# COMS30023/ Cryptology

## Lecture Notes

François Dupressoir Chloe Martindale

2025/26

# Contents

# Lecture 0 – Introduction

Cryptology is old: ever since we started mistrusting each other, we've sought to hide secrets from each other—and that's the first thing cryptography does.

But *modern* cryptology does more: it is not only a set of tools to protect data and the information it contains (whether it is at rest, in transit, or even in use), but also a set of techniques that allow us to precisely analyse the actual security guarantees of cryptographic tools—both by establishing lower bounds on security (by proving that breaking security would imply solving some hard problem), and by establishing upper bounds on security (by studying generic attacks against constructions and against the hard problems they rely on).

The core principle of this analysis is known as *Kerckhoffs' Principle* [Ker83], which can be roughly summed up as:

> Design and analyse your system assuming that your opponents know it in detail.

In other words: whatever algorithm you come up with, its security must not rely on the fact that it is unknown. The only secret you should assume is the *cryptographic key*.

In this unit, we will see how this principle shapes the way in which the modern cryptographer:

1. defines security;
2. designs cryptographic schemes;
3. chooses cryptographic assumptions; and
4. analyses cryptographic security;

and in which the modern cryptanalyst:

1. breaks security in practice;
2. attempts to undermine cryptographic assumptions.

We will do so considering the simple case of two mutually trusting participants attempting to exchange a message over an insecure network.

# Lecture 1 – Enciphering Schemes From Perfect Security to Blockciphers

We first consider a simple setting: Aniket and Barbara want to securely exchange a single message whose length they know in advance. This setting gives rise to simple constructions—*enciphering schemes* that we use to introduce a number of basic and standard methods in modern cryptography.

First, we'll specify which algorithms can be considered enciphering schemes by defining their *syntax*. Then, we'll specify the most basic properties such enciphering schemes should possess: *correctness*. Finally, and most importantly, we will discuss what exactly it might mean for an enciphering scheme to be *secure*.

This will lead us to the modern practice of game-based (or property-based) definitions of security.

## 1.1. Enciphering Schemes: Syntax and Correctness

Informally, we are interested in taking a message in plain text—often referred to as the *plaintext*, taken from some *message space* $\mathcal{M}$—and some key—taken from some *key space* $\mathcal{K}$; and outputs an enciphered message—often referred to as *the ciphertext*—taken from some *cipher space* $\mathcal{C}$; in such a way that the original message can be recovered given knowledge of ciphertext and key, but not without the key.

An enciphering scheme (for us) is such that $\mathcal{M} = \mathcal{C}$, and is specified by three algorithms:

- A probabilistic algorithm that generates keys in $\mathcal{K}$;
- An algorithm that *enciphers* a plaintext under a key, and into a ciphertext; and
- An algorithm that *deciphers* a ciphertext under a key, and into a plaintext.

This exactly specifies the syntax of enciphering scheme. The formal definition (Definition 1.1) does a bit more legwork in introducing some notation, and giving names to those algorithms. This will later give us nice ways of abstracting over specific enciphering schemes.

**Definition 1.1** (Symmetric Enciphering Scheme)**.** A *symmetric enciphering scheme* $E$ is a triple of algorithms $\mathsf{Kg}$, $\mathsf{E}$, and $\mathsf{D}$, where:

- $\mathsf{Kg}$ randomly generates a $k \in \mathcal{K}$;
- $\mathsf{E}$ takes a key $k$ and a message $m \in \mathcal{M}$ to output ciphertext $c \leftarrow \mathsf{E}_k(m) \in \mathcal{C}$, with $\mathcal{C} = \mathcal{M}$; and

- D takes a key $k$ and a ciphertext $c \in \mathcal{C}$ and to output a purported message $m' \leftarrow \mathsf{D}_k(c)$.

With this definition in place, we can generically define what it means for an enciphering scheme to be *correct.*

**Definition 1.2** (Correctness of Enciphering Schemes)**.** An enciphering scheme $E = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$ is correct iff, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$, we have $\mathsf{D}_k(\mathsf{E}_k(m)) = m$.

