant](#) families?

**Answer.** For these [hash families](#), [pigeon-hole](#) attacks are effective, but [birthday paradox](#) attacks are not effective.  $\otimes$

For [collision resistant](#) systems, adversaries only need to find an arbitrary collision. Every  $x_n$  drawn from the domain space potentially conflicts with  $x_1, x_2, x_3, \dots, x_{n-1}$  which were drawn previously. For [second pre-image resistant](#) families however,  $x_1, x_2, x_3, \dots, x_{n-1}$  are irrelevant, and we only care if  $x_n$

<sup>2</sup>Remember also that  $|X| \geq 2^\ell$  is true for a [hash function](#).

collides with the original  $x$  which was provided as input to  $A$ . Therefore,  $\ell = 128$  is considered to be on the lower end of acceptable key space sizes for [hash families](#) that are [second pre-image resistant](#).

### 4.3 Merkle-Damgård Construction

Sometimes we may be interested in hashing messages with variable sizes.

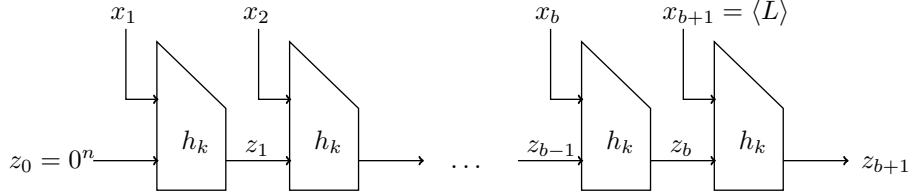
**Example.** Suppose that we are sending a package over the internet and want to create a check sum by hashing the contents of package. This check sum will allow the receiver of the package to verify that the package they received wasn't corrupted by hashing its contents and comparing the result to the check sum. In this use case, the size of the package can vary.

**Problem.** Is there a way that we can construct a [hash function](#) which can accept variable length messages using a [hash function](#) which accepts messages of a fixed-length?

**Answer.** The solution here is the so-called *Merkle-Damgård construction*. ⊗

cite

Suppose we have some [hash function](#)  $h: \mathcal{K} \times \{0, 1\}^{2n} \rightarrow \{0, 1\}^n$  which is [collision resistant](#). If  $x$  is our message, suppose then that we split  $x$  into blocks  $x_1 \| x_2 \| x_3 \dots \| x_b$ , where we let  $L$  represent the length of (un-padded)  $x$  such that  $b = \lceil L/n \rceil$ . We can then implement the following algorithm to “chain” the blocks together



Let this new construction be the function  $H$ , and note the following.

**Note.** We append 0's to  $x$  to make it of length  $bn$  so  $|x_i| = n$  indeed. Let the “initialization value”  $z_0 = 0^n$  and compute  $z_i = h(z_{i-1} \| x_i)$  for all  $1 \leq i \leq b+1$ , where  $x_{b+1} = \langle L \rangle \in \{0, 1\}^n$  is the  $n$ -bit binary representation of  $L$ . Finally, output  $H_k(x) := z_{b+1}$  as the hash value.

**Theorem 4.3.1.** If  $(\text{Gen}, h)$  is [collision resistant](#), then  $(\text{Gen}, H)$  is also [collision resistant](#).

**Proof.** Let  $A$  be an arbitrary probabilistic polynomial time attacker against  $(\text{Gen}, H)$ , we build a new attacker  $A'$  against  $(\text{Gen}, h)$  such that  $A'(k)$  receives a hash key  $k$  and runs  $A(k)$ , which outputs two distinct strings  $x \neq x'$ .

We show that whenever  $H$  succeeds, our constructed attacker  $A'$  against  $h$  also succeeds. Then from the fact that  $(\text{Gen}, h)$  is [collision resistant](#), so is  $(\text{Gen}, H)$ , i.e., if  $H_k(x) = H_k(x')$ , we must describe  $A'$  which obtains (from  $x, x'$ ) two distinct  $2n$ -bit strings  $w \neq w'$  such that  $h_k(w) = h_k(w')$ .

Let  $L = |x|$ ,  $L' = |x'|$  be the length of the messages, and let  $z_i, z'_i$  be the intermediate values for input  $x$  and  $x'$ , respectively. Recall that  $z_{b+1} = H_k(x) = H_k(x') = z'_{b'+1}$ .

- If  $L \neq L'$ , then we know that  $z_b \| \langle L \rangle \neq z'_{b'} \| \langle L' \rangle$  and  $h(z_b \| \langle L \rangle) = z_{b+1} = z'_{b'+1} = h(z'_{b'} \| \langle L' \rangle)$ , so  $w = z_b \| \langle L \rangle$  and  $w' = z'_{b'} \| \langle L' \rangle$  is a collision in  $h_k$ .
- If  $L = L'$ , then  $x$  and  $x'$  have the same number of blocks ( $b = b'$ ). We check whether  $z_b = z'_b$ :
  - If not, then by the same logic above,  $w = z_b \| \langle L \rangle$  and  $w' = z'_b \| \langle L' \rangle$  form a collision in  $h_k$ .
  - Otherwise, we have  $z_b = z'_b$ , and we “work backwards” from there: we have  $h(z_{b-1} \| x_b) = h(z'_{b-1} \| x'_b)$ , so we check whether  $z_{b-1} \| x_b = z'_{b-1} \| x'_b$ . If not, we have found a collision in  $h_k$ , and if so, we have  $z_{b-1} = z'_{b-1}$  and continue working backwards.

**Claim.** There is some  $i$  such that  $h(z_{i-1} \| x_i) = h(z'_{i-1} \| x'_i)$  but  $z_{i-1} \| x_i \neq z'_{i-1} \| x'_i$ .

**Proof.** If not, then all blocks would satisfy  $z_{i-1} \| x_i = z'_{i-1} \| x'_i$ , i.e.,  $x_i = x'_i$  for  $1 \leq i \leq b$ , hence  $x = x'$ , a contradiction.  $\otimes$

## Lecture 18: Hash-and-Mac, HMAC, Public Key Cryptography

With the [Merkle-Damgård construction](#), we see *the power of hash functions*: we can now transform arbitrary-length data to a fixed-length (very versatile). 15 Mar. 10:30

**Remark.** Hence, we can now use [hash functions](#) to turn a primitive that can only handle a fixed number of bits into one that can take arbitrary lengths.

One such example is [MACs](#).

### 4.4 Arbitrary Length UFCMA MAC

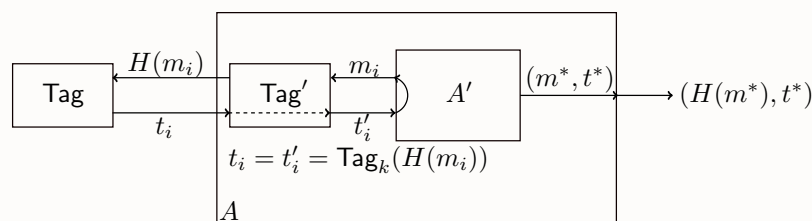
As previously seen. Recall the [CBC-MAC](#).

Suppose we now have a [hash function](#)  $H: X \rightarrow \{0,1\}^{n^3}$  which is [collision resistant](#), and a fixed-length [MAC](#)  $\Pi = (\text{Gen}, \text{Tag}, \text{Ver})$ . Naturally, to tag an arbitrary length  $m \in \{0,1\}^*$ , simply output  $\text{Tag}_k(H(x))$ .

**Theorem 4.4.1.** This composition gives a [UFCMA MAC](#).

**Proof.** We prove this by reduction, where we must prove security of both components. First, from  $\Pi = (\text{Gen}, \text{Tag}, \text{Ver})$  (fixed-length), we define  $\Pi' = (\text{Gen}', \text{Tag}', \text{Ver}')$  where  $\text{Tag}'_k(m) = \text{Tag}_k(H(m))$ .

Then, given any probabilistic polynomial time adversary  $A$  against  $\Pi'$ , we built an adversary  $A'$  against  $\Pi$ .



To show that  $A'$  has only [negligible advantage](#), we look at the collision event. Let  $Q$  be the query set of  $A'$ , i.e., the set of all  $m_i$ , and suppose  $m^* \notin Q$ .<sup>a</sup> Then, for any collision event,  $\exists m \in Q$  such that  $H(m^*) = H(m)$ .

There are two cases, i.e., a collision either happen or doesn't happen, where

- if we have this collision, then we have found a collision in the hash function;
- otherwise, winning the game for  $A'$  and for  $A$  would be related.

Formally, we have

$$\begin{aligned} \Pr(A' \text{ forges}) &= \Pr(A' \text{ forges} \wedge \text{collisions}) + \Pr(A' \text{ forges} \wedge \neg \text{collisions}) \\ &\leq \Pr(\text{collision}) + \Pr(A' \text{ forges} \wedge \neg \text{collisions}) \\ &= \text{negl}(n) + \text{negl}(n) \\ &= \text{negl}(n), \end{aligned}$$

where

- $\Pr(\text{collision}) = \text{negl}(n)$  because  $H$  is [collision resistant](#), and

<sup>3</sup>We no longer specify the key while defining the [Hash](#) as we discussed before.

- $\Pr(A' \text{ forges} \wedge \neg \text{collisions}) = \text{negl}(n)$  because  $\Pi$  is **UFCMA**.

■

<sup>a</sup>Since forged message must be new, so if  $m^* \in Q$ , we don't need to include this into our probability.

**Remark.** We see that a fixed-length **MAC** with a **hash function** is an arbitrary length **MAC**!

**Problem.** How can we do this same type of proof for **PRFs**?

### 4.4.1 HMAC

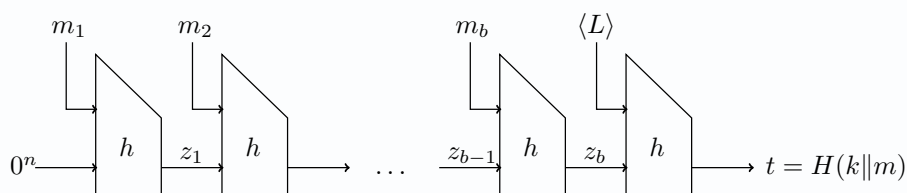
We see that the above construction starts with a fixed-length **MAC** and an arbitrary length **hash function**. Is that possible to use only a **hash function** to directly construct **MACs/PRFs**? In other words, are **hash functions** stronger than **MACs**?

Surprisingly, the answer is yes. Given an arbitrary length **hash function**  $H$ , it's possible to construct an arbitrary length **MAC** directly.

However, one might naively try the following.

**Example (Amateur construction).** To use an arbitrary length **hash function**  $H$  to construct an arbitrary length **MAC**, consider  $\text{Tag}_k(m) = H(k\|m)$ . This is not **UFCMA** in general!

**Proof.** We see that it is easily **forgeable** if  $H$  uses **Merkle-Damgård**.<sup>a</sup> Consider the following.



We see that the length extension attack is trivial, we simply continue the chain to get the correct tag, i.e.,  $m^* = m\|\langle L \rangle\|\text{anything}$  with  $t^* = H(k\|m^*)$ . ⊗

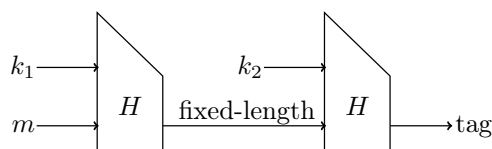
<sup>a</sup>As most mainstreams **hashes** do.

To fix this, someone proposed the so-called **HMAC**, and the idea is that given key  $k = (k_1, k_2)$ , generate  $\text{Tag}_k(m) = H(k_2\|H(k_1\|m))$ .

**Note.**  $H(k_1\|m)$  is fixed-length.

**Theorem 4.4.2.** If  $H(k_1\|m)$  is a **secure** fixed-length **MAC**, and  $H$  has appropriate “pseudorandom properties”, then the **HMAC** is **unforgeable** for arbitrary lengths.

**Remark.** This is informal, and it's beyond our scope. But the intuition should be clear, i.e.,



Now, the length extension attack doesn't work anymore.

To avoid 2 keys  $k_1$  and  $k_2$ , people use heuristics.



**Example.** Take one key  $k$  and let  $k_1 = k \oplus \text{ipad}$  and  $k_2 = k \oplus \text{opad}$ , where **ipad** (inner-pad) and **opad** (outer-pad) are two fixed “randomly looking” string.

## Chapter 5

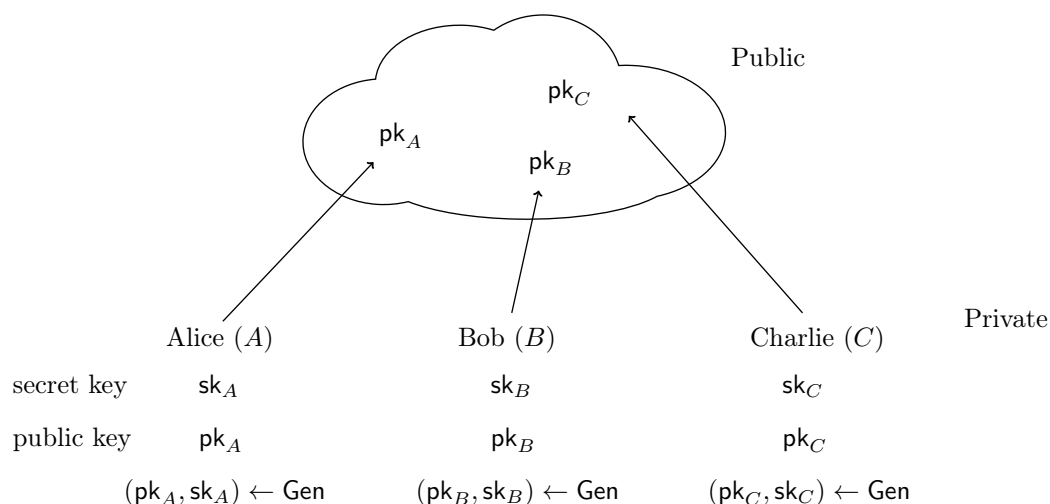
# Asymmetric Public Key Message Security

What we have considered so far (i.e., CPA and MAC) is the so-called **symmetric** key cryptography, i.e., same key for encryption, decryption, or tagging. In this chapter, we again, first focus on the message security. However, symmetric key encryption is not realistic since it isn't scalable, e.g., Internet. So, the following question arose.

**Problem.** Do we really need the same key for encryption/decryption? In other words, what if we used different keys for encryption and decryption?

**Answer.** Actually, we can achieve quite a lot! ⊛

The idea is that we use 2 correlated (i.e. not independently generated) keys, the *public key*  $pk$  and *secret (private) key*  $sk$ .



**Example.** If Bob wants to send  $m$  to Alice, he encrypts it using  $pk_A$ . Only Alice who has  $sk_A$  can decrypt.

**Remark.** In the real world, we *combine* public key and secret key.

Even better, in 1976, Diffie and Hellman come up with the idea of exchanging a secret key over a public communication channel, which makes the whole thing works. The theory behind this is *number theory*!

## Lecture 19: Elementary Number Theory

### 5.1 Number Theory

20 Mar. 10:30

Let's start by developing some basic number theory.

**Definition 5.1.1 (Integer).** The set of *integers* is denoted by  $\mathbb{Z}$ ,

$$\mathbb{Z} := \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}.$$

**Definition 5.1.2 (Natural number).** The set of *natural numbers* is denoted by  $\mathbb{N}$  or  $\mathbb{Z}^+$ ,

$$\mathbb{N} = \mathbb{Z}^+ = \{1, 2, 3, \dots\}.$$

**Note.** We do not include 0 in  $\mathbb{N}$ !

#### 5.1.1 Primes

The following is the most important object in number theory.

**Definition 5.1.3 (Prime).** A *natural number*  $p > 1$  is *prime* if it has no divisors other than 1 and  $p$ .

**Example.** 2, 3, 5, 7, 11, 13, 101,  $\dots$  are all *primes*.

**Definition 5.1.4 (Composite).** A *natural number*  $p > 1$  is *composite* if it is not *prime*.

**Example.** 4, 6, 8, 9, 15, 102,  $\dots$  are all *composites*.

**Remark.** 1 is neither *prime* nor *composite*.

Perhaps one of the most important facts in basic number theory is the following.

**Theorem 5.1.1 (Prime factorization).** Every *integer*  $N > 1$  can be written uniquely (up to ordering) as a product of (powers of) *primes*, i.e.,

$$N = \prod_i p_i^{e_i}$$

where  $p_i$  are *prime* with  $e_i \geq 1$ .

**Example.**  $6 = 2 \cdot 3$ ,  $102 = 2 \cdot 3 \cdot 17$ , and  $72 = 2^3 \cdot 3^2$ .

**Lemma 5.1.1 (Division with remainder).** Let  $a \in \mathbb{Z}$  and  $b \in \mathbb{Z}^+$ . There exist unique *integers*  $q, r$  such that

$$a = bq + r$$

for some  $0 \leq r < b$ .

**Remark.** These *integers* can be efficiently computed, i.e. we can find  $q, r$  in time polynomial in bit-length, e.g.,  $\log_2 a + \log_2 b + O(1)$ .

**Definition 5.1.5** (Greatest common divisor). The *greatest common divisor*  $\gcd(a, b)$  of **integers**  $a, b$  is the largest **integer**  $g$  such that  $g \mid a$  and  $g \mid b$ .

**Example.**  $\gcd(24, 36) = 12$ ,  $\gcd(15, 55) = 5$ ,  $\gcd(12, 35) = 1$ .

**Definition 5.1.6** (Co-prime). Two **integers**  $a$  and  $b$  are said to be *co-prime* (or *relatively prime*) if  $\gcd(a, b) = 1$ .

It's important to note that the naive algorithm for computing **GCD** is inefficient.

**Example.** Consider  $24 = 2^3 \cdot 3$  or  $36 = 2^2 \cdot 3^2$ , where  $\gcd$  is found by taking the least power of **primes** common in both factorization. In this case  $\gcd(24, 36) = 2^2 \cdot 3 = 12$ . This is inefficient as factoring is inefficient.

Next we try to build a more efficient method of computing **GCD**'s. It's based on the following.

**Theorem 5.1.2** (Bezout theorem). Let  $a, b \in \mathbb{Z}^+$ . Then there exist  $x, y \in \mathbb{Z}^+$  such that

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

Moreover,  $\gcd(a, b)$  is the smallest positive **integer** that can be written this way.<sup>a</sup>

<sup>a</sup>That is, **GCD** is the smallest positive linear combination of  $a, b$ .

**Proof.** Let  $I := \{a\hat{x} + b\hat{y} \mid \hat{x}, \hat{y} \in \mathbb{Z}\}$ , in particular,  $a, b \in I$ , so  $I$  has positive **integers**.

Let  $d = ax + by \in I$  for some  $x, y \in \mathbb{Z}$  be the smallest positive **integer** in  $I$ . We must show that  $d \mid a$ ,  $d \mid b$  and if  $d' \mid a$  and  $d' \mid b$ , then  $d' \mid d$  (or alternatively  $d' \leq d$ , both conditions are equivalent).

**Claim.**  $d$  divides every element of  $I$ .

**Proof.** Say  $c = a\hat{x} + b\hat{y} \in I$ . Dividing  $c$  by  $d$ , we have  $c = qd + r$  for some  $0 \leq r < d$ . Hence,

$$r = c - qd = a\hat{x} + b\hat{y} - q(ax + by) = a(\hat{x} - qx) + b(\hat{y} - qy) \in I.$$

But since  $d$  is the smallest positive **integer** in  $I$  we must have  $r = 0$ , therefore  $d \mid c$ . ⊗

**Claim.**  $d$  is the **largest common divisor** of  $a, b$ .

**Proof.** Suppose we have some  $d'$  such that  $d' \mid a$  and  $d' \mid b$ . Then  $d' \mid ax$  and  $d' \mid by$ . So,  $d' \mid ax + by = d$ , and  $d' \leq d$ . ⊗

■

## 5.1.2 Extended Euclid's Algorithm

We now want an efficient algorithm to compute not only  $\gcd(a, b)$ , but also the  $x, y$  coefficients as described in **Bezout theorem**. This is done using **extended Euclid's algorithm**.