## 1.2. The One-Time Pad

The one-time pad is a very simple enciphering scheme where keys, messages and ciphertexts are all bitstrings of some fixed length $\ell$. Figure 1.1 specifies its algorithms Kg, E and D for some $\ell > 0$. Note also the notation—which will become pervasive—for sampling a value $x$ uniformly at random in a (finite) set $S$: $x \leftarrow_\$ S$. (We will also use it to denote storing in a variable the result of running a probabilistic algorithm.) $\oplus$ is bitwise exclusive or (XOR).

```
Kg()
    $k \leftarrow_\$ \{0,1\}^\ell$
    return $k$
```

```
$\mathsf{E}_k(m)$
    $c \leftarrow m \oplus k$
    return $c$
```

```
$\mathsf{D}_k(c)$
    $m \leftarrow c \oplus k$
    return $m$
```

Figure 1.1.: The one-time pad; $\ell$ is the intended message length

## 1.3. Security of Enciphering Schemes

A natural question, then is: how much *security* does such a simple construction as the one-time pad provide? The answer, as we see next, is simultaneously ”it provides perfect security,” and ”it provides no security whatsoever”. The main crux of what looks like a paradox right now, is that we do not even know what it means for an enciphering scheme to be secure. Let’s remedy that.

### 1.3.1. Key Recovery

By Kerckhoffs’ principle, we must certainly ensure that the key cannot be recovered from the system—if an adversary can recover the key from a ciphertext—and is assumed to know all details of the algorithm used, then they can surely decipher that ciphertext as well.

**Adversary Goal** So we first attempt to define what it means for an enciphering scheme (*any* enciphering scheme) to be secure against key recovery. We do so using a *security experiment* (or *security game*), and defining an *adversarial advantage*—which essentially measures the *insecurity* of a scheme against an adversary.

```
$\mathsf{Exp}_E^{\text{kr-pas}}(\mathbb{A})$
$k^* \leftarrow_{\$} \mathsf{Kg}$
$\widehat{k} \leftarrow_{\$} \mathbb{A}$
```

```
$\mathbb{A}_{\text{guess}}$
$\widehat{k} \leftarrow_{\$} \mathsf{Kg}$
return $\widehat{k}$
```

Figure 1.2.: The passive key-recovery game $\mathsf{Exp}_E^{\text{kr-pas}}(\cdot)$, and the “guessing” adversary $\mathbb{A}_{\text{guess}}$ (right).

**Definition 1.3** (Passive Key Recovery Security for Enciphering Schemes)**.** Let $E$ be an enciphering scheme. The *advantage of* $\mathbb{A}$ *in passively recovering the key* is defined as follows, for the experiment $\mathsf{Exp}_E^{\text{kr-pas}}()$ defined in Figure 1.2.

$$\mathsf{Adv}_E^{\text{kr-pas}}(\mathbb{A}) \overset{def}{=} \Pr\left[\mathsf{Exp}_E^{\text{kr-pas}}(\mathbb{A}) : \widehat{k} = k^*\right]$$

An enciphering scheme $E$ is said to be $(t, \epsilon)$*-secure against passive key recovery* if, for every algorithm $\mathbb{A}$ running in time at most $t$, we have $\mathsf{Adv}_E^{\text{kr-pas}}(\mathbb{A}) \leq \epsilon$.

With this definition of security, it is clear that the adversary cannot do much: they get to see nothing that depends on the key, so the best they can do is guess. Such an adversary is given on the right hand side of Figure 1.2: they simply run the key generation algorithm, and hope it outputs the same key. If, say, $\mathsf{Kg}$ samples its output uniformly at random in the key space $\mathcal{K}$, then $\mathsf{Adv}_E^{\text{kr-pas}}(\mathbb{A}_{\text{guess}}) = 1/|\mathcal{K}|$.

**Adversarial powers** Clearly, beyond allowing us to introduce concepts and notations slowly, passive key recovery isn't very interesting as a security notion. Can the adversary recover the key when observing a ciphertext? We certainly hope not! But what other powers could we give the adversary?

Figure 1.3 shows three different experiments for key recovery under increasing adversary powers: (one-time) *known ciphertext attacks* (kr-1kca), (one-time) *known plaintext attack* (kr-1kpa), and (one-time) *chosen plaintext attack*. The shape of their advantage expression is determined entirely by the goal of key recovery: the only thing that changes is which game the adversary plays, but their winning condition is the same.

```
$\mathsf{Exp}_E^{\text{kr-1kca}}(\mathbb{A})$
$k^* \leftarrow_{\$} \mathsf{Kg}$
$m \leftarrow_{\$} \mathcal{M}$
$c \leftarrow \mathsf{E}_{k^*}(m)$
$\widehat{k} \leftarrow_{\$} \mathbb{A}(c)$
```

```
$\mathsf{Exp}_E^{\text{kr-1kpa}}(\mathbb{A})$
$k^* \leftarrow_{\$} \mathsf{Kg}$
$m \leftarrow_{\$} \mathcal{M}$
$c \leftarrow \mathsf{E}_{k^*}(m)$
$\widehat{k} \leftarrow_{\$} \mathbb{A}(m, c)$
```

```
$\mathsf{Exp}_E^{\text{kr-1cpa}}(\mathbb{A})$
$k^* \leftarrow_{\$} \mathsf{Kg}$
$m \leftarrow_{\$} \mathbb{A}$
$c \leftarrow \mathsf{E}_{k^*}(m)$
$\widehat{k} \leftarrow_{\$} \mathbb{A}(c)$
```

Figure 1.3.: Adding one-time powers to the key-recovery game in three different ways.

## 1.3.2. One-Wayness

The goal of enciphering is not to protect the key, but to protect the plaintext. How can we express that no adversary can reasonably do so? Let us first consider the notion of one-wayness, which captures the idea that recovering the plaintext *in full* from the ciphertext should be hard.

```
$\mathsf{Exp}_E^{\mathsf{ow\text{-}pas}}(\mathbb{A})$
$k \leftarrow_\$ \mathsf{Kg}$
$m^* \leftarrow_\$ \mathcal{M}$
$c^* \leftarrow \mathsf{E}_k(m^*)$
$\widehat{m} \leftarrow_\$ \mathbb{A}(c^*)$
```

Figure 1.4.: Passive one-time one-way security for symmetric enciphering scheme $E$

**Definition 1.4** (Passive One-Time One-Way Security for Enciphering Schemes)**.** Let $E$ be an enciphering scheme. We define the *advantage of* $\mathbb{A}$ *in passively breaking one-wayness* as follows, for the experiment $\mathsf{Exp}_E^{\mathsf{ow\text{-}pas}}()$ defined in Figure 1.4.

$$\mathsf{Adv}_E^{\mathsf{ow\text{-}pas}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}_E^{\mathsf{ow\text{-}pas}}(\mathbb{A}) : \widehat{m} = m^*\right]$$

An enciphering scheme $E$ is said to be $(t, \epsilon)$*-secure against passive one-wayness attacks* if, for every algorithm $\mathbb{A}$ running in time at most $t$, we have $\mathsf{Adv}_E^{\mathsf{ow\text{-}pas}}(\mathbb{A}) \leq \epsilon$.

## 1.3.3. Perfect Secrecy

Ensuring that no adversary is able to recover the message in full is nice. Ensuring that the adversary learns *no information* about the message at all is nicer.

Definition 1.5 captures this by expressing the fact that, whatever distribution the message is sampled from, the distribution over ciphertexts induced by enciphering a random plaintext under a freshly generated key is independent from the plaintext being enciphered.

**Definition 1.5** (Perfect Secrecy)**.** A symmetric enciphering scheme $E$ satisfies perfect secrecy if and only if for all message distributions over $\mathcal{M}$, the following holds.

$$\forall c \in \mathcal{C}, m \in \mathcal{M}.\, \Pr\left[m^* = m \mid c^* = c\right] = \Pr\left[m^* = m\right]$$

The probabilities are taken over $k \leftarrow_\$ \mathsf{Kg}$ and $m^* \leftarrow_\$ \mathcal{M}$ (according to the aforementioned message distribution), which fixes $c^* = \mathsf{E}_k(m^*)$.