**Notation.**  $a \bmod b$  means **remainder of division** of  $a$  by  $b$  ( $b \neq 0$ ).

**Theorem 5.1.3** (Extended Euclid's algorithm). Let  $a, b > 1$  be some **integers**. If  $b \mid a$  then  $\gcd(a, b) = b$ . Now suppose  $b \nmid a$ , then

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

**Proof.** We first note the following.

**Claim.**  $(a, b)$  and  $(b, a \bmod b)$  have the same common divisors.

**Proof.** We need to show that  $d \mid a, d \mid b \Leftrightarrow d \mid b, d \mid (a \bmod b)$ . First,  $d \mid b \Leftrightarrow d \mid b$  is trivial. Also, we know that  $(a \bmod b) = a - qb$  for some  $q \in \mathbb{Z}$ . Now

- if  $d \mid a$  and  $d \mid b$ , then  $d \mid a - qb = (a \bmod b)$ ;
- if  $d \mid b$  (so  $d \mid qb$ ) and  $d \mid (a \bmod b) = a - qb$ , then  $d \mid (a - qb) + qb = a$ .

⊗

So  $(a, b)$  and  $(b, a \bmod b)$  have the same common divisors, and so in particular, the same **greatest common divisor**. ■

This naturally leads to the following (recursive) algorithm for computing **GCD**.

---

**Algorithm 5.1:** Extended Euclidean Algorithm

---

**Data:**  $a, b \in \mathbb{Z}^+$  with  $a \geq b > 0$ .

**Result:**  $x, y$  such that  $ax + by = \gcd(a, b)$

```

1 if  $b \mid a$  then
2   return  $(0, 1)$ 
3 else
4    $a = qb + r$  with  $0 < r < b$                                 // From Lemma 5.1.1
5    $(x', y') \leftarrow \text{ExtendedEuclid}(b, r)$ 
6   return  $(y', x' - y'q)$ 
```

---

**Theorem 5.1.4.** The **extended Euclid theorem** is correct.

**Proof.** Since

$$bx' + ry' = \gcd(b, r) = \gcd(a, b)$$

from **Theorem 5.1.3**, and

$$bx' + (a - qb)y' = ay' + b(x' - qy') = ax + by,$$

so the **extended Euclid theorem** outputs  $x, y \in \mathbb{Z}$  such that  $ax + by = \gcd(a, b)$ . ■

**Remark.** The **extended Euclid theorem** makes a linear number (in the input length) of recursive calls, hence is efficient.

### 5.1.3 Group Theoretic View of Numbers

Consider the following.

**Definition 5.1.7** (Integers modulo  $n$ ). The set  $\mathbb{Z}_n$  of *integers modulo  $n$*  is defined as

$$\mathbb{Z}_n := \{0, 1, 2, \dots, n-1\},$$

which is the set of all possible remainders of division by  $n$ .

**Definition 5.1.8** (Integers co-prime to  $n$ ). The set  $\mathbb{Z}_n^*$  of *integers co-prime to  $n$*  is defined as

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

which is the set of all possible remainders of division by  $n$  that are **co-prime** to  $n$ .

**Example.**  $\mathbb{Z}_6^* = \{1, 5\}$ ,  $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$ ,  $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$ .

**Remark.** For any prime  $p$  we have  $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$ .

**Proof.** Since  $p$  is prime all numbers less than  $p$  (except 0) are not divisible by  $p$ , and so must be co-prime to  $p$ .<sup>a</sup> ⊛

<sup>a</sup>The only factors of  $p$  are 1,  $p$ .

### 5.1.4 Modular Arithmetic

Consider the following.

**Definition 5.1.9** (Equivalence modulo  $n$ ). We define

$$a = b \pmod{n} \Leftrightarrow n \mid a - b,$$

i.e., elements are identified if their remainders of division of  $a$  and  $b$  by  $n$  are the same.

**Notation.**  $a \equiv b \pmod{n}$  and  $a \equiv^n b$  mean the same as  $a = b \pmod{n}$ .

**Remark** (Equivalence Relation). Congruence modulo  $n$  is an equivalence relation.

**Proof.** We see that

- (a) for all  $a \in \mathbb{Z}$ ,  $n \mid 0 = a - a$  so  $a \equiv a \pmod{n}$ ;
- (b) for all  $a, b \in \mathbb{Z}$  if  $a \equiv b \pmod{n}$  that is,  $n \mid a - b$  then  $n \mid b - a$ , i.e.,  $b \equiv a \pmod{n}$ ;
- (c) for all  $a, b, c \in \mathbb{Z}$  if we have  $a \equiv b \pmod{n}$  and  $b \equiv c \pmod{n}$ . Then  $n \mid a - b$  and  $n \mid b - c$ . So  $n \mid (a - b) + (b - c) = a - c$ , hence  $a \equiv c \pmod{n}$ .

⊛

## Lecture 20: Group Theory

**Remark** (Operation of mod). For  $a = a' \pmod{n}$  and  $b = b' \pmod{n}$ , we have

- $a + b = a' + b' \pmod{n}$ ;
- $a - b = a' - b' \pmod{n}$ ;
- $a \cdot b = a' \cdot b' \pmod{n}$ .

We don't have such a definition for division.

**Example.** Observe the counter-example that  $3 \cdot 2 = 15 \cdot 2 \pmod{24}$  however,  $3 \not\equiv 15 \pmod{24}$ .

**Definition 5.1.10** (Invertible).  $b$  is *invertible* if there is some  $c$  such that  $b \cdot c = 1 \pmod{n}$ .

**Notation** (Modular inverse). In this case, we say that  $c$  is the *modular inverse* of  $b$ .

**Note.** In  $\mathbf{R}$ , we don't see inverses of integers being integers; however, with modular arithmetic we

22 Mar. 10:30

can actually define [integer inverses](#).

The following lemma tells us when do we have [inverses](#) for [integers](#).

**Lemma 5.1.2.** Given  $a \geq 1$  and  $N > 1$ ,  $a$  is [invertible](#) if and only if  $\gcd(a, N) = 1$ .

**Proof.** The forward direction is easy. Let  $a \cdot c = 1 \pmod{N}$ , then  $a \cdot c - 1 = N \cdot q$ , implying  $ac - Nq = 1$ , which is  $\gcd(a, N)$  since [GCD](#) is the smallest positive [integer](#) expressible in this way.

For the backward direction, since  $\gcd(a, N) = 1$ , we know that there exists  $x, y$  such that  $x \cdot a + y \cdot N = 1$ , hence  $x \cdot a = 1 \pmod{N}$ , i.e.,  $x$  is the [modular inverse](#). ■

**Corollary 5.1.1.** We can calculate the [modular inverse](#) of [integers](#).

**Proof.** We use the [extended Euclid theorem](#) with [Lemma 5.1.2](#). ■

**Theorem 5.1.5 (Uniqueness of modular inverses).** If  $c, c'$  are both [inverses](#) of  $a$ , then  $c = c' \pmod{N}$ .

**Proof.** Since

$$\begin{cases} a \cdot c = 1 & \Rightarrow N \mid a \cdot c - 1; \\ a \cdot c' = 1 & \Rightarrow N \mid a \cdot c' - 1 \end{cases} \Rightarrow N \mid a \cdot (c - c') \Rightarrow \gcd(N, a) = 1$$

since the [inverse](#) exist. Hence,  $N \mid c - c'$ , so  $c = c' \pmod{N}$ . ■

**Example.** Let  $a = 11, N = 17$ , then

$$(-3) \cdot 11 + \underbrace{2 \cdot 17}_{} = 1 \Rightarrow -3 = 14 \pmod{17}.$$

Verifying, we see that indeed  $11 \cdot 14 = 1 \pmod{17}$ .

**Note.** We take [modulo](#) 17 on both sides of the first equation.

## 5.2 Group Theory

Let's introduce [group](#) formally.

**Definition 5.2.1 (Group).** A *group* is a set  $G$  along with a binary operation  $\circ: G \times G$  for which the following conditions hold.

- Closure: for all  $g, h \in G$ ,  $g \circ h \in G$ .
- Existence of an identity: there exists  $e \in G$  such that for all  $g \in G$ ,  $e \circ g = g = g \circ e$ .
- Existence of inverses: for all  $g \in G$ , there exists an inverse  $h \in G$  such that  $g \circ h = e = h \circ g$ .
- Associativity: for all  $g_1, g_2, g_3 \in G$ ,  $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$ .

**Notation.** We sometimes denote  $e$  as 1.

**Definition 5.2.2 (Abelian group).** A [group](#)  $G$  with operation  $\circ$  is *Abelian* if  $\circ$  is commutative, i.e., for all  $g, h \in G$ ,  $g \circ h = h \circ g$ .

**Note.** When the binary operation is understood, we simply call the set  $G$  a [group](#).

**Definition 5.2.3 (Finite group).** When  $G$  has a finite number of elements, we say  $G$  is a *finite group*.

**Definition 5.2.4** (Infinite group). If  $G$  has infinitely many elements, we say  $G$  is an *infinite group*.

**Definition 5.2.5** (Order). The *order* of a group  $G$  is defined as  $|G|$ .<sup>a</sup>

<sup>a</sup>I.e., the number of elements in  $G$ .

**Example.**  $(\mathbb{Z}_N, +)$  is a group.

**Proof.**  $(\mathbb{Z}_N, +)$  with order  $N$  satisfies:

- ✓  $a + b = b + a \pmod{N}$ ;
- ✓  $a + 0 = 0 + a = a \pmod{N}$ ;
- ✓  $a + (-a) = 0 \pmod{N}$ ;
- ✓  $a + (b + c) = (a + b) + c \pmod{N}$ .

⊛

**Example.**  $(\mathbb{Z}_N^*, \cdot)$  is a group.

**Proof.**  $(\mathbb{Z}_N^*, \cdot)$ , where  $p$  is prime and has order  $p - 1$  satisfies:

- ✓  $a \cdot b = b \cdot a \pmod{N}$ ;
- ✓  $a \cdot 1 = 1 \cdot a = a \pmod{N}$ ;
- ✓  $a \cdot (-a) = 1 \pmod{N}$ ;
- ✓  $a \cdot (b \cdot c) = (a \cdot b) \cdot c \pmod{N}$ .

⊛

**Definition 5.2.6** (Cyclic group). A group  $G$  is *cyclic* if  $\exists g \in G$  such that

$$G = \{1 = g^0, g^1, g^2, \dots, g^{n-1}\}.$$

**Notation** (Generated). We say that a cyclic group  $G$  is *generated* by  $g$ , and written as  $G = \langle g \rangle$ .

**Note.** In the above notation, we assume that  $|G| = n$ . However, we can also define a cyclic group as an infinite group generated by a single element  $g$  and its inverse  $g^{-1}$  and still write  $G = \langle g \rangle$ .

<sup>a</sup>Think about why.

### 5.2.1 Lagrange's Theorem

The first important theorem in group theory is considering the size (order) of the subgroup and its parent.

**Definition 5.2.7** (Subgroup). A group  $G' \subseteq G$  is a *subgroup* if  $(G', \circ)$  is a group.

**Theorem 5.2.1** (Lagrange's theorem). If  $G' \subseteq G$  is a subgroup, then  $|G'| \mid |G|$ .<sup>a</sup>

<sup>a</sup>I.e., the order of subgroup divides the order of the original group.



**Note.** Lagrange's theorem tell us that the orders of sub- and parent groups cannot be too close.

**Example.** Consider  $\mathbb{Z}_p^*$  for prime  $p$  with  $p = 7$ , i.e.,  $\mathbb{Z}_7^* = \{1, \dots, 6\}$ .

**Proof.** Consider the set of powers of 3:  $\{1, 3, 9 = 2, 6, 18 = 4, 12 = 5, 15 = 1\} = \{1, 3, 2, 6, 4, 5\}$ . We see that 3 generate  $\mathbb{Z}_7^*$ .

However, consider the set of powers of 2:  $\{1, 2, 4, 8 = 1, \dots\} = \{1, 2, 4\}$ . Clearly, 2 is not a generator since  $3 \nmid 7$ . ⊛

## 5.2.2 Group Exponentiation

In Cryptography, it is often useful for us to be able to describe a group operation applied, say  $m$ , number of times to a group element  $g \in G$ . We will now state and prove a very useful theorem.

**Theorem 5.2.2.** Let  $G$  be a finite group with  $m = |G|$ . Then for any element  $g \in G$ ,  $g^m = 1$ .

**Proof.** For simplicity, we prove for when  $G$  is Abelian.<sup>a</sup> Fix arbitrary  $g \in G$ , and let  $g_1, \dots, g_m$  be the elements of  $G$ . We claim that

$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m).$$

Observe that  $gg_i = gg_j \Rightarrow g_i = g_j$  since we can multiply both sides by  $g^{-1}$ , so each of the  $m$  elements in parentheses on the right-hand side of the displayed equation is distinct.

Also notice how there are exactly  $m$  elements in  $G$ . We see that the  $m$  elements being multiplied together on the right-hand side are just the elements of  $G$ , just in some permuted order.

Now as,  $G$  is Abelian, we have commutativity and the order of multiplication doesn't matter. We can also "pull out" all occurrences of  $g$  to obtain

$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m) = g^m \cdot (g_1 \cdot g_2 \cdots g_m),$$

giving us  $g^m = 1$ . ■

<sup>a</sup>Though it holds for any finite group.

Theorem 5.2.2 gives us several very useful corollaries that we'll state below whose proofs could be found in [KL20, §7].

**Corollary 5.2.1** (Fermat's little theorem). For all prime  $p$ ,  $\gcd(a, p) = 1$  implies  $a^{p-1} = 1 \pmod{p}$ .

**Definition 5.2.8** (Euler's totient function). The Euler's totient function  $\varphi(N)$  is defined as  $\varphi(N) = |\{a \mid 1 \leq a \leq N, \gcd(a, N) = 1\}|$ .

**Example.** We see that  $|\mathbb{Z}_N^*| = \varphi(N)$ .

**Corollary 5.2.2** (Euler's theorem). If  $\gcd(g, N) = 1$ , then  $g^{\varphi(N)} = 1 \pmod{N}$ .

**Corollary 5.2.3.** For  $m = |G|$ , for all  $g \in G$  and all  $x \in \mathbb{Z}$ ,  $g^x = g^{x \bmod m}$ .

**Proof.** Since we know that  $g^m = 1$  from Theorem 5.2.2. ■

**Corollary 5.2.4.** For  $m = |G| > 1$ ,  $e \in \mathbb{Z}$ , and  $\gcd(e, m) = 1$ . Let  $d = e^{-1} \pmod{m}$ , then the function  $f_e: G \rightarrow G$  defined as  $f_e(g) = g^e$  is a bijection with  $f_d$  being the inverse.

**Proof.** We see that  $f_d(f_e(g)) = f_d(g^e) = (g^e)^d = g^{e \cdot d} = g^{e \cdot d \bmod m} = g^1 = g$ . ■

### 5.2.3 Fast Exponentiation

Say we have a [group](#) element  $g$ , and we want to compute  $g^M$ .

**Example (Naive method).** We simply calculate  $g^M$  as

$$g^M = \underbrace{g \cdot g \cdots g}_{M \text{ times}} = \underbrace{(((g \cdot g) \cdot g) \cdot g) \cdots}_{M \text{ times}}.$$

This clearly not-efficient (polynomial runtime) in the bit-length of  $M$ , so we want something faster. The observation below forms the foundation for fast exponentiation, that'd allow us to do this computation in  $O(\log |M|)$  time.

**Note.** When  $M = 2^m$ ,

$$g^{2^m} = g^{2^{m-1}} \cdot g^{2^{m-1}} = \left(g^{2^{m-1}}\right)^2$$

That is we can get the following terms recursively:

$$g, g^2, g^4, g^8, g^{16}, \dots$$

**Problem.** But how many operations do we end up performing in this case?

**Answer.** We see that  $T(M) = T(M/2) + 1$ , hence  $T(M) = \log M = m$ . ⊛

We can generalize this as follows. Firstly, write  $M$  as

$$M = \sum_{i=0}^{\ell} m_i \cdot 2^i$$

for  $m_i \in \{0, 1\}$ .

**Intuition.** Think about the binary representation of  $M$ .

Then, for  $g^M$ , we see that

$$g^M = \prod_{i=0}^{\ell} g^{m_i \cdot 2^i},$$

hence by applying the above trick for each  $g^{2^i}$  for those  $m_i = 1$ , we only need  $O(\ell^2)$  multiplications altogether if  $\log M = \ell$ .

**Corollary 5.2.5.** Fast exponentiation allows us to compute inverses very fast as  $g^{-1} = g^{|G|-1}$ .

**Proof.** Since  $g^{|G|} = 1$ . ■

**Remark.** For  $\mathbb{Z}_p^*$ , we have an even faster method, i.e., the [extended Euclid algorithm](#).

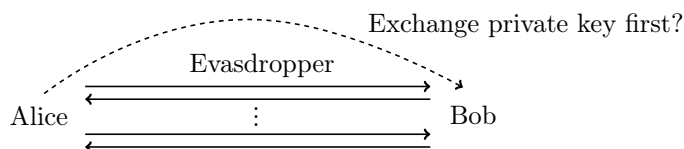
## 5.3 Diffie-Hellman Key Exchange

We now know that how to compute  $g^m$  from  $g$  efficiently, but do we know how to compute  $m$  from  $g^m$ ? This is known as the discrete logarithm function.<sup>1</sup> This problem is conjectured to be extremely difficult, hence the main idea is that, can we utilize the hardness of computing  $m$  from  $g^m$  to build a strong cryptographic protocol?

<sup>1</sup>Think of it as “ $\log_g$ ”  $g^m = m$ .

## Lecture 21: Diffie-Hellman Key Exchange

The answer is yes! In fact, imagine Alice and Bob want to communicate on the public channel, and they want to exchange keys first. 27 Mar. 10:30



The up shot is that, we can use the hardness of computing discrete logarithm to build such a secure key exchange protocol! To do this, we start by fixing a large **cyclic group**  $G$  of known order  $q$ , where the length of  $q$  is approximately the security parameter.<sup>2</sup>

Usually, we let  $G = \mathbb{Z}_{q+1}^*$  for  $q+1$  being a **prime** number.<sup>3</sup> Since  $q$  corresponds to the security parameter, so one stand-alone question which is worth thinking about is the following.

**Exercise.** How to generate any large **prime** number (with some certain number of bits)?

**Answer.** Gauss has solved this for us. Specifically, he showed a fundamental question in number theory: there are  $\pi(n) \approx n / \ln n$  many **prime** numbers between 1 to  $n$ ? By interpreting this as some sorts of “density” ( $\pi(n)/n \approx \ln^{-1} n$ ) of **primes**, modern computer tries different random numbers w.r.t. this density and test whether they are **prime** to generate one. \*

Hence, we’re able to generate a large **prime**  $q+1$ , hence generate  $\mathbb{Z}_{q+1}^*$ . Then we define the following key exchange procedure, called **Diffie-Hellman protocol**.

**Definition 5.3.1** (Diffie-Hellman protocol). The *Diffie-Hellman key exchange protocol* works as

cite

- Alice sends Bob the **group**  $G$ , the generator  $g$ , and  $A = g^a$  where  $a$  is randomly picked in  $G$ ;
- Bob sends back  $B = g^b$  where  $b$  is randomly picked in  $G$ ;
- both of them calculate the key  $K = g^{ab}$ .

$$\begin{array}{ccc} \text{Alice} & \xrightarrow{\langle G \rangle, g, A = g^a \in G} & \text{Bob} \\ a \leftarrow \mathbb{Z}_q & & b \leftarrow \mathbb{Z}_q \\ & \xleftarrow{B = g^b \in G} & \end{array}$$

**Note.** Given  $G$ , we can obtain a generator  $g$  for  $G$ .

**Proof.** There’s an efficient non-trivial trial-and-error approach [KL20, Appendix B.3]. \*

We see that an Eavesdropper can obtain  $A$  and  $B$ , but not  $a, b$ . Moreover, getting  $a, b$  from  $A, B$  requires solving the *discrete logarithm problem*. There is no known polynomial time algorithm. But on the other hand, Alice and Bob now shares the key  $K$  where

$$K = A^b = (g^a)^b = (g^b)^a = B^a.$$

Informally, Eavesdropper will have to take discrete logarithm to break this, i.e., after getting  $\langle G \rangle, g, A = g^a$ , and  $B = g^b$ , to get  $K = g^{ab}$ , the only way is to obtain either  $a$  or  $b$ . Formally, the security primitive is based on the **decisional Diffie-Hellman assumption**.

**Definition 5.3.2** (Decisional Diffie-Hellman assumption). Given a **group**  $G = \langle g \rangle$  with  $q = |G|$ , the *decisional Diffie-Hellman assumption (DDH)* holds if  $(g, g^a, g^b, g^{ab}) \in G^4$  where  $a, b \leftarrow \mathbb{Z}_q$  is indistinguishable from  $(g, g^a, g^b, g^c)$  where  $c \leftarrow \mathbb{Z}_q$ .

<sup>2</sup>Since the length of  $q$  is the security parameter ( $1^n$ ), i.e.,  $q = \log(1^n)$ .

<sup>3</sup>The reason to use  $q+1$  is, if  $q+1$  is a large **prime**, then  $q$  is always an even number.

**Intuition.** The [decisional Diffie-Hellman assumption](#) is trying to say that if  $g^a, g^b$  are uniform random from the [group](#), then the key  $K$  generated as  $g^{ab}$  will also be a uniform random element in the [group](#).

**Conjecture 5.3.1.** DDH holds for  $\mathbb{Z}_{q+1}^*$  with  $q+1$  [prime](#).

An immediate consequence of [Conjecture 5.3.1](#) is that [Diffie-Hellman protocol](#) is secure.

**Note.** Since there are so many sophisticated algorithms for solving the discrete logarithm problem for groups like  $\mathbb{Z}_q^*$ , we normally require  $q > 2^{1000}$  to make the security parameter larger than 1000.

However, we note that our network system works with bit-strings, not [group elements](#) (not even  $\mathbb{Z}_{p+1}^*$ ), i.e., we want a key to be a uniform random bit-string.

**Note (Key derivation).** There are many *key derivation algorithms* which turn a random [group](#) element  $g^{ab}$  into a random bit-string, some involving [hashing](#). This is used in practice heavily for instance inside VPN protocols.

**Remark.** The [Diffie-Hellman protocol](#) might not work alone if the Eavesdropper is active in the real world, but it works fine in our passive setup.

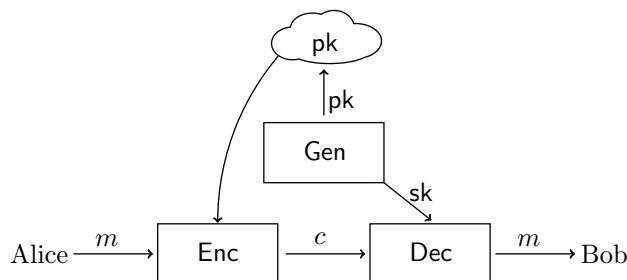
## 5.4 Public Key Message Encryption

By using [Diffie-Hellman protocol](#), we can first exchange the key at the beginning of the communication. But what if we want to have a protocol (Gen, Enc, Dec) that can *directly* handle public key encryption? How to model this? Analogs of [EAV/CPA security](#) in public key setting? Consider the following.

**Definition 5.4.1** (Public key scheme). The *public key scheme* is a [cryptosystem](#)  $\Pi = (\text{Gen}, \text{Dec}, \text{Enc})$  where

- $\text{Gen}(1^n)$ : outputs  $(pk, sk)$ ;
- $\text{Enc}(pk, m)$ : outputs ciphertext  $c$  where  $m \in \mathcal{M}$ ;
- $\text{Dec}(sk, c)$ : outputs  $m \in \mathcal{M}$  (or fail “ $\perp$ ”<sup>a</sup>).

<sup>a</sup>With some protocols happen with [negligibles probability](#).



Naturally, we should have the following.

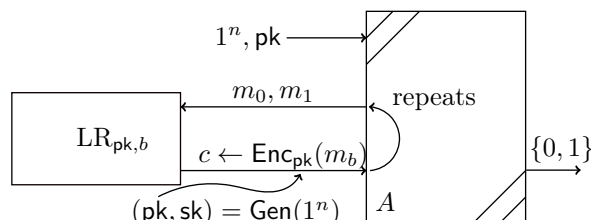
**Definition 5.4.2** (Correctness). A [public key scheme](#) is *correct* if for  $(pk, sk) \leftarrow \text{Gen}(1^n)$  and all  $m \in \mathcal{M}$ ,

$$\text{Dec}(sk, \text{Enc}(pk, m)) = m.$$

### 5.4.1 Chosen Plaintext Attack Secrecy

How about the security definition? Analogous to how we define the [CPA secrecy](#), we again use a [left-right oracle](#) to define the [CPA game](#), but this time, we replace  $k$  by  $pk$ .

**Definition 5.4.3** (Chosen plaintext attack game). The *public key version chosen plaintext attack game* for a probabilistic polynomial time adversary  $A$  against a [public key scheme](#)  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  allows  $A$  to make  $\text{poly}(n)$  number of queries  $(m_0, m_1)$  to a [Left-right oracle](#)  $\text{LR}_{pk,b}(\cdot, \cdot)$  to get a ciphertext for each query, and output a decision bit in the end.



Correspondingly, the [advantage](#) is defined as follows.

**Definition 5.4.4** (Advantage). Given an adversary  $A$  in a [CPA game](#), the *advantage*  $\text{Adv}_{\Pi}^{\text{CPA}}(A)$  in distinguishing “world 0” and “world 1” is given by

$$\text{Adv}_{\Pi}^{\text{CPA}}(A) := \left| \Pr_{(pk, sk) \leftarrow \text{Gen}(1^n)} (A^{\text{LR}_{pk,0}(\cdot, \cdot)} \text{ accepts}) - \Pr_{(pk, sk) \leftarrow \text{Gen}(1^n)} (A^{\text{LR}_{pk,1}(\cdot, \cdot)} \text{ accepts}) \right|.$$

Then, the public key version [CPA secrecy](#) is defined as follows.

**Definition 5.4.5** (Chosen Plaintext attack secrecy). A [public key scheme](#)  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is *chosen plaintext attack secure* if for every probabilistic polynomial time adversary  $A$ , the [advantage](#) is [negligible](#).

### 5.4.2 EAV Secrecy and CPA Secrecy

One might ask, why do we directly allow  $\text{poly}(n)$  queries, rather than start by only one query as in the [EAV game](#)? The reason is that in fact, the number of queries to the [left-right oracle](#) doesn’t matter! I.e., if we have “[EAV](#)”-secrecy, then we also have security under multiple queries.

**Intuition.** The intuition behind the proof is, imagine a many-query attacker  $A$  that makes up to  $q = \text{poly}(n)$  queries. Consider the following worlds:

- Hybrid 0 (left world): all queries  $(m_0, m_1)$  to the [LR oracle](#) are answered by  $c \leftarrow \text{Enc}_{pk}(m_0)$ .
- Hybrid 1: First query  $(m_0, m_1)$  is answered by  $c \leftarrow \text{Enc}_{pk}(m_1)$ , then  $\text{Enc}_{pk}(m_0)$  thereafter.
- Hybrid 2: Similar for the first 2 queries
- $\vdots$
- Hybrid  $q$  (right world): All queries are answered by  $\text{Enc}_{pk}(m_1)$ .

Then, by sequentially bound the [advantage](#), we’re done.

## Lecture 22: CPA Security and ElGamal Cryptosystem

Formally, we have the following.

29 Mar. 10:30

**Theorem 5.4.1.** For public key encryption schemes  $\Pi$ , EAV secrecy is equivalent to CPA secrecy.

**Proof.** Since EAV is weaker than CPA in terms of secrecy, we only need to show that EAV implies CPA. Imagine a many-query attacker  $A$  that makes up to  $q = \text{poly}(n)$  queries, and consider the following worlds:

- Left world ( $H_0$ ): all queries  $(m_0, m_1)$  to the LR oracle are answered by  $c \leftarrow \text{Enc}_{\text{pk}}(m_0)$ .
- Hybrid 1 ( $H_1$ ): First query  $(m_0, m_1)$  is answered by  $c \leftarrow \text{Enc}_{\text{pk}}(m_1)$ , then  $\text{Enc}_{\text{pk}}(m_0)$  thereafter.
- Hybrid 2 ( $H_2$ ): Similarly to  $H_1$ , but answers  $c \leftarrow \text{Enc}_{\text{pk}}(m_1)$  for the first 2 queries.
- $\vdots$
- Hybrid  $q$  ( $H_q$ ): All queries answered by  $c \leftarrow \text{Enc}_{\text{pk}}(m_1)$  (i.e., right world).

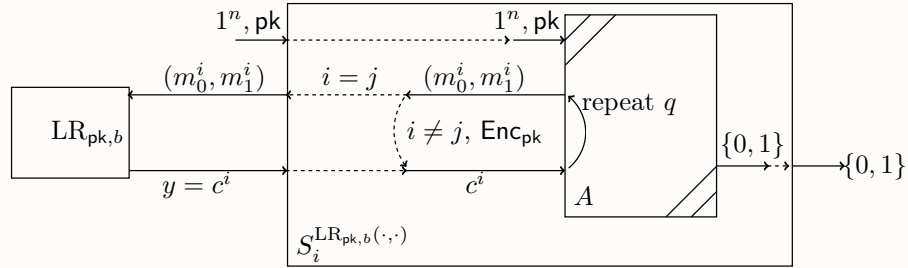
**Intuition.** The difference between  $H_{i-1}$  and  $H_i$  is only how the  $i$ -th query is answered.

Now, we build a “simulator”  $S_i^{\text{LR}_{\text{pk},b}(\cdot, \cdot)}(\text{pk})$  that gets *one query* and simulates either  $H_{i-1}$  or  $H_i$  depending on  $b$ . Specifically, on the  $j$ -th query  $(m_0^j, m_1^j)$  of  $A$ :

- If  $j < i$ ,  $S_i$  runs  $c^j \leftarrow \text{Enc}_{\text{pk}}(m_1^j)$ .
- If  $j > i$ ,  $S_i$  runs  $c^j \leftarrow \text{Enc}_{\text{pk}}(m_0^j)$ .
- If  $j = i$ ,  $S_i$  queries its LR oracle and gives the result to  $A$ .

We see that

- if  $S_i$  is in the left world ( $b = 0$ ), then we perfectly simulate  $H_{i-1}$ ;
- if  $S_i$  is in the right world ( $b = 1$ ), then we perfectly simulate  $H_i$ .

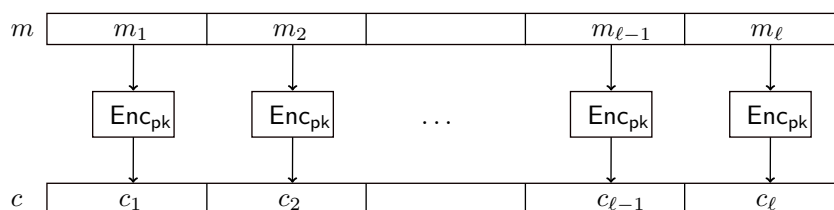


By the triangle inequality, we have

$$\begin{aligned}
 \text{Adv}_{\Pi}^{\text{CPA}} &= |\Pr(A = 1 \text{ in } H_0) - \Pr(A = 1 \text{ in } H_q)| \\
 &= \left| \sum_{i=1}^q (\Pr(A = 1 \text{ in } H_{i-1}) - \Pr(A = 1 \text{ in } H_i)) \right| \\
 &\leq \sum_{i=1}^q |\Pr(A = 1 \text{ in } H_{i-1}) - \Pr(A = 1 \text{ in } H_i)| \leq \sum_{i=1}^q \underbrace{\text{Adv}_{\Pi}^{\text{EAV}}(S_i)}_{\text{negl}(n)} = \text{negl}(n)
 \end{aligned}$$

where  $\text{Adv}_{\Pi}^{\text{EAV}}(S_i) = \text{negl}(n)$  by assumption and  $q \cdot \text{negl}(n) = \text{poly}(n) \cdot \text{negl}(n) = \text{negl}(n)$ .  $\blacksquare$

Theorem 5.4.1 implies that we can encrypt long messages bit-by-bit, block-by-block, or broken up in any other reasonable way. One call to Enc on “long” messages translates to many calls on “short” messages, which is allowed by Theorem 5.4.1.



**Theorem 5.4.2.** Any public key encryption scheme with deterministic  $\text{Enc}_{\text{pk}}(\cdot)$  algorithm can't be CPA secure.<sup>a</sup>

<sup>a</sup>Not even for one query due to Theorem 5.4.1.

**Proof.** Query  $c \leftarrow \text{LR}_{\text{pk},b}(m_0, m_1)$  for any  $m_0 \neq m_1$ . Then run  $c' = \text{Enc}_{\text{pk}}(m_0)$ .<sup>a</sup> If  $c = c'$ , output 0, else 1. This will produce a perfect advantage. ■

<sup>a</sup>Notice that both  $\text{pk}$  and the encryption function are public.

### 5.4.3 ElGamal Encryption

Let's construct a CPA secure public key encryption scheme! We kind of already saw this in Diffie-Hellman, which is formalized by ElGamal — “the public key encryption version of Diffie-Hellman”.

cite

**Intuition.** ElGamal converted Diffie-Hellman into  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$ . Basically message is  $m \in G$ , the “one-time-pad effect” would involve multiplying  $m$  with something random (e.g.,  $K$ ).

Formally, consider the following.

**Definition 5.4.6 (ElGamal encryption scheme).** The ElGamal encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  is defined as

- $\text{Gen}(1^n)$ : choose random  $a \leftarrow \mathbb{Z}_q$ , output  $(\text{pk}, \text{sk}) = (g^a, a)$ ;<sup>a</sup>
- $\text{Enc}(\text{pk}, m)$  for and  $m \in G$ : choose random  $b \leftarrow \mathbb{Z}_q$ , output ciphertext  $(B, c) = (g^b, m \cdot \text{pk}^b)$ ;<sup>b</sup>
- $\text{Dec}(\text{sk}, (B, c))$ : compute  $K = B^{\text{sk}}$ , output  $c \cdot K^{-1} \in G$ .

<sup>a</sup>This is what Alice does in Diffie-Hellman.

<sup>b</sup>We see that  $B$  is random and  $\text{pk}^b$  is essentially the “key”: this is what Bob does in Diffie-Hellman.

**Intuition.** Consider substituting  $\text{pk} = A = g^a$  and  $\text{sk} = a$ .

**Note.** Recall that given  $K$ , we can compute  $K^{-1}$  efficiently.

We now show that the ElGamal scheme is really what we want.

**Claim.** The ElGamal scheme is correct.

**Proof.** Since for all  $m \in G$ ,  $(\text{pk}, \text{sk}) = (g^a, a)$ , hence

$$\text{Enc}(\text{pk}, A) = (B, c) = (g^b, m \cdot (g^a)^b),$$

so

$$\text{Dec}(B, c) = c \cdot (B^a)^{-1} = m \cdot g^{ab} \cdot (g^{ab})^{-1} = m.$$

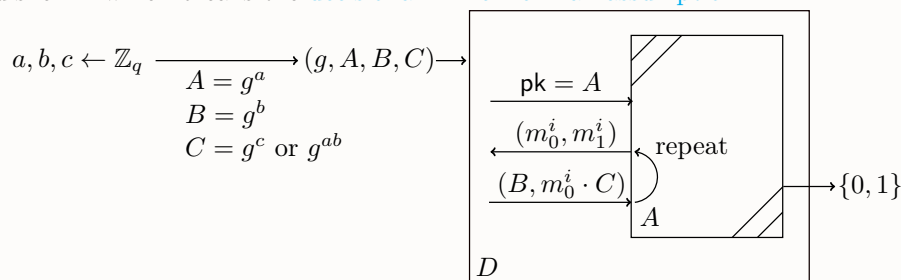
⊛

As for CPA security, we again consider the decisional Diffie-Hellman assumption over  $G$ , i.e.,

$(g, g^a, g^b, g^{ab}) \in G^4$  for  $a, b \leftarrow \mathbb{Z}_q$  is indistinguishable from  $(g, g^a, g^b, g^c) \in G^4$  for  $a, b, c \leftarrow \mathbb{Z}_q$ .

**Theorem 5.4.3.** Assuming [decisional Diffie-Hellman assumption](#) for  $G$ , then the ElGamal scheme  $\Pi$  is [CPA secure](#).

**Proof.** Let  $A$  be any feasible probabilistic polynomial time attacker against  $\Pi$ , we use  $A$  to build a distinguisher  $D$  which breaks the [decisional Diffie-Hellman assumption](#).



There are two worlds:

- “real” world: if  $(g, A, B, C)$  is a [Diffie-Hellman](#) tuple,  $D$  perfectly simulates the left [CPA](#) world since  $C = g^{ab}$ , so the ciphertext is  $\text{Enc}_{pk}(m_0^i) = (g^b, m_0^i \cdot pk^b) = (g^b, m_0^i \cdot g^{ab}) = (g^b, m_0^i \cdot C)$ ;
- “ideal” world: if  $(g, A, B, C)$  is random, then  $D$  simulates a “hybrid” [CPA](#) world where the ciphertext is two independent random group elements (regardless of the message).

Symmetrically, we can construct  $D'$  against [decisional Diffie-Hellman assumption](#) that replies to  $A$  with  $(B, m_1^i \cdot C)$ . Then, we have

$$\text{Adv}^{\text{CPA}}(A) \leq \text{Adv}^{\text{DH}}(D) + \text{Adv}^{\text{DH}}(D') = \text{negl}(n) + \text{negl}(n) = \text{negl}(n),$$

which is a contradiction, hence we're done. ■

## Lecture 23: RSA Function and RSA Encryption

### 5.5 RSA Cryptosystem

3 Apr. 10:30

The [hardness assumption](#) of [Diffie-Hellman](#) relies on the hardness of the discrete logarithm problem (finding an unknown exponent, i.e., given  $g^a$ ) in a [group](#) of known order. We now ask the following.

**Problem.** Is there other hardness assumption we can use?

**Answer.** Yes! As we will see, we can utilize the [RSA function](#), which relies on the hardness of “factoring”,<sup>a</sup> and finding the base (unknown but known exponent) in a [group](#) of unknown order.  $\circledast$

<sup>a</sup>I.e., the [prime factorization](#).

By utilizing this new hardness, beyond [public key encryption](#) (as provided by [ElGamal](#)), we can also sign messages, i.e., we get a [cryptosystem](#). We first focus on encrypting.

#### 5.5.1 RSA Function

Let  $N = pq$  be the product of two large distinct [primes](#)  $p$  and  $q$ . Then,  $\mathbb{Z}_N^* = \{a \in \mathbb{Z}_N \mid \gcd(a, N) = 1\}$ , i.e., we start with  $\mathbb{Z}_N$  and throw out all multiples of  $p$  and  $q$ .

**Remark.** This means  $\varphi(N) = (p-1)(q-1)$ .

**As previously seen.** In any [group](#)  $G$ ,  $a^{|G|} \equiv 1 \pmod{N}$  for all  $a \in G$ . Furthermore, [Euler's theorem](#) states that for any  $a \in \mathbb{Z}_N^*$ , we have  $a^{\varphi(N)} \equiv 1 \pmod{N}$ .

**Intuition.** If any element is raised to a multiple of  $\varphi(N) \pmod{N}$ , we get back the original element!



This motivates the following: given  $N$ ,  $e$ , and  $d$ , we defined the so-called **RSA function** and its inverse.