**Security of the One-Time Pad** The One-Time Pad turns out to be perfectly secure.

**Theorem 1.1** (Security of the One-Time Pad)**.** *The One-Time Pad satisfies perfect security.*

**Shannon's Theorem** Unfortunately for us, the One-Time Pad turns out to be (up to isomorphism) the only enciphering scheme that is perfectly secure.

**Theorem 1.2** (Shannon's Theorem)**.** *Let $E = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$ be an enciphering scheme with $\mathcal{K} = \mathcal{M}$. Then $E$ is perfectly secure iff $\mathsf{Kg}$ draws from $\mathcal{K}$ uniformly at random and $E$ satisfies that for all $(m, c)$ pairs, there exists a unique key $k$ such that $\mathsf{E}_k(m) = c$.*

## 1.3.4. Indistinguishability

We want to weaken perfect secrecy a little bit so that more schemes satisfy the notion, but without weakening it so much as to make it meaningless. We've already seen a few notions that allowed a bit of sloppiness. Can we express perfect secrecy as a game, then loosen it a little bit?

There are a few equivalent ways of expressing perfect secrecy. That given in Definition 1.5 is the original one given by Shannon [Sha49]. However, it is not directly useful to express security as a game: it quantifies over the plaintext distribution, and it directly talks about the independence of some distribution.

Definition 1.6

**Definition 1.6** (Perfect Indistinguishability)**.** A symmetric enciphering scheme $E$ satisfies perfect indistinguishability if and only if the following holds.

$$\forall c \in \mathcal{C}, m \in \mathcal{M}.\, \Pr\left[c^* = c \mid m^* = m\right] = |\mathcal{C}|^{-1}$$

The probability is taken over $k \leftarrow_{\$} \mathsf{Kg}$.

**Theorem 1.3.** *An enciphering scheme has perfect secrecy if and only if it has perfect indistinguishability.*

We can express this property as a game—however, the adversary's goal here is no longer to *recover* or compute a value, but to distinguish two different experiments.

```
$\mathsf{Exp}_E^{\mathsf{1ind\text{-}real}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $m \leftarrow_{\$} \mathbb{A}$
    $c^* \leftarrow \mathsf{E}_k(m)$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}(c^*)$
```

```
$\mathsf{Exp}_E^{\mathsf{1ind\text{-}ideal}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $m \leftarrow_{\$} \mathbb{A}$
    $c^* \leftarrow_{\$} \mathcal{C}$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}(c^*)$
```

Figure 1.5.: One-time indistinguishability

**Definition 1.7** (Game-Based Perfect Indistinguishability)**.** Let $E$ be an enciphering scheme. We define the *advantage of* $\mathbb{A}$ *in one-time distinguishing* $E$ *from random* as follows, for the experiments $\mathsf{Exp}_E^{\mathsf{1ind\text{-}real}}()$ and $\mathsf{Exp}_E^{\mathsf{1ind\text{-}ideal}}()$ defined in Figure 1.5.

$$\mathsf{Adv}_E^{\mathsf{1ind}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}_E^{\mathsf{1ind\text{-}real}}(\mathbb{A}) : \widehat{b} = 1\right] - \Pr\left[\mathsf{Exp}_E^{\mathsf{1ind\text{-}ideal}}(\mathbb{A}) : \widehat{b} = 1\right]$$

An enciphering scheme $E$ is said to be *perfectly indistinguishable* if, for every algorithm $\mathbb{A}$, we have $\mathsf{Adv}_E^{\mathsf{1ind}}(\mathbb{A}) = 0$.

Now *this* definition can effectively be weakened in two ways: first, we can—instead of considering all possible algorithms—only consider adversaries with bounded resources, as we did for key recovery and one-wayness; and second, we can—instead of requiring that the adversary's advantage be 0—consider a scheme secure if any (bounded) adversary's advantage is small.

**Definition 1.8** (Game-Based Indistinguishability)**.** An enciphering scheme $E$ is said to be $(t, \epsilon)$*-indistinguishable* if, for every algorithm $\mathbb{A}$ that runs in time at most $t$, we have $\mathsf{Adv}_E^{\mathsf{1ind}}(\mathbb{A}) \leq \epsilon$.

This will be our baseline security notion for confidentiality in the rest of this unit, and serves as the (heuristic) assumption the entirety of practical cryptography relies on: that *blockciphers* are indistinguishable from random permutations.