**Definition 5.5.1 (RSA function).** The *RSA function*  $\text{RSA}_{N,e}(x)$  is defined as

$$\text{RSA}_{N,e}(x) = x^e \bmod N.$$

A particularly useful property of RSA is that by defining  $d \equiv e^{-1} \pmod{\varphi(N)}$ , the **RSA function** is a bijection from  $\mathbb{Z}_N^*$  to itself with the inverse function being

$$\text{RSA}_{N,d}(y) = y^d \bmod N.$$

**Proposition 5.5.1.** RSA is a bijection with  $\text{RSA}_{N,d}(y)$  for  $d \equiv e^{-1} \pmod{\varphi(N)}$  being the inverse of  $\text{RSA}_{N,e}(x)$ .

**Proof.** We need to show that  $\text{RSA}_{N,d}(\text{RSA}_{N,e}(x)) = x$  for all  $x \in \mathbb{Z}_N^*$ . We have

$$\text{RSA}_{N,d}(\text{RSA}_{N,e}(x)) = (x^e)^d \bmod N.$$

Using the property that  $e \cdot d \equiv 1 \pmod{\varphi(N)}$ , we know that there exists an integer  $k$  such that  $e \cdot d = 1 + k \cdot \varphi(N)$ . Therefore,

$$(x^e)^d = x^{ed} = x^{1+k \cdot \varphi(N)} = x \cdot (x^{\varphi(N)})^k \bmod N.$$

By **Euler's theorem**, we have  $x^{\varphi(N)} \equiv 1 \pmod{N}$  for all  $x \in \mathbb{Z}_N^*$ . Thus,

$$x \cdot (x^{\varphi(N)})^k \equiv x \cdot 1^k \equiv x \bmod N,$$

hence  $\text{RSA}_{N,d}(\text{RSA}_{N,e}(x)) = x$  for all  $x \in \mathbb{Z}_N^*$ . This also proves that the RSA is a bijection, and we can efficiently evaluate and invert it using the trapdoor information  $d$ . ■

## 5.5.2 RSA Construction

From **Proposition 5.5.1**, it's possible to construct a **public key scheme** such that given a message  $m$ , we encrypt it as  $c := \text{RSA}_{N,e}(m) = m^e \pmod{N}$ ; to decrypt, we use  $\text{RSA}_{N,d}(c) = c^d \pmod{N}$ , which is proved to be  $m$ , i.e., we consider **Figure 5.1**.

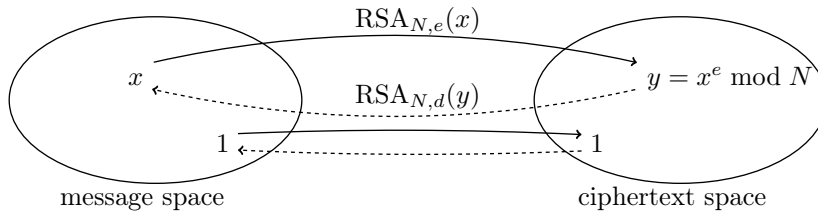


Figure 5.1: RSA is a bijection

Formally, consider the following key generation mechanism for RSA.

**Definition 5.5.2 (RSA key).** The *RSA key*  $(\text{pk}, \text{sk})$  is generated as follows:

- choose random, independent, large **primes**  $p$  and  $q$  having bit lengths approximately related to  $n$ , and compute  $N = pq$ ;
- choose  $e \in \mathbb{Z}_{\varphi(N)}^*$  such that  $\gcd(e, \varphi(N)) = 1$ , and compute  $d \equiv e^{-1} \pmod{\varphi(N)}$ ;
- output  $(\text{pk}, \text{sk})$  where the public key  $\text{pk} = (N, e)$  and the private key  $\text{sk} = (N, d)$ .

**As previously seen** (Compute  $d$ ). Given a random  $e$ , to compute  $d$ , we note that since  $\gcd(e, \varphi(N)) = 1$ , by running [extended Euclid algorithm](#), we can find  $A, B$  such that  $1 = Ae + B\varphi(N) \bmod \varphi(N)$ . Then, we have  $Ae = 1 - B\varphi(N) = 1 \bmod \varphi(N)$ , i.e., we have  $d = A = e^{-1} \bmod \varphi(N)$ .

**Example.** Common choices for  $e$  are  $e = 3$  or  $e = 2^{16} + 1$ .

**Proof.** They are both [primes](#), hence it's always the case that  $\gcd(e, \varphi(N)) = 1$ . It's particular nice since  $\langle e \rangle$  only have two bits that are 1, so computing modular exponentiation is fast.  $\otimes$

Then what we just saw is the following [public key scheme](#).

**Definition 5.5.3** (Textbook RSA cryptosystem). The *RSA cryptosystem* given in [KL20] is a [public key scheme](#)  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  where

- $\text{Gen}(1^n)$ : output  $(\text{pk}, \text{sk}) = ((N, e), (N, d))$  from [RSA key generation](#);
- $\text{Enc}(\text{pk}, m)$  for  $m \in \mathbb{Z}_N^*$ : output  $c = \text{RSA}_{N,e}(m) = m^e \bmod N$ ;
- $\text{Dec}(\text{sk}, c)$  for  $c \in \mathbb{Z}_N^*$ : output  $m = \text{RSA}_{N,d}(c) = c^d \bmod N$ .

The [text RSA cryptosystem](#) is certainly [correct](#) due to [Proposition 5.5.1](#); however, it is not [secure](#): since the encryption is deterministic, hence [Theorem 5.4.2](#) applies! To fix it,<sup>4</sup> we need to understand what hardness assumption does RSA rely on exactly.

### 5.5.3 Security of RSA

From the previous [example](#), we see that what we really want is that for any probabilistic polynomial time adversary  $A$ , the probability that  $A$  can find the pre-image  $x = m$  of  $y = c$  under  $\text{RSA}_{N,e}$  for some random  $y \in \mathbb{Z}_N^*$  is [negligible](#) in  $n$ . Formally, we have the following.

**Conjecture 5.5.1** (RSA hardness). The *RSA hardness* assume that for all probabilistic polynomial time  $A$ ,

$$\Pr_{\substack{(\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^n) \\ y \leftarrow \mathbb{Z}_N^*}} (A(1^n, \text{pk}, y) \text{ outputs } x = \text{RSA}_{N,e}^{-1}(y)) = \text{negl}(n)$$

for  $\text{pk} = (N, e)$ .

**Intuition.** It's hard to find the pre-image  $x = y^d \bmod N$  given  $(N, e)$  and  $y (= x^e \bmod N)$ .

**Remark.** The hardness assumption is believed to be true.

**Note.** The [RSA hardness assumption](#) is specific:

- it holds only when  $y$  is sampled randomly: for particular values of  $y$ , it might be easy to find the pre-image of  $y$ ;<sup>a</sup>
- the pre-image of a random  $y$  cannot be “completely” recovered: perhaps we can recover partial information about it.

<sup>a</sup>The inverse of some ciphertexts, e.g., 1, are easy to compute since  $1^e = 1$ ; see [Figure 5.1](#).

The problem of finding the pre-image of RSA is related to the factoring, and indeed, we can compare the hardness of them.

**Claim.**  $\text{RSA} \leq \text{factoring}$ , i.e., if there's an efficient algorithm for [prime factorization](#), then there is one for solving [RSA](#).

<sup>4</sup>I.e., to make  $\text{Enc}$  not deterministic.

**Proof.** Given  $\text{pk} = (N, e)$ ,  $y \in \mathbb{Z}_N^*$ , by factoring  $N = p \cdot q$ , we obtain  $\varphi(N) = (p-1)(q-1)$ . Then, it's easy to compute  $d = e^{-1} \pmod{\varphi(N)}$  hence  $x = y^d \pmod{N}$ .  $\circledast$

**Claim.** Computing  $\varphi(N) \Leftrightarrow$  factoring.

**Proof.** If we know  $\varphi(N) = p \cdot q - p - q + 1 = (N+1) - (p+q)$ , then we can solve for  $p+q = (N+1) - \varphi(N)$  and  $p \cdot q = N$ . This is enough to have a factoring of  $N$ .  $\circledast$

**Claim.** Factoring  $\Leftrightarrow$  finding  $d$  from  $(N, e)$ .

**Proof.** We know  $e \cdot d - 1 = k \cdot \varphi(N)$  for some  $k$ . Through fancy math (see book!), it is multiple of  $\varphi(N)$  to recover enough information to have a factoring of  $N$ .  $\circledast$

**Remark.** It's still an open problem that whether  $\text{RSA} \equiv \text{factoring}$ .<sup>a</sup>

<sup>a</sup>We suspect that factoring is easier than taking roots.

### 5.5.4 An Even Better RSA Cryptosystem

Since the [textbook construction](#) of RSA is not [CPA secure](#) because it has deterministic encryption, so an adversary can easily tell when the same message is sent twice.

**Note.** It is also not necessarily [EAV secure](#).

**Proof.** Since messages don't usually follow a uniform random distribution, so the ciphertexts  $y$ 's won't be uniformly random either. This doesn't fit the [RSA hardness assumption](#) directly, so we cannot say it is [EAV secure](#).  $\circledast$

To construct a [secure public key scheme](#) using RSA, we modify the above [textbook construction](#) and incorporate a [cryptographic hash function](#)  $H$ , and fix it by applying  $\text{RSA}_{N,e}$  on a random  $x \leftarrow \mathbb{Z}_N^*$ .<sup>5</sup>

**Intuition.** We don't have a guarantee that the encryption is random, but we do know it is hard to compute so if we [hash](#)  $x$  and pad that to the message, only someone with the private key will be able to recover the message.

So the ciphertext would be  $c = (\text{RSA}_{N,e}(x), H(x) \oplus m)$ .

## Lecture 24: Digital and RSA Signatures

However, in this case, we require something stronger than [collision resistance](#) for  $H$ .

5 Apr. 10:30

**Intuition.** A good [hash function](#) “practically behaves” like a uniform random function.

**Notation** (Random oracle). Such an  $H$  is also known as a *random oracle*.

**Example.** SHA-3 is quite “random-like”.

**Remark.** If  $x$  is not completely known, then  $H(x)$  is completely random.

Formally, consider the following.

<sup>5</sup>To take advantage of the assumption.

**Definition 5.5.4** (Randomized RSA cryptosystem). Given a random oracle Hash  $H: \{0, 1\}^n \rightarrow \mathbb{Z}_N^*$ , the randomized RSA cryptosystem is a public key scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  where

- $\text{Gen}(1^n)$ : output  $(\text{pk}, \text{sk}) = ((N, e), (N, d))$  from RSA key generation;
- $\text{Enc}(\text{pk}, m)$  for  $m \in \mathbb{Z}_n^*$ : output  $(y, z) = (\text{RSA}_{N,e}(x), H(x) \oplus m)$  for  $x \leftarrow \mathbb{Z}_n^*$ ;
- $\text{Dec}(\text{sk}, c)$  for  $c = (y, z)$ : output  $H(\text{RSA}_{N,d}(y)) \oplus z$ .

The correctness follows directly from Proposition 5.5.1 and computation, since

$$H(\text{RSA}_{N,d}(y)) \oplus z = H(\text{RSA}_{N,d}(\text{RSA}_{N,e}(x))) \oplus (H(x) \oplus m) = H(x) \oplus H(x) \oplus m = m.$$

We claim that this construction is CPA secure.

**Theorem 5.5.1.** Randomized RSA cryptosystem is CPA secure.

**Proof Idea.** As noted,  $H$  behaves like a random function/oracle. Thus, without fully knowing  $x$ ,  $H(x)$  looks completely random, meaning  $H(x) \oplus m$  looks completely random. By the RSA hardness assumption, an adversary cannot fully know  $x$ . So, an adversary cannot distinguish the encryption of a message from randomness. This ensures CPA secure. ■

## Chapter 6

# Asymmetric Public Key Message Authentication

We have studied [MAC](#), where a sender and receiver share *symmetric keys*. Only senders who know the secret key can tag messages, and only receivers who know the secret key can verify them.

Now, we want to generalize this idea, i.e., we want to extend it to the setup of *asymmetric*, i.e., now a sender and receiver share asymmetric keys. Only senders who know the private key can sign messages, and only receivers who know the corresponding public key can verify them. This is called *signatures*.

### 6.1 Digital Signatures

Consider the following.

**Definition 6.1.1** (Signature scheme). A *signature scheme* is a tuple  $\Pi = (\text{Gen}, \text{Sign}, \text{Ver})$  where

- $\text{Gen}(1^n)$ : outputs  $(\text{vk}, \text{sk})$ ;
- $\text{Sign}(\text{sk}, m)$  for  $\text{sk} \leftarrow \text{Gen}(1^n)$  and  $m \in \mathcal{M}$ : outputs  $\sigma$ ;
- $\text{Ver}(\text{vk}, m, \sigma)$  for  $\text{vk} \leftarrow \text{Gen}, m \in \mathcal{M}, \sigma$ : outputs Accept or Reject.

**Notation** (Digital signature). We sometimes call [signature scheme](#) the *digital signature*.

To decrypt [Definition 6.1.1](#) a bit, we see that

- $\text{Gen}$  is given the security parameter and outputs a key-pair, where  $\text{vk}$ , sometimes denoted  $\text{pk}$ , is the *verification key* or *public key*; and  $\text{sk}$  is the *secret key* or *signing key*.
- $\text{Sign}$  is given a verification key from  $\text{Gen}$  as well as a message  $m$ . It outputs a *signature*  $\sigma$ .
- $\text{Ver}$  is given a public key from  $\text{Gen}$ , a message  $m$ , and a signature  $\sigma$ . It accepts or rejects.

Naturally, we want the following.

**Definition 6.1.2** (Correctness). A [signature scheme](#)  $\Pi$  is *correct* if for all  $(\text{vk}, \text{sk}) \leftarrow \text{Gen}(1^n)$  and for all  $m \in \mathcal{M}$ ,

$$\text{Ver}(\text{vk}, m, \text{Sign}(\text{sk}, m)) = \text{Accept}.$$

There's something fundamentally different from [MACs](#) we have seen.

**As previously seen.** With [MACs](#), verification came for free (i.e., [canonical verification](#)) since the sender and receiver shared the secret key, the receiver could verify messages by running the tag algorithm themselves.

Compared to [MACs](#), [digital signatures](#) have no analogue of [canonical verification](#) because the receiver doesn't know the signing key, hence can't run  $\text{Sign}$ .

### 6.1.1 Chosen Message Attack

Nevertheless, apart from the above difference, the security definitions for [digital signatures](#) and [MACs](#) are very similar: we first define the analog to [chosen message attack](#) for [MAC](#).

**Definition 6.1.3** (Chosen message attack). The *chosen message attack* game given an adversary  $A$ , a parameter  $n$ , and a [signature scheme](#)  $\Pi$  is conducted as follows.

1. Let  $(vk, sk) \leftarrow \text{Gen}(1^n)$ .
2.  $A(1^n, vk)$  receives polynomially many signatures for messages by querying the signing oracle  $\text{Sign}(sk, \cdot)$ .
3.  $A$  eventually outputs  $(m^*, \sigma^*)$ .

**Definition 6.1.4** (Weak forgery). In the [CMA](#),  $A$  *forges* if  $\text{Ver}(vk, m^*, \sigma^*)$  accepts and  $m^*$  is not a query from  $A$  to the signing oracle.

**Definition 6.1.5** (Advantage). Given an adversary  $A$  in an [CMA](#), the *advantage*  $\text{Adv}_{\Pi}^{\text{CMA}}(A)$  in generating a fake message/signature pair is given by

$$\text{Adv}_{\Pi}^{\text{CMA}}(A) := \Pr_{(vk, sk) \leftarrow \text{Gen}(1^n)} (A^{\text{Sign}(sk, \cdot)}(1^n, vk) \text{ forges}).$$

In addition, we also consider the following as what we have done in [MACs](#).

**Definition 6.1.6** (Strong forgery). In the [CMA](#),  $A$  *forges* if  $\text{Ver}(vk, m^*, \sigma^*)$  accepts and  $(m^*, \sigma^*)$  is not a query-response pair from  $A$  to the signing oracle.

**Note.**  $\text{Adv}_{\Pi}^{\text{SCMA}}(A)$  is defined in the natural way as in [Definition 6.1.5](#).

### 6.1.2 Chosen Message Attack Security

Following the same vein as [MACs](#), we define the following.

**Definition 6.1.7** (Unforgeable). A [signature scheme](#)  $\Pi$  is (*existentially*) *unforgeable* under [CMA](#) if for every probabilistic polynomial time adversary  $A$ , the [advantage](#) is [negligible](#).

**Notation** (UFCMA). An [unforgeable signature scheme](#) under [CMA](#) is abbreviated as *UFCMA*.

Similarly, we define the same notion for [strong forgery](#).

**Notation** (Strongly UFCMA). A [signature scheme](#)  $\Pi$  is *strongly unforgeable* under [CMA](#), or *strongly UFCMA*, if for every probabilistic polynomial time adversary  $A$ , the  $\text{Adv}_{\Pi}^{\text{SCMA}}(A) = \text{negl}(n)$ .

When we discussed [MACs](#), this caused some confusion: here again, we note the following.

**Claim.** [strongly UFCMA](#) is a strong security notion (i.e., more secure) than [UFCMA](#).

**Proof.** We observe that

- to break [strong UFCMA](#), an attacker must forge  $(m^*, \sigma^*)$  such that either  $m^*$  or  $\sigma^*$  is new;
- to break [UFCMA](#), an attacker must forge  $(m^*, \sigma^*)$  such that  $m^*$  is new.

If you can do the latter, you can do the former. Thus,  $\neg \text{UFCMA} \Rightarrow \neg \text{strong UFCMA}$ . By taking the contrapositive, [strong UFCMA](#)  $\Rightarrow$  [UFCMA](#). \*

**Remark.** [UFCMA](#) does not rule out, e.g., replay attacks (although time-stamping messages does).

## 6.2 RSA Signatures

In RSA encryption,  $e$  allows the public to encrypt messages and  $d$  allows one party to decrypt them. RSA's encryption and decryption functions are very similar, so with minimal changes, we can construct a scheme where  $e$  allows the public to decrypt messages and  $d$  allows one party to encrypt them. This leads to a [digital signature scheme](#).

### 6.2.1 RSA Signature Scheme Construction

**Definition 6.2.1** (Textbook RSA signature). The *RSA signature* given in [KL20] is a [signature scheme](#)  $\Pi = (\text{Gen}, \text{Sign}, \text{Ver})$  where

- $\text{Gen}(1^n)$ : output  $(\text{vk}, \text{sk}) = ((N, e), (N, d))$  from [RSA key generation](#);
- $\text{Sign}(\text{sk}, m)$  for  $m \in \mathbb{Z}_N^*$ : output  $\sigma = \text{RSA}_{N,d}(m)$ .
- $\text{Ver}(\text{vk}, m, \sigma)$  for  $m \in \mathbb{Z}_N^*$ : output **Accept** if  $\text{RSA}_{N,e}(\sigma) = m$ , otherwise output **Reject**.

The [textbook RSA signature](#) is [correct](#) from the [Proposition 5.5.1](#). Moreover, given  $m$ ,  $\text{vk}$ , and not  $d$ , it is hard to find  $\sigma = \text{Sign}(\text{sk}, m)$  by the [RSA hardness assumption](#). Thus, a forger cannot attack the [textbook RSA signature](#) by choosing a message and finding its signature. However, we note the following.

**Remark.** The [RSA hardness assumption](#) does not rule out choosing a signature and finding a message that produces it. In fact, given  $\sigma$ ,  $\text{vk}$ , and not  $d$ , it is easy to find  $m$  such that  $\sigma = \text{Sign}(\text{sk}, m)$ .

This leads to the following.

**Theorem 6.2.1.** The [textbook RSA signature](#) is not [UFCMA](#).

**Proof.** Consider a probabilistic polynomial time adversary  $A$  in the [CMA game](#) against the [textbook RSA signature](#), where  $A$  first choose any  $\sigma^* \in \mathbb{Z}_N^*$  and return  $(m^*, \sigma^*) = (\text{RSA}_{N,e}(\sigma^*), \sigma^*)$ . We see that  $\text{Ver}(\text{vk}, m^*, \sigma^*)$  always accepts because  $\text{RSA}_{N,e}(\sigma^*) = m^*$ . Thus,  $A$  [forges](#) with probability 1 while making no queries. ■

**Another Proof.** Consider a probabilistic polynomial time adversary  $A$  in the [CMA game](#) against the [textbook RSA signature](#), where for arbitrary  $m$  and  $m'$ ,  $A$  queries to obtain  $(m, \sigma)$  and  $(m', \sigma')$ . Then, it outputs  $(m \cdot m', \sigma \cdot \sigma')$ .<sup>a</sup> We see that  $\text{Ver}(\text{vk}, m \cdot m', \sigma \cdot \sigma')$  always accepts because  $\text{RSA}_{N,e}(\sigma \cdot \sigma') \equiv (\sigma \cdot \sigma')^e \equiv \sigma^e \cdot \sigma'^e \equiv m \cdot m' \pmod{N}$ . So,  $A$  always [forges](#) if  $mm'$  is neither  $m$  nor  $m'$ . ■

<sup>a</sup>Remember that  $m \in \mathbb{Z}_N^*$  in general, hence it makes to talk about multiplication.

### 6.2.2 An Even Better RSA Signature Scheme

One way to fix the [textbook RSA signature](#) is the following.

**Definition 6.2.2** (Hash-and-Sign RSA signature). Given a [random oracle Hash](#)  $H: \{0, 1\}^n \rightarrow \mathbb{Z}_N^*$ , the *Hash-and-Sign RSA signature* is a [signature scheme](#)  $\Pi = (\text{Gen}, \text{Sign}, \text{Ver})$  where

- $\text{Gen}(1^n)$ : output  $(\text{vk}, \text{sk}) = ((N, e), (N, d))$  from [RSA key generation](#);
- $\text{Sign}(\text{sk}, m)$  for  $m \in \{0, 1\}^n$ : output  $\sigma = \text{RSA}_{N,d}(H(m))$ ;
- $\text{Ver}(\text{vk}, m, \sigma)$  for  $m \in \{0, 1\}^n$ : output **Accept** if  $\text{RSA}_{N,e}(\sigma) = H(m)$ , otherwise output **Reject**.

Again, the [correct](#) from the [Proposition 5.5.1](#). We claim that this construction is [UFCMA](#).

**Theorem 6.2.2.** The Hash-and-Sign RSA signature is UFMA.

**Proof Idea.** To forge, an adversary can either

- (i) choose a message and try to find a matching signature, or
- (ii) choose a signature and try to find a matching message.

By the RSA hardness assumption, (i) is infeasible because the adversary doesn't know  $d$ ; (ii) is also infeasible because it amounts to finding  $m \in H^{-1}(\{y\})$  for some  $y$  of the adversary's choice while  $H$  is "secure". ■

## Lecture 25: Schnorr's Identification and Fiat-Shamir

### 6.2.3 Comparison between RSA Function and Discrete Logarithm

10 Apr. 10:30

We showed that a simple application of the RSA function can create a digital signature.

**Intuition.** RSA function contains a "trapdoor" for calculating the inverse, making it easier for us to perform the verification step when ensuring that message was in fact signed by the authentic user.

**Remark.** Contrarily, it is not so easy to repurpose the Diffie-Hellman key exchange for this because it does not contain such a "trapdoor", where the hardness comes from the discrete logarithm problem.

But, as we will see, it can really be thought of as a one-way function, i.e., it's possible to construct digital signature based on the hardness of discrete logarithm.

### 6.3 Identification Schemes

An *identification scheme* is a digital signature utilizes the hardness of discrete logarithm problem, where it allows one to prove that you have the secret key without revealing it. In particular, we consider the Schnorr's identification, which (after some modifications) is a "zero-knowledge" protocols that utilizes the hardness of a discrete logarithm.

**Note.** The modifications are needed, as we will see, since Schnorr's identification has a flaw that it can reveal  $x$ .

Specifically, the enhancement of Schnorr's identification is done by using the Fiat-Shamir transform that utilizes hashing to increase randomness and conceal the secret key. Let's first see the Schnorr's identification.

#### 6.3.1 Schnorr's Identification

Consider the following discrete logarithm problem setup.

**Problem 6.3.1 (Secret discrete logarithm problem).** Let  $G$  be a large cyclic group of known prime order  $q$ ,<sup>a</sup> generated by  $g$ , i.e.,  $|G| = q$  and  $G = \{g^0, g^1, g^2, \dots, g^{q-1}\}$ . Let the secret key be a random number  $x \in \mathbb{Z}_q$ , and the public key be  $y = g^x \in G$ . The *secret discrete logarithm problem* asks for a prover  $P(x)$  who knows the private key  $x$  to prove that it has the knowledge of  $x$  without revealing it to a verifier  $V(y)$  who only has access to the public key  $y$ .

<sup>a</sup>For example,  $G = \mathbb{Z}_q^*$ .

**Intuition.** Everyone knows  $y = g^x$ , but if  $P$  want to "prove" to  $V$  that  $P$  has the secret key  $x$ ,  $P$  need to solve the discrete logarithm problem. With the fact that  $P$  doesn't want to reveal  $x$ , we have the secret version of the discrete logarithm problem as defined above.



Claus-Peter Schnorr describes a four-step, *interactive protocol* between a prover (has  $x$ ) and a verifier (has  $y$ ) as described in the [secret discrete logarithm setup](#).

[cite](#)

**Definition 6.3.1 (Schnorr's identification).** The *Schnorr's identification* is a four-step interactive protocol between a prover  $P(x)$  and a verifier  $V(y)$  under the [secret discrete logarithm problem](#) setup, which works as follows.

1. The prover generates  $k \leftarrow \mathbb{Z}_q$  and sends  $c = g^k \in G$  to the verifier.<sup>a</sup>
2. The verifier generates  $r \leftarrow \mathbb{Z}_q$  and sends it to the prover.
3. The prover  $P(x)$  calculates  $s = k + r \cdot x \pmod{q} \in \mathbb{Z}_q$  and sends it to the verifier.
4. The verifier  $V(y)$  verifies whether  $g^s = g^{k+r \cdot x \pmod{q}} = c \cdot y^r$ , and accepts if this is correct.

<sup>a</sup>This commits the prover to some public key  $g^k$ .

### 6.3.2 Issues with Schnorr's Identification

[Schnorr's identification](#) is correct since if  $P$ ,  $V$  run the protocol honestly, then  $V$  will accept; and if  $V$  accepts, then that means that  $P$  *knows*  $x$ . fAs for soundness,<sup>1</sup> we want to make sure that if  $V$  accepts with high probability, then  $P$  actually “knows”  $x$ .

**Example (Thought experiment).** Consider two challenges  $r_1 \neq r_2$  sent by  $V$  to  $P$  for which  $P$  manages to make  $V$  accept. Then, say  $P$ 's responses are  $s_1$  and  $s_2$ , respectively, at the third step. We know that there are some  $r_1$  and  $r_2$ , respectively, such that

$$g^{s_1} = c \cdot y^{r_1}, \quad g^{s_2} = c \cdot y^{r_2},$$

$$\text{i.e., } g^{s_1 - s_2} = y^{r_1 - r_2} = g^{x(r_1 - r_2)}.$$

From the above example, we see that since  $g$  is a [generator](#),  $s_1 - s_2 = x(r_1 - r_2) \pmod{q}$ , i.e.,

$$x = (s_1 - s_2) \cdot (r_1 - r_2)^{-1} \pmod{q},$$

implying that we can extract  $x$  via the [extended Euclid algorithm](#) by finding the inverse of  $(r_1 - r_2)$ .

**Remark.** [Schnorr's identification](#) may unknowingly reveal information about  $x$ .

This is what we wanted to keep secret according to the [problem setup](#), hence [Schnorr's identification](#) doesn't really work.

### 6.3.3 Fiat-Shamir Transform

To make [Schnorr's identification](#) truly “zero knowledge”, we want to create an *efficient* simulator that samples the distribution of the exchanged information *without knowing*  $x$ .

**Intuition.** The idea is to change the ordering of our random variable sampling to  $(c, r, s)$ .

The only difference is that we first sample  $r$ , then  $s$ , then  $c$ . For now, we first choose  $r \leftarrow \mathbb{Z}_q$ , and  $s \leftarrow \mathbb{Z}_q$ , and finally set  $c = g^s \cdot y^{-r} \in G$ . This has exactly the same distribution as in [Schnorr's identification](#), but knows nothing about  $x$  that  $V$  doesn't already know. So,  $V$  learns nothing new about  $x$  other than the fact that  $P$  knows it.

**Intuition.** Since  $x$  is in fact unused, we don't give any knowledge about  $x$ .

**Remark.** Simply by changing the order of the verification scheme, we don't give knowledge of  $x$ .

<sup>1</sup>I.e., you can't prove the wrong thing.

However, there are two issues: we didn't sign messages, and we had an interactive protocol. To get around this we do the normal [hash](#) function to reuse out [OTP](#) principles. This can be done by *Fiat-Shamir transform*.

cite

Fiat-Shamir transform is a technique for taking an interactive proof of knowledge and creating a [digital signature](#) based on it. This way, some fact (for example, knowledge of the secret key) can be publicly proven without revealing underlying information.

**Definition 6.3.2** (Fiat-Shamir transformed Schnorr's identification). Given a [random oracle Hash](#)  $H: \{0,1\}^n \rightarrow \mathbb{Z}_N^*$ , the *Fiat-Shamir transformed Schnorr's identification* is a [signature scheme](#)  $\Pi = (\text{Gen}, \text{Sign}, \text{Ver})$  where

- $\text{Gen}(1^n)$ : generate  $x \leftarrow \mathbb{Z}_q$ , and output  $(\text{vk}, \text{sk}) = (g^x, x) = (y, x)$  where  $y \in G$ ;
- $\text{Sign}(\text{sk}, m)$  for  $m \in \{0,1\}^*$ : generate  $k \leftarrow \mathbb{Z}_q$ , compute  $c = g^k \in G$ , and output  $\sigma = (r, s) = (H(m, c), k + rx \bmod q)$ ;
- $\text{Ver}(\text{vk}, m, \sigma)$  for  $m \in \{0,1\}^*$ : output **Accept** if  $H(m, c) = r$  for  $c = g^s \cdot y^{-r}$ , otherwise output **Reject**.

**Remark.** We replaced the interactive step by invoking  $H$ .

**Theorem 6.3.1.** Assuming [Conjecture 5.3.1](#) on  $G$  and  $H$  is a [random oracle](#), the [Fiat-Shamir Schnorr identification](#) is [unforgeable](#).

**Proof idea.** Because  $H$  is a [random oracle](#), the values  $r = H(m, c)$  that a forger must deal with are like truly random challenges in [Schnorr's identification](#) protocol. A forger won't be able to answer such a challenge unless  $H$  gets extremely luck in receiving the one challenge it knows how to handle, and it's able to compute  $x = \log_g y$  for the legit signer's public key  $y$  (a uniform random for the legit signer's element of  $G$ ). ■

# Chapter 7

## Post-Quantum and Lattice-Based Cryptography

### Lecture 26: Lattice-Based Cryptography

Finally, we give a *very* brief overview on the recent advances in cryptography, i.e., the post-quantum cryptography. Due to Shor, some hardness assumptions we rely on is already broken using quantum algorithms. Hence, people study the “post”-quantum cryptography, which aims to find new hardness assumptions against even quantum computers that we can rely on to build a secure cryptosystem.

12 Apr. 10:30

cite

A particular important subject is the [lattice](#) theory, which relies on the discrete nature of integers.

### 7.1 Post-Quantum Cryptography

#### 7.1.1 Shor’s Algorithm

Peter Shor had shocked the world with a quantum algorithm that can factorize integers in polynomial time. The idea was to use the quantum “weirdness”, also known as “complex probabilities”. This means that [RSA](#) could be broken.

cite

**Remark.** Quantum search is also weird. You can search in an unstructured array of size  $N$  using  $O(\sqrt{N})$  operations. This is Grover’s algorithm.

cite

In cryptography, quantum computers can brute force for a key of length  $N$  in time  $2^{\frac{N}{2}}$  rather than  $2^N$ .

**Remark.** A remedy to this is to double the key length.

Shor’s algorithm also computes discrete logarithm in polynomial time. It breaks [Diffie-Hellman](#), [ElGamal](#), etc.

cite

#### 7.1.2 Post-Quantum Cryptography

Post-quantum cryptography can’t rely on the hardness of factoring or discrete logarithm. An older proposal was to rely on hardness of things like subset sum or [hash functions](#). Beyond this, more successful proposals rely on coding theory and lattice theory.

cite

cite

### 7.2 Lattice-Based Cryptography

Lattice-based cryptography is to build cryptography based on hardness of problems about [lattices](#).

#### 7.2.1 Lattice

Consider the following.

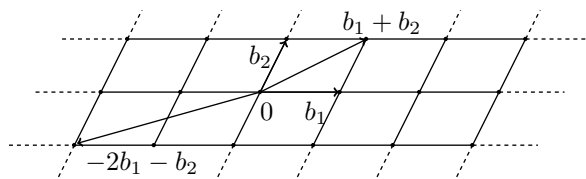
**Definition 7.2.1 (Lattice).** A lattice  $\mathcal{L}(B)$  is a  $\mathbb{Z}$ -vector space spanned by  $B$ .

**Note.** We will mainly consider  $B \subseteq \mathbb{R}^n$ , i.e.,  $\mathcal{L}(B)$  is a  $\mathbb{Z}$ -vector space embedded in  $\mathbb{R}^n$ .

Basically, consider a set of basis vectors  $B = \{b_1, b_2, \dots, b_n \mid b_i \in \mathbb{R}^n\}$ , then

$$\mathcal{L}(B) = \{z_1 b_1 + \dots + z_n b_n : z_i \in \mathbb{Z}\} \subseteq \mathbb{R}^n.$$

**Intuition.** It's a periodic, infinite grid in an  $n$ -dimensional space, generated by  $n$  basis vectors in  $\mathbb{R}^n$ .



**Remark.** In matrix form,  $\mathcal{L}(B) = \{B \cdot Z : Z \subseteq \mathbb{Z}^n\}$ .

## 7.2.2 Hardness on Lattices

One conjectured hard [lattice](#) problem is the [shortest vector problem](#).

**Problem 7.2.1 (Shortest vector problem).** Given  $B$ , the *shortest vector problem* asks for finding the shortest (or a “very short”) nonzero vector  $v \in \mathcal{L}(B)$ .

Another conjectured hard [lattice](#) problem is decoding, also known as [closest vector problem](#).

**Problem 7.2.2 (Closest vector problem).** Given  $B$  and a target point  $t \in \mathbb{R}^n$ , the *closest vector problem* asks for finding the closest vector in  $\mathcal{L}(B)$  to  $t$ .

**Problem 7.2.3 (Learning with errors).** Given  $n$  and a [prime](#)  $q \approx n^2$ , pick a secret  $s \leftarrow \mathbb{Z}_q^n$ . An instance of *learning with errors* problem is where a random matrix  $A \leftarrow \mathbb{Z}_q^{m \times n}$  is chosen and  $b = A \cdot s + e$  is calculated for  $e$  being a short error vector (noise).<sup>a</sup> The task is to find  $s$  given  $n$ ,  $q$ ,  $A$ , and  $b$ .

<sup>a</sup>All operations are over  $\mathbb{Z}_q$ .

**Remark.** The problem of [Learning with errors](#) is conjectured to be hard.

**Example.** One example of  $e$  is that each entry is randomly chosen from  $[-10, 10]$ .

[Learning with errors](#) is closely related to [closest vector problem](#), where  $b \approx A \cdot s$  is a target vector close to the [lattice](#) point  $v = A \cdot s$ , where  $A$  is a [lattice](#) basis.

**Problem 7.2.4 (Decision learning with errors).** The *decision learning with errors* is to, given  $(A, b)$ , distinguish between  $(A, b \approx A \cdot s)$  and  $(A, b)$  selected uniformly at random.

The relation between [Learning with errors](#) and [decision learning with errors](#) is illustrated by [Theorem 7.2.1](#).

**Theorem 7.2.1.** [Learning with errors](#) and [decision learning with errors](#) are equivalent under a probabilistic polynomial time reduction, i.e., each can be solved efficiently if the other can.

### 7.2.3 Learning with Errors for Key Exchange

Consider a key exchange between Alice and Bob using [learning with errors](#) and the product  $r^\top As$ :

$$\begin{pmatrix} r^\top \end{pmatrix} \cdot \begin{pmatrix} A \end{pmatrix} \cdot \begin{pmatrix} s \end{pmatrix}.$$

Specifically, it works as follows.

1. Alice first chooses a random  $A \leftarrow \mathbb{Z}_q^{n \times n}$  and sends it to Bob.
2. Bob chooses a “short”  $s \leftarrow \mathbb{Z}_q^n$  and sends  $u = A \cdot s + e \approx A \cdot s$  to Alice, where  $e$  is some errors.
3. Alice chooses a short  $r \leftarrow \mathbb{Z}_q^n$  and sends  $v^\top = r^\top A + d^\top \approx r^\top \cdot A$  to Bob, where  $d$  is some errors.
4. Bob calculates a key  $k_B = v^\top \cdot s (r^\top A + d^\top) \cdot s = r^\top As + d^\top s \approx r^\top As$ , where  $d^\top s$  is some noise.
5. Alice calculates a key  $k_A = r^\top \cdot u = r^\top (A \cdot s + e) = r^\top As + r^\top \cdot e \approx r^\top As$ .

Finally,

- if  $k_A$  and  $k_B$  are both between 0 and  $\frac{q}{2}$ , then a 1 is transmitted;
- if  $k_A$  and  $k_B$  are both between  $-\frac{q}{2}$  and 0, then a 0 is transmitted,

and a message 0 corresponds to 0, and a message 1 corresponds to  $\frac{q}{2}$ .

**Note.** Alice and Bob can agree on a bit in the end.

This can be turned into a [public key encryption](#), just like how we turn [Diffie-Hellman](#) into [ElGamal](#):

- to encrypt a bit  $m \in \{0, 1\}$ , Alice can compute  $c \approx r^\top \cdot u + m \cdot (\frac{q}{2})$ ;
- to recover  $m$ , compute

$$p = c - v^\top \cdot s \approx r^\top A \cdot s + m \frac{q}{2} - r^\top As = m \cdot \frac{q}{2},$$

where the  $v^\top \cdot s$  corresponds to  $k_B$ , the first  $r^\top A \cdot s$  corresponds to  $k_A$ , and the  $r^\top As$  corresponds to  $k_B$ .

Everything works as guaranteed by the following.

**Theorem 7.2.2.** This [public key encryption](#) is [CPA secure](#) assuming [decision learning with errors](#) is hard.