# 1.4. Blockciphers

To do anything worth while, we need to allow ourselves to define security when the same key can be used multiple times—recall that we've so far only defined notions under one-time attacks! We take the smallest possible step in this direction by defining blockciphers.

**Definition 1.9** (Blockcipher)**.** A *blockcipher* $E$ with *block length* $\ell$ is a symmetric enciphering scheme with $\mathcal{M} = \mathcal{C} = \{0,1\}^\ell$.

**Lemma 1.4** (Blockciphers as Keyed Permutations)**.** *Let* $E = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$ *be a* correct *blockcipher. Then, for any fixed key k output by* $\mathsf{Kg}$*, the enciphering function* $\mathsf{E}_k$ *is a permutation.*

*Proof.* Left as an exercise to the reader, recalling the definition of correctness for enciphering schemes. □

We do not discuss the construction (or *realisation*) of blockciphers in this unit. The second (optional) half of the worksheet explores this question in depth—mostly negatively, to show that designing a good blockcipher is hard and that you are strongly encouraged to show humility if you try. But let us consider instead what kind of security we might want, how we can define it, and the boringest ways in which we can break it—this will inform some design constraints.

## 1.4.1. Security: Key Recovery

Let us first revisit key recovery under chosen plaintext attacks. Unlike previously—where we were considering one-time security notions, we now want to allow the adversary to interact with the key *multiple times*—but we still can't give the adversary they key!

The solution here is to consider adversaries that have *oracle access* to the encryption algorithm with a fixed key: this controls the way in which the adversary is allowed to make

```
$\mathsf{Exp}_E^{\text{kr-cpa}}(\mathbb{A})$
$k^* \leftarrow_\$ \mathsf{Kg}$
$\widehat{k} \leftarrow_\$ \mathbb{A}^{\mathcal{E}(\cdot)}$

    $\mathcal{E}\ (m)$
    $c \leftarrow \mathsf{E}_{k^*}(m)$
    return $c$
```

```
$\mathbb{A}_{\text{search}}$
$Ks \leftarrow \emptyset$
for $k \in \mathcal{K}$
    if $\mathsf{E}_k(m) = c$
    then $Ks \leftarrow Ks \cup \{k\}$
$\widehat{k} \leftarrow_\$ Ks$
return $\widehat{k}$
```

Figure 1.6.: The “key-recovery under chosen plaintext attack game” (left) and the “exhaustive search” adversary that uses a single known-plaintext-ciphertext pair (right).

use of the key in a minimally intrusive way. In particular, unless a specific note is made otherwise, the adversary can choose their queries to their oracles *adaptively*: make a query, see the result, *then* choose the next query.

**Definition 1.10** (Key Recovery Security for Blockciphers)**.** Let $E$ be a blockcipher. We define the *advantage of* $\mathbb{A}$ *in recovering the key from* $E$ *under chosen plaintext attack* as follows, where experiment $\mathsf{Exp}_E^{\text{kr-cpa}}(\mathbb{A})$ is defined in Figure 1.6.

$$\mathsf{Adv}_E^{\text{kr-cpa}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}_E^{\text{kr-cpa}}(\mathbb{A}) : \widehat{k} = k^*\right]$$

$E$ is said to be $(t, q, \epsilon)$*-secure against chosen plaintext key recovery* if, for every algorithm $\mathbb{A}$ running in time at most $t$ and making at most $q$ queries to its chosen plaintext oracle $\mathcal{E}(\cdot)$, we have $\mathsf{Adv}_E^{\text{kr-cpa}}(\mathbb{A}) \leq \epsilon$.

**Exhaustive search as baseline security level.** A simple (but costly) attack, given one or several plaintext-ciphertext pairs, is to simply iterate through all the keys and eliminate those that fail to map the plaintexts to the corresponding ciphertexts.

The number of encipherings it takes to run an exhaustive search (or rather, its base 2) is often used as a baseline for the security of a blockcipher. When a blockcipher’s actual key recovery security strays a bit too far from this then the blockcipher is considered broken. This gives somewhat uniform measures for all notions of security: ”we want 256-bit security” translates to ”we want breaking whatever security we just asked you to obtain to be as costly as running $2^{256}$ encipherings of something”. However, it’s a lot less uniform in practice than we’d like—as a science—to pretend.