# Appendix

# Appendix A

## Acknowledgement

The following is a list of students, organized by the lecture scribe notes they are responsible for.<sup>1</sup>

### A.1 Winter 2023

**Lecture 1.** Pingbang Hu.

**Lecture 2.** Pingbang Hu.

**Lecture 3.** Jason Zeng, Park Szachta, Meredith Benson.

**Lecture 4.** Nancy Liu, Nicklaus Sicilia.

**Lecture 5.** Bryan Nie, Andrew Marshall, Shuangyu Lei.

**Lecture 6.** Mathurin Gagnon, Aditya Sriram, Angelina Zhang, Adam Marakby.

**Lecture 7.** Edison Situ, Zhou Xinyue, Ashley Jeong, Samuel Costa.

**Lecture 8.** Matt Palazzolo, Mingye Chen.

**Lecture 9.** Nicholas Karns, Shufeng Chen, Jai Narayanan.

**Lecture 10.** Jason Obrycki.

**Lecture 11.** Trisha Dani, Nathan Curdo, Anthony Li.

**Lecture 12.** Yiwen Tseng, Erik Zhou, Lilly Phillips, Yoonsung Ji, Dylan Shelton.

**Lecture 13.** Benjamin Miller, Michael Hu, Enzo Metz.

**Lecture 14.** Jeremy Margolin, Katie Wakevainen, Jason Zheng, Jonathan Giha.

**Lecture 15.** Keming Ouyang, Chen Yuxiang, Haoyu Chen, Tao Zhu, Sean Chen.

**Lecture 16.** Dongqi Wu, Kevin Hua, Xun Wang, Benjamin Miller.

**Lecture 17.** Alex Young, Sohil Ramachandra.

---

<sup>1</sup>Noticeably, in the main document, the space of the header is limited, so I only list the main scribe notes I was referring to when organizing.

---

**Lecture 18.** Mei Lanting, Ava Banerjee, Lilly Phillips.

**Lecture 19.** Shaurya Gupta, Hussain Lokhandwala, David Yei.

**Lecture 20.** Madhav Shekhar Sharma, Ethan Kennedy, Yiwen Tseng, Noah Peters, Zhongqi Ma.

**Lecture 21.** Zeyu Chang, Meredith Benson, Ziyun Chi, Trisha Dani, Ethan Kennedy, Eric Leu, Yi Liu, Lilly Phillips.

**Lecture 22.** Trisha Dani, Aidan Gauthier, Ethan Kennedy, Yi Liu, Yiwen Tseng, Ashley Jeong, Vinamr Arya, Jai Narayanan.

**Lecture 23.** Ethan Kennedy, Yiwen Tseng, Aroosh Moulik, Justin Paul, Jeremy Roszko, Erik Zhou, Yi Liu, Nicklaus Sicilia.

**Lecture 24.** Ethan Kennedy, Luke Miga, Yi Liu, Nicklaus Sicilia, Yiwen Tseng, Zhiyuan Chen, Benjamin Miller, Vinamr Arya, Zhongqi Ma, Lauren Friedrich.

**Lecture 25.** Justin Paul, Park Szachta, Ji YoonSung, Sean Chen, Yi Liu, Lauren Friedrich, Julian Lane.

**Lecture 26.** Park Szachta, Lauren Friedrich.

**More on Graph Theory** Jason Thegn, Pingbang Hu.



# Appendix B

## More on Group Theory

Group theory provides a foundation for understanding the mathematical underpinnings of many cryptographic concepts and techniques. Understanding **groups**' structure often provides tools for analyzing the security of certain cryptographic algorithms. We have seen examples of these algorithms already, such as the factoring and the discrete logarithm problems, and there are many, many more. In this chapter, we provide more expository material regarding group theory.

**Note.** The notation is a bit different from what is used in lectures here and there, and also the way things are defined since I want to introduce everything in the most natural way. For example, the way I introduce **quotient groups** is different from most of the textbook.

In particular, we will use *lots of examples*, together with comments and motivations to provide a self-contained introduction to group theory. At the end of the chapter, we will see the *first isomorphism theorem*, which turns out to be natural if we follow everything.

### B.1 Groups

**Example (Protopical example).** The **additive group**  $(\mathbb{Z}, +)$  and the **cyclic group**  $\mathbb{Z}_p = \mathbb{Z} / p\mathbb{Z}$ . Just don't forget that most **groups** are non-commutative.

Generally speaking, **groups** encodes *symmetries*: given an element, there always exist a unique inverse of which w.r.t. an operation. Before written down the formal definition of a **group**, let us first see some examples.

**Example (Additive integers).** Consider the pair  $(\mathbb{Z}, +)$  where  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$  and  $+$  is the associative *addition* operation. Observe that

- the element  $0 \in \mathbb{Z}$  is an *identity*:  $a + 0 = 0 + a = a$  for any  $a \in \mathbb{Z}$ ;
- every element  $a \in \mathbb{Z}$  has an additive *inverse*:  $a + (-a) = (-a) + a = 0$ .

**Example (Nonzero rationals).** Consider the pair  $(\mathbb{Q}^*, \cdot)$  where  $\mathbb{Q}^* = \mathbb{Q} \setminus \{0\}$  be the set of *nonzero rational numbers*, and  $\cdot$  is the associative *multiplication*. Again, we see that

- the element  $1 \in \mathbb{Q}^*$  is an *identity*:  $a \cdot 1 = 1 \cdot a = a$ ;
- for any rational number  $x \in \mathbb{Q}^*$ , there is an inverse  $x^{-1}$  such that  $x \cdot x^{-1} = x^{-1} \cdot x = 1$ .

Now, you might want to make the following definition.

**Definition B.1.1 (Group).** A *group* is a pair  $(G, \star)$  consisting of a set of elements  $G$ , and a binary operator  $\star: G \times G \rightarrow G$ , such that

- (a)  $G$  has an identity element  $1_G$  (or just 1) such that for all  $g \in G$ ,  $1_G \star g = g \star 1_G = g$ ;

- (b)  $\star$  is associative, i.e.,  $(a \star b) \star c = a \star (b \star c)$  for any  $a, b, c \in G$ ;
- (c) every element  $g \in G$  has an inverse  $h \in G$  such that  $g \star h = h \star g = 1_G$ .

**Note (Unimportant pendantic point).** Some authors like to add a “closure” axiom, i.e., to say explicitly that  $g \star h \in G$ . This is implied already by the fact that  $\star$  is a binary operation on  $G$  (and the fact that we already specified  $\star: G \times G \rightarrow G$ ).

**Remark.** It is not required that  $\star$  is commutative, i.e.,  $a \star b = b \star a$ .

**Definition B.1.2 (Abelian group).** A **group**  $(G, \star)$  is *Abelian* if  $\star$  is commutative.

**Notation.** From now on, we will refer a **group** as just  $G$  if the operation is irrelevant or is canonically realized.

**Definition B.1.3 (Order).** The *order* of a **group**  $G$  is the number of elements of  $G$ .

### B.1.1 Lots of Examples

It’s almost impossible to understand algebraic objects without **lots of** examples, so here we see some.

**Example (Non-examples).** Let’s see some non-examples.

- The pair  $(\mathbb{Q}, \cdot)$  is not a **group**: While there is an identity element 1 the element  $0 \in \mathbb{Q}$  does not have an inverse.
- The pair  $(\mathbb{Z}, \cdot)$  is not a **group**: (Why?)
- Let  $\text{Mat}_{2 \times 2}(\mathbb{R})$  be the set of  $2 \times 2$  real matrices. Then  $(\text{Mat}_{2 \times 2}(\mathbb{R}), \cdot)$  (where  $\cdot$  is the matrix multiplication) is not a **group**: Indeed, even though we have an identity matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

we still run into the same issue as before: the zero matrix does not have a multiplicative inverse.

**Example.** Even if we delete the zero matrix from  $\text{Mat}_{2 \times 2}(\mathbb{R})$ , the resulting structure is still not a **group**: those of you that know some linear algebra might recall that any matrix with determinant zero cannot have an inverse.

Let’s resume writing down examples. Here are some **Abelian** examples of **groups**.

**Example (Complex unit circle).** Let  $S^1$  denote the set of complex numbers  $z$  with absolute value one; that is,  $S^1 := \{z \in \mathbb{C} \mid |z| = 1\}$ . Then  $(S^1, \times)$  is a group since

- the complex number  $1 \in S^1$  serves as the identity, and
- each complex number  $z \in S^1$  has an inverse  $1/z$  which is also in  $S^1$ , since  $|z^{-1}| = |z|^{-1} = 1$ .

There is one thing I ought to also check: that is,  $z_1 \times z_2$  is actually still in  $S^1$ . But this follows from the fact that  $|z_1 z_2| = |z_1| |z_2| = 1$ .

Here is an example from number theory as you might remember.

**Example (Addition modulo (Cyclic group)).** Let  $n > 1$  be an integer, and consider the residues (remainders) modulo  $n$ . These form a **group** under addition. We call this the *cyclic group* of **order**  $n$ , and denote it as  $\mathbb{Z}/n\mathbb{Z}$  or  $\mathbb{Z}_n$ , with elements  $\bar{0}, \bar{1}, \dots, \bar{n-1}$ . The identity is  $\bar{0}$ .

**Example (Multiplication mod  $p$ ).** Let  $p$  be a **prime**. Consider the nonzero residues modulo  $p$ , which we denote by  $(\mathbb{Z}/p\mathbb{Z})^\times$  or  $\mathbb{Z}_p^*$ . Then,  $(\mathbb{Z}_p^*, \times)$  is a **group**.

**Exercise.** Why do we need the fact that  $p$  is **prime**?

Here are some non-**Abelian** examples.

**Example (General linear group).** Let  $n$  be a positive integer. Then  $GL_n(\mathbb{R})$  is defined as the set of  $n \times n$  real matrices which has nonzero determinant. It turns out that with this condition, every matrix does indeed have an inverse, so  $(GL_n(\mathbb{R}), \times)$  is a **group**, called the *general linear group*.<sup>a</sup>

<sup>a</sup>The fact that  $GL_n(\mathbb{R})$  is closed under  $\times$  follows from the linear algebra fact that  $\det(AB) = \det A \det B$ .

**Example (Special linear group).** Recall  $GL_n(\mathbb{R})$ , let  $SL_n(\mathbb{R})$  denote the set of  $n \times n$  matrices whose determinant is actually 1. Again, for linear algebra reasons it turns out that  $(SL_n(\mathbb{R}), \times)$  is also a **group**, called the *special linear group*.

**Example (Symmetric group).** Let  $S_n$  be the set of permutations of  $\{1, \dots, n\}$ . By viewing these permutations as functions from  $\{1, \dots, n\}$  to itself, we can consider *compositions*  $\circ$  of permutations. Then the pair  $(S_n, \circ)$  is also a **group**, because

- there is an identity permutation, and
- each permutation has an inverse.

The group  $S_n$  is called the *symmetric group* on  $n$  elements.

If this is a bit confusing, consider  $S^n$  for  $n = 3$ .

**Example ( $S^3$ ).** For example,  $(123)$  is the bijection on  $\{1, 2, 3\}$  reads, from left to right, as

$$1 \mapsto 2, \quad 2 \mapsto 3, \quad 3 \mapsto 1.$$

Formally, the elements of  $S_3$  are  $(1), (12), (23), (13), (123), (132)$ . As an example of composition, we obtain  $(12)(23) = (123)$ , while  $(23)(12) = (132)$ .

$\circ$	(1)	(12)	(23)	(13)	(123)	(132)
(1)	(1)	(12)	(23)	(13)	(123)	(132)
(12)	(12)	(1)	(123)	(132)	(23)	(13)
(23)	(23)	(132)	(1)	(123)	(13)	(12)
(13)	(13)	(123)	(132)	(1)	(12)	(23)
(123)	(123)	(13)	(12)	(23)	(132)	(1)
(132)	(132)	(23)	(13)	(12)	(1)	(123)

Table B.1: The complete binary operation compositions on  $S^3$

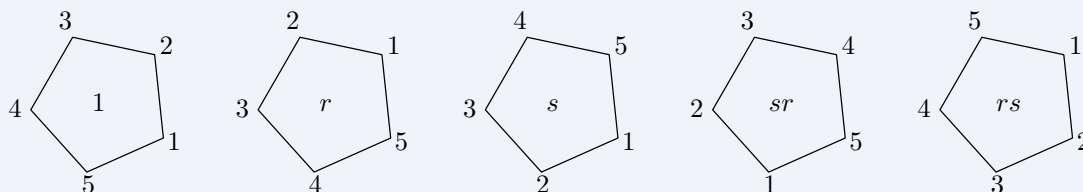
**Example (Dihedral group).** The *dihedral group* of **order**  $2n$ , denoted as  $D_{2n}$ , is the **group** of symmetries of a regular  $n$ -gon  $A_1 A_2 \dots A_n$ , which includes rotations and reflections. It consists of the  $2n$  elements

$$\{1, r, r^2, \dots, r^{n-1}, s, sr, sr^2, \dots, sr^{n-1}\}.$$

The element  $r$  corresponds to rotating the  $n$ -gon by  $2\pi/n$ , while  $s$  corresponds to reflecting it across the line  $OA_1$  for  $O$  being the center of the polygon. So  $rs$  mean “reflect then rotate”, just like function composition, we read from right to left.

In particular,  $r^n = s^2 = 1$ , and it's easy to see that  $r^k s = s r^{-k}$ .

**Example ( $D_{10}$ ).** Here is a picture of some elements of  $D_{10}$ .



**Remark.**  $D_{12}$  is one of my favorite example of a non-Abelian group, and is the first group I try for any exam question of the form “find an example...”.

We can also build a new group based on the exists groups.

**Example (Product of groups).** Let  $(G, \star)$  and  $(H, *)$  be groups. We can define a product group  $(G \times H, \circ)$ , as follows. The elements of the group will be ordered pairs  $(g, h) \in G \times H$ , and we let

$$(g_1, h_1) \circ (g_2, h_2) := (g_1 \star g_2, h_1 * h_2) \in G \times H$$

be the group operation.

**Exercise.** What are the identity and inverses of the product group?

**Example (Trivial group).** The trivial group, often denoted 0 or 1, is the group with only an identity element. However, we use the notation  $\{1\}$ .

**Exercise.** Verify that the trivial group is actually a group.

## B.1.2 Properties of Groups

**Example (Prototypical example).**  $\mathbb{Z}_p^*$  is possibly best.

**Notation.** From now on, given a group  $(G, \star)$ , we will abbreviate  $a \star b$  to just  $ab$ , and since  $\star$  is associative, we will omit unnecessary parentheses, e.g.,  $(ab)c = a(bc) = abc$ .

**Notation.** For any  $g \in G$  and  $n \in \mathbb{N}$ , we abbreviate

$$g^n := \underbrace{g \star \cdots \star g}_{n \text{ times}}.$$

Moreover, we let  $g^{-1}$  denote the inverse of  $g$ , and  $g^{-n} = (g^{-1})^n$ .

In mathematics, a common theme is to require that objects satisfy certain minimalistic properties, with certain examples in mind, but then ignore the examples on paper and try to deduce as much as you can just from the properties alone.<sup>1</sup>

Let's try to do this here, and see what we can conclude just from knowing Definition B.1.1.

<sup>1</sup>For people who know something about functional equations, knowing a single property about a function is enough to determine the entire function.

**Claim.** Let  $G$  be a [group](#).

- (a) The identity of a group is unique.
- (b) The inverse of any element is unique.
- (c) For any  $g \in G$ ,  $(g^{-1})^{-1} = g$ .

**Proof.** This mostly just some formal manipulations, and you needn't feel bad skipping it.

- (a) If  $1$  and  $1'$  are identities, then  $1 = 1 \star 1' = 1'$ .
- (b) If  $h$  and  $h'$  are inverses to  $g$ , then  $1_G = g \star h \Rightarrow h' = (h' \star g) \star h = 1_G \star h = h$ .
- (c) Trivial.

⊗

The following is slightly more useful.

**Proposition B.1.1 (Inverse of products).** Let  $G$  be a [group](#), and  $a, b \in G$ . Then  $(ab)^{-1} = b^{-1}a^{-1}$ .

**Proof.** Since

$$(ab)(b^{-1}a^{-1}) = a(bb^{-1})a^{-1} = aa^{-1} = 1_G,$$

and  $(b^{-1}a^{-1})(ab) = 1_G$  similarly, hence  $(ab)^{-1} = b^{-1}a^{-1}$ . ■

Finally, we state a very important lemma about [groups](#), which highlights why having an inverse is so valuable.

**Lemma B.1.1.** Let  $G$  be a [group](#), and pick a  $g \in G$ . Then the map  $G \rightarrow G$  given by  $x \mapsto gx$  is a bijection.

**Proof.** This can be done by showing injectivity and surjectivity directly. ■

**Example ( $\mathbb{Z}_7^*$ ).** Let  $G = \mathbb{Z}_7^*$  and pick  $g = 3$ . [Lemma B.1.1](#) states that the map  $x \mapsto 3 \cdot x \pmod{7}$ , and we can see this explicitly:

$$\begin{array}{ll} 1 \mapsto 3 \pmod{7}, & 4 \mapsto 5 \pmod{7}, \\ 2 \mapsto 6 \pmod{7}, & 5 \mapsto 1 \pmod{7}, \\ 3 \mapsto 2 \pmod{7}, & 6 \mapsto 4 \pmod{7}. \end{array}$$

**Notation.** Later on, sometimes the identity is denoted  $0$  instead of  $1$ .

In most of our examples up until now, the operation  $\star$  was thought of like multiplication of some sort, which is why  $1 = 1_G$  was a natural notation for the identity element.

But there are [groups](#) like  $\mathbb{Z} = (\mathbb{Z}, +)$  where the operation  $\star$  is thought of as addition, in which case the notation  $0 = 0_G$  might make more sense instead.

**Remark.** In general, whenever an operation is denoted  $+$ , it is almost certainly commutative.

### B.1.3 Unimportant Long Digression

A common question is: why these axioms? For example, why associative but not commutative? One general heuristic is that, whenever you define a new type of general object, there's always a balancing act going on:

- on the one hand, you want to include enough constraints that your objects are “nice”;
- on the other hand, if you include too many constraints, then the definition applies to too few objects.

So, for example, we include “associative” because that makes our lives easier and most operations we run into are associative. In particular, associativity is required to the inverse of an element to necessarily be unique.

However, we don’t include “commutative”, because examples below show that there are lots of non-Abelian groups we care about. Another comment is the following.

**Intuition.** A good motivation for the inverse axioms is that we get a large amount of *symmetry*. The set of positive integers with addition is not a group, for example, because we can’t subtract 6 from 3: some elements are “larger” than others. By requiring an inverse element to exist, we get rid of this issue.

### B.1.4 Lagrange’s Theorem for Orders

**Example** (Prototypical example).  $\mathbb{Z}_p^*$ .

Recall that we say the **order** of a group  $G$  is the number of elements of  $G$ . However, as is typical in math, we use the word “order” for way too many things. In groups, there is another definition.

**Definition B.1.4 (Order).** The *order*  $\text{ord}(g)$  of an element  $g \in G$  is the smallest positive integer  $n$  such that  $g^n = 1_G$ , or  $\infty$  if no such  $n$  exists.

**Example.** The **order** of  $-1$  in  $\mathbb{Q}^\times$  is 2 while the **order** of 1 in  $\mathbb{Z}$  is infinite.

It’s easy to see the following.

**Proposition B.1.2.** If  $g^n = 1_G$ , then  $\text{ord}(g)$  divides  $n$ .

**Remark.** Any element of a group has a finite **order**.

**Proof.** Consider the infinite sequence  $1_G, g, g^2, \dots$ , and find two elements that are the same.  $\circledast$

One of the most first important result is the **Lagrange’s theorem**.

**Theorem B.1.1 (Lagrange’s theorem for orders).** Let  $G$  be any finite group. Then  $x^{|G|} = 1_G$  for any  $x \in G$ .

**Proof.** See **Corollary B.4.1**. You can assume it for now.  $\blacksquare$

## B.2 Isomorphisms

**Example** (Prototypical example).  $\mathbb{Z} \cong 10\mathbb{Z}$ .

Next, we discuss the maps between groups. Consider the following two groups:

- $\mathbb{Z} = (\{\dots, -2, -1, 0, 1, 2, \dots\}, +)$ ;
- $10\mathbb{Z} = (\{\dots, -20, -10, 0, 10, 20, \dots\}, +)$ .

These groups are “different”, but only superficially so – you might even say they only differ in the names of the elements. Specifically, the map

$$\phi: \mathbb{Z} \rightarrow 10\mathbb{Z}, \quad x \mapsto 10x$$

is a bijection of the underlying sets which respects the group operation, i.e.,

$$\phi(x + y) = \phi(x) + \phi(y).$$

**Intuition.**  $\phi$  is a way of re-assigning names of the elements without changing the structure of the group.

This is just formalism for capturing the obvious fact that  $(\mathbb{Z}, +)$  and  $(10\mathbb{Z}, +)$  are the same thing. Formally, we have the following.

**Definition B.2.1 (Isomorphism).** Let  $G = (G, \star)$  and  $H = (H, *)$  be groups. A bijection  $\phi: G \rightarrow H$  is called an *isomorphism* if

$$\phi(g_1 \star g_2) = \phi(g_1) * \phi(g_2)$$

for all  $g_1, g_2 \in G$ .

**Notation (Isomorphic).**  $G$  and  $H$  are *isomorphic*, denoted as  $G \cong H$ , if there exists an isomorphism from  $G$  to  $H$ .

**Note.** The left-hand side  $\phi(g_1 \star g_2)$  uses the operation of  $G$ , while the right-hand side  $\phi(g_1) * \phi(g_2)$  uses the operation of  $H$ .

## B.2.1 Lots of Examples

Let's see some examples.

**Example.** Let  $G$  and  $H$  be groups. We have the following isomorphisms.

- $\mathbb{Z} \cong 10\mathbb{Z}$ .
- There is an isomorphism  $G \times H \cong H \times G$  by the map  $(g, h) \mapsto (h, g)$ .
- The identity map  $\text{id}: G \rightarrow G$  is an isomorphism, hence  $G \cong G$ .
- There is another isomorphism of  $\mathbb{Z}$  to itself: send every  $x$  to  $-x$ .

**Example (Primitive roots modulo 7).** As a nontrivial example, we claim that  $\mathbb{Z}_6 \cong \mathbb{Z}_7^*$ , where the bijection is given by

$$\phi(a \bmod 6) = 3^a \bmod 7.$$

**Proof.**  $\phi$  is a bijection by explicit calculation<sup>a</sup>

$$(3^0, 3^1, 3^2, 3^3, 3^4, 3^5) \equiv (1, 3, 2, 6, 4, 5) \pmod{7}.$$

Moreover,  $\phi$  respects the group actions, i.e.,  $\phi(a + b) = \phi(a) \times \phi(b)$  since the operation of  $\mathbb{Z}_6$  and  $\mathbb{Z}_7^*$  is  $+$  and  $\times$ , respectively. But this is just saying

$$3^{a+b \bmod 6} \equiv 3^a \bmod 6 3^b \bmod 6 \pmod{7},$$

which is true. ⊛

---

<sup>a</sup>To be pedantic, I should write  $3^0 \bmod 6 = 1$ , etc.

More generally, we have the following.

**Example (Primitive roots).** For any prime  $p$ , there exists an element  $g \in \mathbb{Z}_p^*$  called a *primitive root* modulo  $p$  such that  $1, g, g^2, \dots, g^{p-2}$  are all different modulo  $p$ . One can show by copying the above proof that  $\mathbb{Z}_{p-1} \cong \mathbb{Z}_p^*$  for all primes  $p$ .

**Exercise.** Assuming the existence of primitive roots, give the isomorphism explicitly between  $\mathbb{Z}_{p-1}$  and  $\mathbb{Z}_p^*$ .

Now, you might wonder why we care about **isomorphism**? The answer is that it's an **equivalence relation**.

**Definition B.2.2 (Equivalence relation).** A relation is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

**Equivalence relations** appear everywhere in mathematics.

**Example.** Here are some examples.

- (a) The relation  $R$  on  $\mathbb{Z} \times \mathbb{N}$  by  $(n_1, m_1)R(n_2, m_2)$  provided that  $n_1m_2 = n_2m_1$ .
- (b) The relation  $R$  on  $\mathbb{C}$  by  $z_1Rz_2$  if and only if  $|z_1| = |z_2|$ .
- (c) For an  $n \in \mathbb{Z}$ , define a relation  $R$  on  $\mathbb{Z}$  by  $m_1Rm_2$  provided that  $m_1 - m_2$  is a multiple of  $n$ .

**Intuition.** Given a **equivalence relation**, we can start by talking about things that should be viewed as “equal”.

Often time, in mathematics, after you define something, you would want to classify them. In this case, **isomorphisms** help us classify **groups** since it's not hard to see that  $\cong$  is an **equivalence relation**. Moreover, because we really only care about the structure of **groups**, we usually consider two **groups** to be the same when they are **isomorphic**.

## B.2.2 Subgroups

**Example (Prototypical example).**  $\text{SL}_n(\mathbb{R})$  is a **subgroup** of  $\text{GL}_n(\mathbb{R})$ .

Earlier we saw that  $\text{GL}_n(\mathbb{R})$ , the  $n \times n$  matrices with nonzero determinant, formed a **group** under matrix multiplication. But we also saw that a subset of  $\text{GL}_n(\mathbb{R})$ , the  $\text{SL}_n(\mathbb{R})$ , also formed a **group** with the same operation. This property should have its own name.

**Definition B.2.3 (Subgroup).** Let  $G = (G, \star)$  be a **group**. A *subgroup* of  $G$  is a **group**  $H = (H, \star)$  where  $H \subseteq G$ .

**Notation (Proper subgroup).** We say  $H$  is a *proper subgroup* of  $G$  if  $H \subsetneq G$ .

**Remark.** To specify a **group**  $G$ , I need to tell you both what the set  $G$  was and the operation  $\star$  was. But to specify a **subgroup**  $H$  of  $G$ , I only need to tell you who its elements are since the operation of  $H$  is just inherited from  $G$ .

**Example.** For any **group**  $G$ , the **trivial group**  $\{1_G\}$  and the entire **group**  $G$  are **subgroup** of  $G$ .

**Example.**  $2\mathbb{Z}$  is a **subgroup** of  $\mathbb{Z}$ , which is **isomorphic** to  $\mathbb{Z}$  itself.

**Example.** Consider the **symmetric group**  $S_n$  on  $n$  elements. Let  $T$  be the set of permutations  $\tau: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  for which  $\tau(n) = n$ . Then  $T$  is a **subgroup** of  $S_n$ , and is **isomorphic** to  $S_{n-1}$ .

**Intuition.** We're fixing the last element of the permutation.

**Example.** Consider the **product group**  $G \times H$  between  $G$  and  $H$  and the set  $\{(g, 1_H) \mid g \in G\}$ . This is a **subgroup** of  $G \times H$ , and is **isomorphic** to  $G$  by the **isomorphism**  $(g, 1_H) \mapsto g$ .



Next is an especially important example.

**Example (Generated subgroup).** Let  $x$  be an element of a **group**  $G$ . Consider the set

$$\langle x \rangle := \{\dots, x^{-2}, x^{-1}, 1, x, x^2, \dots\}.$$

This is a **subgroup** of  $G$  called the *subgroup generated by  $x$* .

Finally, we present some non-examples of **subgroups**.

**Example.** Consider the **group**  $\mathbb{Z} = (\mathbb{Z}, +)$ .

- (a) The set  $\{0, 1, 2, \dots\}$  is not a **subgroup** of  $\mathbb{Z}$  since it does not contain inverses.
- (b) The set  $\{n^3 \mid n \in \mathbb{Z}\} = \{\dots, -8, -1, 0, 1, 8, \dots\}$  is not a **subgroup** since it's not closed under addition: the sum of two cubes is not in general a cube.
- (c) The empty set  $\emptyset$  is not a **subgroup** of  $\mathbb{Z}$  because it lacks an identity element.

## B.3 Homomorphisms

### B.3.1 Generators and Group Presentations

**Example (Prototypical example).**  $D_{2n} = \langle r, s \mid r^n = s^2 = 1 \rangle$ .

Let  $G$  be a **group**. Recall that for some element  $x \in G$ , we could consider the **subgroup**

$$\{\dots, x^{-2}, x^{-1}, 1, x, x^2, \dots\}$$

of  $G$ . Here's a more pictorial version of what we did:

**Intuition.** Put  $x$  in a box, seal it tightly, and shake vigorously.

Using just the element  $x$ , we get a pretty explosion that produces the **subgroup** above. What happens if we put two elements  $x, y$  in the box? Among the elements that get produced are things like

$$xyxyx, \quad x^2y^9x^{-5}y^3, \quad y^{-2015}, \quad \dots$$

Essentially, I can create any finite product of  $x, y, x^{-1}, y^{-1}$ . This leads us to define the following.

**Definition B.3.1 (Generated subgroup).** Let  $S$  be a subset of  $G$ . The **subgroup generated by  $S$** , denoted  $\langle S \rangle$ , is the set of elements which can be written as a finite product of elements in  $S$  (and their inverses).

**Definition B.3.2 (Generator).** If  $\langle S \rangle = G$  then we say  $S$  is a set of *generators* for  $G$ , as the elements of  $S$  together create all of  $G$ .

**Exercise.** Why is the condition “and their inverses” not necessary if  $G$  is a finite **group** (assume Lagrange's theorem)?

**Example ( $\mathbb{Z}$  is the infinite cyclic group).** Consider 1 as an element of  $\mathbb{Z} = (\mathbb{Z}, +)$ . We see  $\langle 1 \rangle = \mathbb{Z}$ , meaning  $\{1\}$  generates  $\mathbb{Z}$ . It's important that  $-1$ , the inverse of 1 is also allowed: we need it to write all integers as the sum of 1 and  $-1$ .

This gives us an idea for a way to try and express **groups** compactly.

**Problem.** Why not just write down a list of generators for the [group](#)? For example, we could write  $\mathbb{Z} \cong \langle a \rangle$  meaning that  $\mathbb{Z}$  is just the [group generated](#) by one element.

**Answer.** There's one issue: the [generators](#) usually satisfy certain properties. For example, consider  $\mathbb{Z}_{100}$ . It's also generated by a single element  $x$ , but this  $x$  has the additional property that  $x^{100} = 1$ . \*

This motivates us to write

$$\mathbb{Z}_{100} = \langle x \mid x^{100} = 1 \rangle.$$

I'm sure you can see where this is going. All we have to do is specify a set of generators and *relations* between the generators, and say that two elements are equal if and only if you can get from one to the other using relations.

**Definition B.3.3 (Group presentation).** Such an expression is appropriately called a *group presentation*.

**Example (Dihedral group).** The  $D_{2n}$  has a [presentation](#)

$$D_{2n} = \langle r, s \mid r^n = s^2 = 1, rs = sr^{-1} \rangle.$$

Thus, each element of  $D_{2n}$  can be written uniquely in the form  $r^\alpha$  or  $sr^\alpha$ , where  $\alpha = 0, 1, \dots, n-1$ .

**Example (Klein four group).** The *Klein four group*, [isomorphic](#) to  $\mathbb{Z}_2 \times \mathbb{Z}_2$ , is given by the [presentation](#)

$$\langle a, b \mid a^2 = b^2 = 1, ab = ba \rangle.$$

**Example (Free group).** The *free group on  $n$  elements* is the [group](#) whose [presentation](#) has  $n$  [generators](#) and no relations at all. It is denoted  $F_n$ , so

$$F_n = \langle x_1, x_2, \dots, x_n \rangle.$$

In other words,  $F_2 = \langle a, b \rangle$  is the set of strings formed by appending finitely many copies of  $a$ ,  $b$ ,  $a^{-1}$ ,  $b^{-1}$  together.

**Remark.** Notice that  $F_1 \cong \mathbb{Z}$ .

**Notation.** One might unfortunately notice that “[subgroup generated](#) by  $a$  and  $b$ ” has exactly the same notation as the [free group](#)  $\langle a, b \rangle$ . We'll try to be clear based on context which one we mean.

[Presentations](#) are nice because they provide a compact way to write down [groups](#). They do have some shortcomings, though.<sup>2</sup>

**Example (Presentations can look very different).** The same [group](#) can have very different [presentations](#). For instance consider

$$D_{2n} = \langle x, y \mid x^2 = y^2 = 1, (xy)^n = 1 \rangle.$$

**Proof.** To see why this is equivalent, set  $x = s$ ,  $y = rs$ . \*

## B.3.2 Homomorphisms

**Example (Prototypical example).** The “mod out by 100” map,  $\mathbb{Z} \rightarrow \mathbb{Z}_{100}$ .

<sup>2</sup>Actually, determining whether two elements of a [presentation](#) are equal is undecidable. In fact, it is undecidable to even determine if a [group](#) is finite from its [presentation](#).

How can **groups** talk to each other?

**As previously seen.** Two **groups** are “the same” if we can write an **isomorphism** between them.

If you know something about “metric space” (i.e., space with a distance function), two metric spaces are “the same” if we can write a “homeomorphism” (i.e., continuous and bijective) between them. But what’s the **group** analogy of a continuous map? We simply drop the bijection condition.

**Definition B.3.4 (Group homomorphism).** Let  $G = (G, \star)$  and  $H = (H, *)$  be **groups**. A *group homomorphism* is a map  $\phi: G \rightarrow H$  such that for any  $g_1, g_2 \in G$ ,

$$\phi(g_1 \star g_2) = \phi(g_1) * \phi(g_2).$$

**Intuition.** A **homomorphism** between two **groups**  $G, H$  makes the **groups** “behave the same”. In other words, we can “translate” any action performed in one of these **groups** into an action performed in the other.

Let’s see some examples of **homomorphisms**: Let  $G$  and  $H$  be **groups**.

**Example (Trivial homomorphism).** The *trivial homomorphism*  $G \rightarrow H$  sends everything to  $1_H$ .

**Example.** Any **isomorphism**  $G \rightarrow H$  is a **homomorphism**. In particular, the identity map  $G \rightarrow G$  is a **homomorphism**.

**Example.** There is a **homomorphism** from  $\mathbb{Z}$  to  $\mathbb{Z}_{100}$  by sending each integer to its residue modulo 100.

**Example.** There is a **homomorphism** from  $\mathbb{Z}$  to itself by  $x \mapsto 10x$  which is injective but not surjective.

**Example.** There is a **homomorphism** from  $S_n$  to  $S_{n+1}$  by “embedding”: every permutation on  $\{1, \dots, n\}$  can be thought of as a permutation on  $\{1, \dots, n+1\}$  if we simply let  $n+1$  be a fixed point.

**Example.** A **homomorphism**  $\phi: D_{12} \rightarrow D_6$  is given by  $s_{12} \mapsto s_6$  and  $r_{12} \mapsto r_6$ .

**Example.** Specifying a **homomorphism**  $\mathbb{Z} \rightarrow G$  is the same as specifying just the image of the element  $1 \in \mathbb{Z}$ . (Why?)

The last two examples illustrate something.

**Remark.** Suppose we have a **presentation** of  $G$ . To specify a **homomorphism**  $G \rightarrow H$ , we only have to specify where each **generator** of  $G$  goes, in such a way that the relations are all satisfied.

**Remark.** The right way to think about an **isomorphism** is as a “bijective **homomorphism**”.

To be explicit,

**Exercise.** Show that  $G \cong H$  if and only if there exist **homomorphisms**  $\phi: G \rightarrow H$  and  $\psi: H \rightarrow G$  such that  $\phi \circ \psi = \text{id}_H$  and  $\psi \circ \phi = \text{id}_G$ .

**Note.** So the definitions of homeomorphism of metric spaces and **isomorphism** of **groups** are not too different.

Some obvious properties of **homomorphisms** follow.

**Lemma B.3.1.** Let  $\phi: G \rightarrow H$  be a **homomorphism**. Then  $\phi(1_G) = 1_H$  and  $\phi(g^{-1}) = \phi(g)^{-1}$ .

**Proof.** Let  $g \in G$ . We know that  $\phi(g) = \phi(g \star 1_G) = \phi(g) * \phi(1_G)$ . Furthermore,  $\phi(g) = \phi(g) * \phi(1_G) = \phi(g) * 1_H$  by definition of identity. Therefore,  $\phi(1_G) = 1_H$ .

Moreover,  $\phi(1_G) = 1_H = \phi(g \star g^{-1}) = \phi(g) * \phi(g^{-1})$ . Furthermore,  $1_H = \phi(g) * \phi(g^{-1}) = \phi(g) * \phi(g)^{-1}$ . Therefore,  $\phi(g^{-1}) = \phi(g)^{-1}$ . ■

**Lemma B.3.2.** The composition of two **homomorphisms** is a **homomorphism**.

**Proof.** Let  $(G, \star), (H, *), (I, \cdot)$  be **groups**, and let  $\phi: G \rightarrow H, \psi: H \rightarrow I$  be two **homomorphisms**, we show that  $\psi \circ \phi: g \rightarrow I$  is a **homomorphism**. Let  $g_1, g_2 \in G$ , we have

$$\psi \circ \phi(g_1 \star g_2) = \psi(\phi(g_1) * \phi(g_2)) = \psi(\phi(g_1)) \cdot \psi(\phi(g_2)) = \psi \circ \phi(g_1) \cdot \psi \circ \phi(g_2).$$

■

### B.3.3 Kernels

Now let me define a very important property of a **homomorphism**.

**Definition B.3.5 (Kernel).** The *kernel* of a **homomorphism**  $\phi: G \rightarrow H$  is defined by

$$\ker \phi := \{g \in G: \phi(g) = 1_H\}.$$

**Remark.** The **kernel** of a **homomorphism**  $\phi: G \rightarrow H$  is a **subgroup** of  $G$ . In particular,  $1_G \in \ker \phi$  for obvious reasons.

We also have the following important fact.

**Theorem B.3.1 (Kernel determines injectivity).** Let  $\phi: G \rightarrow H$  be a group **homomorphism**. Show that  $\phi$  is injective if and only if  $\ker(\phi) = \{1_G\}$ .

**Proof.** Assume  $\phi$  is injective. Since  $\phi(1_G) = 1_H$ , by injectivity, for all  $x \in G, \phi(x) = 1_H \rightarrow x = 1_G$ . Therefore,  $\ker \phi = \{1_G\}$ .

Now, assume  $\ker \phi = \{1_G\}$ . Fix  $x, y \in G$  such that  $\phi(x) = \phi(y)$ .  $\phi(x) \cdot \phi(y)^{-1} = 1_G$ . We know that  $\phi(x) \cdot \phi(y)^{-1} = \phi(x) \cdot \phi(y^{-1}) = \phi(x \cdot y^{-1})$ . Furthermore,  $x \cdot y^{-1} \in \ker \phi$ , therefore  $x \cdot y^{-1} = 1_G$ , therefore  $x = y$ . We conclude that  $\phi(x) = \phi(y) \rightarrow x = y$ , and  $\phi$  is injective. ■

**Remark.** This extends naturally to vector spaces, i.e., any linear transformation  $T: V \rightarrow W$  is injective if and only if  $\ker(T) = \{0_V\}$ , where  $V, W$  are vector spaces.

To make this concrete, let's compute the **kernel** of each of our examples.

**Example.** The **kernel** of any **isomorphism**  $G \rightarrow H$  is trivial, since an **isomorphism** is injective. In particular, the **kernel** of the identity map  $G \rightarrow G$  is  $\{1_G\}$ .

**Example.** The **kernel** of the trivial **homomorphism**  $G \rightarrow H$  (by  $g \mapsto 1_H$ ) is all of  $G$ .

**Example.** The **kernel** of the **homomorphism**  $\mathbb{Z} \rightarrow \mathbb{Z}_{100}$  by  $n \mapsto \bar{n}$  is precisely

$$100\mathbb{Z} = \{\dots, -200, -100, 0, 100, 200, \dots\}.$$

**Example.** The **kernel** of the map  $\mathbb{Z} \rightarrow \mathbb{Z}$  by  $x \mapsto 10x$  is trivial:  $\{0\}$ .

**Example.** There is a **homomorphism** from  $S_n$  to  $S_{n+1}$  by “embedding”, but it also has trivial **kernel** because it is injective.