Current recommendations: if you really can’t do anything else, 112 bit security is OK (lightweight cryptography); if you don’t really care about the data long-term but need to show you did something short-term, 128 bit security is what you want; if you care about the data long-term, you must aim for 256 bit security.

```
$\mathsf{Exp}_E^{\text{ind-real}}(\mathbb{A})$
    $k^* \leftarrow_\$ \mathsf{Kg}$
    $\widehat{b} \leftarrow_\$ \mathbb{A}^{\mathcal{E}(\cdot)}$

    $\mathcal{E}(m)$
        no repeat queries
        $c \leftarrow \mathsf{E}_{k^*}(m)$
        return $c$
```

```
$\mathsf{Exp}_E^{\text{ind-ideal}}(\mathbb{A})$
    $\widehat{b} \leftarrow_\$ \mathbb{A}^{\mathcal{E}(\cdot)}$

    $\mathcal{E}(m)$
        no repeat queries
        $c \leftarrow_\$ \mathcal{C}$
        return $c$
```

Figure 1.7.: The real and ideal indistinguishability experiments.

## 1.4.2. Pseudorandomness

As before, key recovery is a nice baseline, but what we want is *indistinguishability*. For blockciphers, for some reason, it's called pseudorandomness.

**Definition 1.11** (Pseudorandom Permutation)**.** Let $E$ be a blockcipher. We define the *advantage of $\mathbb{A}$ in distinguishing $E$ from a random permutation* as follows, where experiments $\mathsf{Exp}_E^{\text{ind-real}}(\mathbb{A})$ and $\mathsf{Exp}_E^{\text{ind-ideal}}(\mathbb{A})$ are defined in Figure 1.7.

$$\mathsf{Adv}_E^{\text{ind}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}_E^{\text{ind-real}}(\mathbb{A}) : \widehat{b} = 1\right] - \Pr\left[\mathsf{Exp}_E^{\text{ind-ideal}}(\mathbb{A}) : \widehat{b} = 1\right]$$

$E$ is said to be a $(t, q, \epsilon)$*-secure pseudorandom permutation* if, for every algorithm $\mathbb{A}$ running in time at most $t$ and making at most $q$ queries to its $\mathcal{E}(\cdot)$ oracle, we have $\mathsf{Adv}_E^{\text{ind}}(\mathbb{A}) \leq \epsilon$.

Note here that we *must* restrict the adversary from querying the same input twice to the $\mathcal{E}$ oracle: in the real world, they would get the same response to both identical queries; in the ideal world, they would only get the same response with low probability—a clear and trivial distinguishing attack!

**The birthday bound.** What we cannot rule out immediately is repeat responses: those never happen in the real world, but could happen in the ideal world. This is known as a *collision*—two messages $m \neq m'$ such that $\mathcal{E}(m) = \mathcal{E}(m')$. An adversary that makes $q$ queries to a random permutation will find such a collision with probability roughly $\frac{q \cdot (q-1)}{2 \cdot |\mathcal{C}|}$ (the birthday bound).

If exhaustive search placed a constraint on key size, the birthday bound places a constraint on the block length $\ell$ of a blockcipher if we are hoping for it to be pseudorandom.

# Lecture 2 – Symmetric Encryption

We've seen how to build enciphering schemes that are perfectly secure. We've also seen that "perfectly secure" means both "insecure in practice" and "impractical" (a happy combination if you think about it!). We then considered another way of defining security for enciphering schemes, which allows keys to be reused and weaken the "perfect" requirement into a "computational" requirement, moving from "it should be impossible for an adversary to break this" to "it should be infeasible for a reasonable adversary to break this".

Let's now see how we can construct practical *encryption schemes* from those *blockciphers*, and how we can reason about the fact that the construction does not weaken security too much. As before, we'll define things somewhat formally, consider generic attacks to figure out the best we can hope for and define our objectives, then we'll get to work.