**Example.** A **homomorphism**  $\phi: D_{12} \rightarrow D_6$  is given by  $s_{12} \mapsto s_6$  and  $r_{12} \mapsto r_6$ . You can check that

$$\ker \phi = \{1, r_{12}^3\} \cong \mathbb{Z}_2.$$

**Exercise.** Fix any  $g \in G$ . Suppose we have a **homomorphism**  $\mathbb{Z} \rightarrow G$  by  $n \mapsto g^n$ . What is the **kernel**?

**Exercise.** Show that for any **homomorphism**  $\phi: G \rightarrow H$ , the image  $\phi(G)$  is a **subgroup** of  $H$ . Hence, we’ll be especially interested in the case where  $\phi$  is surjective.

**Theorem B.3.2.** Let  $H$  be a nonempty subset of  $G$  such that for all  $h_1, h_2 \in H$  we have  $h_1 h_2^{-1} \in H$ . Then,  $H$  is a **subgroup** of  $G$ .

**Proof.** Since  $H$  is nonempty, let  $h \in H$ . It is obvious that  $h h^{-1} = 1_G \in H$ . Since  $h \in H, 1_G \in H, 1_g h \in H$ .  $1_g h^{-1} = h^{-1} \in H$ . Let  $h_1, h_2, h_3 \in H$ . Since these are elements of  $G$ , by associativity,  $h_1(h_2 h_3) = (h_1 h_2) h_3$ . Therefore,  $H$  is a **group**, and  $H$  is a **subgroup** of  $G$ . ■

## B.4 Quotient Groups

### B.4.1 Cosets

**Example (Prototypical example).** Modding out by  $n$ :  $\mathbb{Z}/(n \cdot \mathbb{Z}) \cong \mathbb{Z}_n$ .

**Note.** The next few sections are a bit dense.

Let  $G$  and  $Q$  be **groups**, and suppose there exists a **surjective homomorphism**

$$\phi: G \twoheadrightarrow Q.$$

In other words, if  $\phi: G \rightarrow Q$  is injective then  $\phi$  is a bijection, and hence an **isomorphism**.

**Problem.** Suppose we’re not so lucky and  $\ker \phi$  is bigger than just  $\{1_G\}$ , what is the correct interpretation of a more general **homomorphism**?

Let’s look at the special case where  $\phi: \mathbb{Z} \rightarrow \mathbb{Z}_{100}$  is “modding out by 100”. We already saw that the **kernel** of this map is

$$\ker \phi = 100\mathbb{Z} = \{\dots, -200, -100, 0, 100, 200, \dots\}.$$

**As previously seen.** Recall that  $\ker \phi$  is a **subgroup** of  $G$ .

What this means is that  $\phi$  is **indifferent to the subgroup**  $100\mathbb{Z}$  of  $\mathbb{Z}$ , i.e.,

$$\phi(15) = \phi(2000 + 15) = \phi(-300 + 15) = \phi(700 + 15) = \dots$$

So  $\mathbb{Z}_{100}$  is what we get when we “mod out by 100”. Cool.

In other words, let  $G$  be a **group** and  $\phi: G \twoheadrightarrow Q$  be a **surjective homomorphism** with **kernel**  $N \subseteq G$ . We claim the following.

**Intuition.**  $Q$  should be thought of as the “quotient” of  $G$  by  $N$ .

To formalize this, we will define a so-called **quotient group**  $G/N$  in terms of  $G$  and  $N$  only (without referencing  $Q$ ) which will be naturally **isomorphic** to  $Q$ . For motivation, let’s give a concrete description

of  $Q$  using just  $\phi$  and  $G$ . Continuing our previous example, let  $N = 100\mathbb{Z}$  be our [subgroup](#) of  $G$ . Consider the sets

$$\begin{aligned} N &= \{\dots, -200, -100, 0, 100, 200, \dots\} \\ 1 + N &= \{\dots, -199, -99, 1, 101, 201, \dots\} \\ 2 + N &= \{\dots, -198, -98, 2, 102, 202, \dots\} \\ &\vdots \\ 99 + N &= \{\dots, -101, -1, 99, 199, 299, \dots\}. \end{aligned}$$

**Note.** The elements of each set all have the same image when we apply  $\phi$ . Moreover, any two elements in different sets have different images.

Then the main idea is the following.

**Intuition.** We can think of  $Q$  as the [group](#) whose elements are the *sets* above.

Thus, given  $\phi$ , we define an [equivalence relation](#)  $\sim_N$  on  $G$  by saying  $x \sim_N y$  for  $\phi(x) = \phi(y)$ . This  $\sim_N$  divides  $G$  into several equivalence classes in  $G$  which are in obvious bijection with  $Q$ , as above. Now we claim that we can write these equivalence classes very explicitly.

**Exercise.** Show that  $x \sim_N y$  if and only if  $x = yn$  for some  $n \in N$ .<sup>a</sup> Thus, for any  $g \in G$ , the equivalence class of  $\sim_N$  which contains  $g$  is given explicitly by

$$gN := \{gn \mid n \in N\}.$$

<sup>a</sup>In the mod 100 example, this means they “differ by some multiple of 100”.

Here’s the word that describes the types of sets we’re running into now.

**Definition B.4.1 (Left coset).** Let  $H$  be any [subgroup](#) of  $G$ .<sup>a</sup> A set of the form  $gH$  is called a *left coset* of  $H$ .

<sup>a</sup>Not necessarily the [kernel](#) of some [homomorphism](#).

**Remark.** Although the notation might not suggest it, keep in mind that  $g_1N$  is often equal to  $g_2N$  even if  $g_1 \neq g_2$ . In the “mod 100” example,  $3 + N = 103 + N$ . In other words, these [cosets](#) are *sets*.

This means that if I write “let  $gH$  be a [coset](#)” without telling you what  $g$  is, you can’t figure out which  $g$  I chose from just the [coset](#) itself. If you don’t believe me, here’s an example of what I mean.

**Example.** Consider

$$x + 100\mathbb{Z} = \{\dots, -97, 3, 103, 203, \dots\} \Rightarrow x = ?$$

There’s no reason to think I picked  $x = 3$  (I actually picked  $x = -13597$ ).

You might already notice it, but let’s make the following explicit.

**Lemma B.4.1.** All [cosets](#) have the same cardinality.

**Proof.** Given [cosets](#)  $g_1H$  and  $g_2H$ , you can check that the map  $x \mapsto g_2g_1^{-1}x$  is a bijection between them. ■

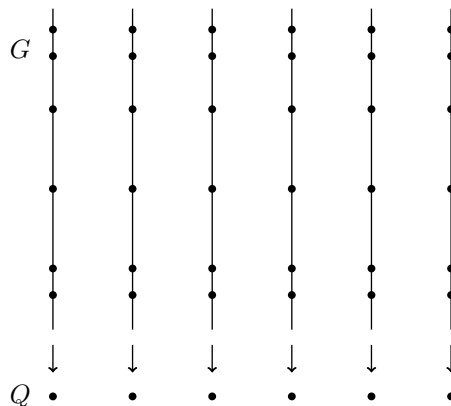
So, long story short,

**Intuition.** Elements of the [group](#)  $Q$  are naturally identified with [left cosets](#) of  $N$ .

In practice, people often still prefer to picture elements of  $Q$  as single points.

**Example.** It's easier to think of  $\mathbb{Z}_2$  as  $\{0, 1\}$  rather than  $\{\dots, -2, 0, 2, \dots\}, \{\dots, -1, 1, 3, \dots\}$ .

If you like this picture, then you might then draw  $G$  as a bunch of equally tall “fibers” (the [cosets](#)), which are then “collapsed” onto  $Q$ .



## B.4.2 Quotient Groups

Now that we've done this, we can give an *intrinsic* definition for the [quotient group](#) we alluded to earlier.

**Definition B.4.2 (Normal).** A [subgroup](#)  $N$  of  $G$  is called *normal* if it is the [kernel](#) of some [homomorphism](#).

**Notation.** We write  $N \trianglelefteq G$  for  $N$  being [normal](#) to  $G$ .

We can now write down the first version of the definition of a [quotient group](#).

**Definition B.4.3 (Quotient group).** Let  $N \trianglelefteq G$ . Then the *quotient group*, denoted  $G/N$  (and read “ $G \bmod N$ ”), is the [group](#) defined as follows.

- The elements of  $G/N$  will be the [left cosets](#) of  $N$ .
- To define the product of two [cosets](#)  $C_1$  and  $C_2$  in  $G/N$ , recall that the [cosets](#) are in bijection with elements of  $Q$ . So let  $q_1$  be the value associated to the [coset](#)  $C_1$ , and  $q_2$  the one for  $C_2$ . Then we can take the product to be the [cosets](#) corresponding to  $q_1 q_2$ .

Quite importantly, to define the operation of a [quotient group](#), we can also do this in terms of **representatives of the cosets**: Let  $g_1 \in C_1$  and  $g_2 \in C_2$ , so  $C_1 = g_1 N$  and  $C_2 = g_2 N$ . Then  $C_1 \cdot C_2$  should be the [coset](#) which contains  $g_1 g_2$ .

**Claim.** This is the same as [Definition B.4.3](#).

**Proof.** We simply observe that  $\phi(g_1 g_2) = \phi(g_1) \phi(g_2) = q_1 q_2$ , i.e., all we've done is define the product in terms of elements of  $G$ , rather than values in  $H$ . ⊛

Using the  $gN$  notation, and with [the remark](#) in mind, we can write this even more succinctly as

$$(g_1 N) \cdot (g_2 N) := (g_1 g_2) N.$$

We now write down the concise version of [Definition B.4.3](#).

**Definition B.4.4 (Quotient group).** Let  $N \trianglelefteq G$ . Then the *quotient group*  $G/N$  is defined as  $(\{gN\}_{g \in G}, \cdot)$  where

$$(g_1 N) \cdot (g_2 N) := (g_1 g_2) N.$$

Now, you see why the integers modulo  $n$  are often written as  $\mathbb{Z}/n\mathbb{Z}$ ! By the way we've built it, the resulting [group](#)  $G/N$  is [isomorphic](#) to  $Q$ .

**Intuition.** In a sense we think of  $G/N$  as “ $G$  modulo the condition that  $n = 1$  for all  $n \in N$ ”.

### B.4.3 Lagrange’s Theorem

As an aside, with the language of [cosets](#) we can now show [Lagrange’s theorem](#) in the general case.

**Theorem B.4.1** (Lagrange’s theorem). Let  $G$  be a finite [group](#), and let  $H$  be any [subgroup](#). Then  $|H|$  divides  $|G|$ .

**Proof.** We first note that  $G/H = \{g_1H, \dots, g_nH\}$  (perhaps with repetitions) where  $n = |G| < \infty$ , and by [Lemma B.4.1](#), for all  $1 \leq i \leq j \leq n$ ,  $|g_iH| = |g_jH|$ . Moreover,  $|g_iH| = |H|$ .

Now, since  $G$  is finite, so it is the union of a finite number of disjoint [cosets](#), i.e.,

$$G = \bigcup_{i=1}^n g_iH \Rightarrow |G| = \sum_{i=1}^n |g_iH|$$

since  $g_iH$  are all disjoint. But with the fact that  $|g_iH| = |H|$ ,  $|G| = n|H|$ . ■

**Corollary B.4.1.**  $x^{|G|} = 1$ .

**Proof.** By taking  $H = \langle x \rangle \subseteq G$ . ■

It should be mentioned at this point that in general, we have the following.

**Remark.** If  $G$  is a finite [group](#) and  $N$  is [normal](#), then  $|G/N| = |G|/|N|$ .

### B.4.4 Eliminating the Homomorphism

**Example** (Prototypical example). Again  $\mathbb{Z}/n\mathbb{Z} \cong \mathbb{Z}_n$ .

As previously seen. Recall [Definition B.4.4](#) of  $G/N$ :

- The elements of  $G/N$  are [cosets](#)  $gN$ , which you can think of as equivalence classes of a relation  $\sim_N$  (where  $g_1 \sim_N g_2$  if  $g_1 = g_2n$  for some  $n \in N$ ).
- Given [cosets](#)  $g_1N$  and  $g_2N$  the [group](#) operation is  $g_1N \cdot g_2N := (g_1g_2)N$ .

**Problem.** Where do we actually use the fact that  $N$  is [normal](#)? We don’t talk about  $\phi$  or  $Q$  anywhere in this definition.

**Answer.** The answer is in [the remark](#). The [group](#) operation takes in two [cosets](#), so it doesn’t know what  $g_1$  and  $g_2$  are. But behind the scenes, the [normal condition](#) guarantees that the [group operation](#) can pick any  $g_1$  and  $g_2$  it wants and still end up with the same [coset](#).

If we didn’t have this property, then it would be hard to define the product of two [cosets](#)  $C_1$  and  $C_2$  because it might make a difference which  $g_1 \in C_1$  and  $g_2 \in C_2$  we picked (hence it’s not *well-defined*). The fact that  $N$  came from a [homomorphism](#) meant we could pick any representatives  $g_1$  and  $g_2$  of the [cosets](#) we wanted, because they all had the same  $\phi$ -value. ⊛

We want some conditions which force this to be true without referencing  $\phi$  at all. Suppose  $\phi: G \rightarrow K$  is a [homomorphism](#) of [groups](#) with  $H = \ker \phi$ . Aside from the fact  $H$  is a [group](#), we can get an “obvious” property:

**Exercise.** Show that if  $h \in H$ ,  $g \in G$ , then  $ghg^{-1} \in H$ . (Check  $\phi(ghg^{-1}) = 1_K$ .)



**Example (Non-normal subgroup).** Let  $D_{12} = \langle r, s \mid r^6 = s^2 = 1, rs = sr^{-1} \rangle$ . Consider the subgroup of order two  $H = \{1, s\}$  and notice that

$$rsr^{-1} = r(sr^{-1}) = r(rs) = r^2s \notin H.$$

Hence,  $H$  is not normal, and cannot be the kernel of any homomorphism.

Well, duh – so what? Amazingly it turns out that this is the *sufficient* condition we want. Specifically, it makes the nice “coset multiplication” we wanted work out. Thus we have a new criterion for “normal” subgroups which does not make any external references to  $\phi$ .

**Theorem B.4.2 (Algebraic condition for normal subgroups).** Let  $H$  be a subgroup of  $G$ . Then  $H \trianglelefteq G$  if and only if for every  $g \in G$  and  $h \in H$ ,  $ghg^{-1} \in H$ .

**Proof.** We already showed the forward direction. For the other direction, we need to build a homomorphism with kernel  $H$ . So we simply define the group  $G/H$  as the cosets. To put a group operation, we need to verify the following.

**Claim.** If  $g'_1 \sim_H g_1$  and  $g'_2 \sim_H g_2$  then  $g'_1 g'_2 \sim_H g_1 g_2$ .

**Proof.** Let  $g'_1 = g_1 h_1$  and  $g'_2 = g_2 h_2$ , so we want to show that  $g_1 h_1 g_2 h_2 \sim_H g_1 g_2$ . Since  $H$  has the property,  $g_2^{-1} h_1 g_2$  is some element of  $H$ , say  $h_3$ . Thus,  $h_1 g_2 = g_2 h_3$ , and the left-hand side becomes  $g_1 g_2 (h_3 h_2)$ , which is fine since  $h_3 h_2 \in H$ .  $\otimes$

With that settled we can just define the product of two cosets (of normal subgroups) by

$$(g_1 H) \cdot (g_2 H) = (g_1 g_2) H.$$

Thus, the claim above shows that this multiplication is well-defined, so  $G/H$  is indeed a group! Moreover, there is an obvious “projection” homomorphism  $G \rightarrow G/H$  (with kernel  $H$ ), by  $g \mapsto gH$ .  $\blacksquare$

**Example (Modding out in the product group).** Consider again the product group  $G \times H$ . Earlier we identified a subgroup

$$G' = \{(g, 1_H) \mid g \in G\} \cong G.$$

You can easily see that  $G' \trianglelefteq G \times H$ . Moreover, you can check that

$$(G \times H)/(G') \cong H.$$

Indeed, we have  $(g, h) \sim_{G'} (1_G, h)$  for all  $g \in G$  and  $h \in H$ .

It is not necessarily true that  $(G/H) \times H \cong G$ .

**Example (Quotients and products don't necessarily cancel).** Consider  $G = \mathbb{Z}/4\mathbb{Z}$  and the normal subgroup  $H = \{0, 2\} \cong \mathbb{Z}/2\mathbb{Z}$ . Then  $G/H \cong \mathbb{Z}/2\mathbb{Z}$ , but  $\mathbb{Z}/4\mathbb{Z} \not\cong \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$ .<sup>a</sup>

<sup>a</sup>The precise condition for this kind of “canceling” is called the Schur-Zassenhaus lemma.

**Example (Another explicit computation).** Let  $\phi : D_8 \rightarrow \mathbb{Z}_4$  be defined by

$$r \mapsto \bar{2}, \quad s \mapsto \bar{2}.$$

The kernel of this map is  $N = \{1, r^2, sr, sr^3\}$ . We can do a quick computation of all the elements of  $D_8$  to get

$$\phi(1) = \phi(r^2) = \phi(sr) = \phi(sr^3) = \bar{0} \text{ and } \phi(r) = \phi(r^3) = \phi(s) = \phi(sr^2) = \bar{2}.$$

The two relevant fibers are

$$\phi^{-1}(\bar{0}) = 1N = r^2N = srN = sr^3N = \{1, r^2, sr, sr^3\}$$

and

$$\phi^{-1}(\bar{2}) = rN = r^3N = sN = sr^2N = \{r, r^3, s, sr^2\}.$$

So we see that  $|D_8 / N| = 2$  is a **group** of **order** two, or  $\mathbb{Z}_2$ . Indeed, the image of  $\phi$  is

$$\{\bar{0}, \bar{2}\} \cong \mathbb{Z}_2.$$

**Problem.** Suppose  $G$  is **Abelian**. Why does it follow that any **subgroup** of  $G$  is **normal**?

Finally, here's some food for thought.

**Problem.** Suppose one has a **group presentation** for a **group**  $G$  that uses  $n$  **generators**. Can you write it as a **quotient** of the form  $F_n / N$ , where  $N$  is a **normal subgroup** of  $F_n$ ?

### B.4.5 (Digression) The First Isomorphism Theorem

One quick word about what other sources usually say. Most textbooks actually *define* **normal** using the  $ghg^{-1} \in H$  property. Then they define  $G/H$  for **normal**  $H$  in the way I did above, using the **coset** definition

$$(g_1H) \cdot (g_2H) = g_1g_2H.$$

Using purely algebraic manipulations (like I did) this is well-defined, and so now you have this group  $G/H$  or something.

**Remark.** The underlying **homomorphism** isn't mentioned at all, or is just mentioned in passing.

I think this is *incredibly dumb*. The **normal** condition looks like it gets pulled out of thin air and no one has any clue what's going on, because no one has any clue what a **normal subgroup** actually should look like.

Other sources like to also write the so-called **first isomorphism theorem**.<sup>3</sup> It goes like this.

**Theorem B.4.3** (First isomorphism theorem). Let  $\phi: G \rightarrow H$  be a **homomorphism**. Then  $G/\ker \phi$  is **isomorphic** to  $\phi(G)$ .

**Note.** To me, this is just a clumsier way of stating the same idea.

About the only merit this claim has is that if  $\phi$  is injective, then the image  $\phi(G)$  is an **isomorphic copy** of  $G$  inside the **group**  $H$  (try to see this directly). This is a pattern we'll often see in other branches of mathematics:

**Note.** Whenever we have an *injective structure-preserving map*, often the image of this map will be some "copy" of  $G$ .<sup>a</sup> In that sense an injective **homomorphism**  $\phi: G \hookrightarrow H$  is an *embedding* of  $G$  into  $H$ .

<sup>a</sup>Here "structure" refers to the **group** multiplication.

<sup>3</sup>There is a second and third isomorphism theorem. But four years after learning about them, I *still* don't know what they are. So I'm guessing they weren't very important.

# Bibliography

- [KL20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020. ISBN: 9781351133012. URL: <https://books.google.com/books?id=RsoOEAAAQBAJ>.