## 2.1. Nonce-Based Encryption

We have a building block which allows us to use a single, relatively short key, to encrypt multiple messages—as long as they fit in a block and are never repeated. We now take our final step towards the construction of an encryption primitive: we *use* blockciphers in structured ways to *encrypt* long messages while allowing repetitions. We do so by instead requiring that some public value called a *nonce* (number used only once) never be repeated instead—this is safer because the nonce can be entirely controlled by the cryptographic layer above, whereas plaintexts come from strange and unknown distributions—you might, for example, be hard-pressed to send three distinct messages from the set $\{Yes, No\}$.

**Definition 2.1** (Nonce-Based Encryption Scheme)**.** A *nonce-based encryption scheme* $E$ is a triple of algorithms $(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, where $\mathsf{Kg}$ randomly generates a key $k \in \mathcal{K}$, $\mathsf{Enc}$ takes a key $k$, a nonce $n \in \mathcal{N}$, and a message $m \in \mathcal{M}$ to output ciphertext $c \leftarrow \mathsf{Enc}_k^n(m) \in \mathcal{C}$, and $\mathsf{Dec}$ takes a nonce $n$, a ciphertext $c \in \mathcal{C}$ and key $k$ to output a purported message $m' \leftarrow \mathsf{Dec}_k^n(c)$.

$E$ is said to be *correct* iff, for all $k \leftarrow_\$ \mathsf{Kg}$, $n \in \mathcal{N}$, and $m \in \mathcal{M}$, $\mathsf{Dec}_k^n(\mathsf{Enc}_k^n(m)) = m$.

**Definition 2.2** (Nonce-Based Indistinguishability)**.** Let $E$ be a nonce-based encryption scheme. We define the *advantage of* $\mathbb{A}$ *in distinguishing* $E$ *from random ciphertexts* as follows, where experiments $\mathsf{Exp}_E^{\mathsf{(n)ind\text{-}real}}(\mathbb{A})$ and $\mathsf{Exp}_E^{\mathsf{(n)ind\text{-}ideal}}(\mathbb{A})$ are defined in Figure 2.1.

$$\mathsf{Adv}_E^{\mathsf{(n)ind}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}_E^{\mathsf{(n)ind\text{-}real}}(\mathbb{A}) : \widehat{b} = 1\right] - \Pr\left[\mathsf{Exp}_E^{\mathsf{(n)ind\text{-}ideal}}(\mathbb{A}) : \widehat{b} = 1\right]$$

$E$ is said to be a $(t, q, \epsilon)$*-indistinguishable nonce-based encryption scheme* if, for every algorithm $\mathbb{A}$ running in time at most $t$ and making at most $q$ queries to its CPA oracle $\mathcal{E}(\cdot,\cdot)$, we have $\mathsf{Adv}_E^{\mathsf{(n)ind}}(\mathbb{A}) \leq \epsilon$.

```
$\mathsf{Exp}_E^{(n)\text{ind-real}}(\mathbb{A})$
  $k^* \leftarrow_\$ \mathsf{Kg}$
  $\widehat{b} \leftarrow_\$ \mathbb{A}^{\mathcal{E}(\cdot,\cdot)}$

  $\mathcal{E}(n, m)$
    no repeat nonces
    $c \leftarrow \mathsf{Enc}_{k^*}^n(m)$
    return $c$
```

```
$\mathsf{Exp}_E^{(n)\text{ind-ideal}}(\mathbb{A})$
  $\widehat{b} \leftarrow_\$ \mathbb{A}^{\mathcal{E}(\cdot,\cdot)}$

  $\mathcal{E}(n, m)$
    no repeat nonces
    $c \leftarrow_\$ \mathcal{C}(|m|)$
    return $c$
```

Figure 2.1.: The real and ideal nonce-based indistinguishability experiments.

```
$\mathsf{Enc}_k^n(m)$
  Assume block length $\ell$ and $m \in \{0,1\}^{\ell \cdot n}$, $\ell < 2^{\ell/2}$, $n \in \{0,1\}^{\ell/2}$
  $(m[1], \ldots, m[n]) \leftarrow \mathsf{parse}(m)$
  for $i \in \{1, \ldots, n\}$
    $X[i] \leftarrow n \| \langle i \rangle_{\ell/2}$
    $Y[i] \leftarrow \mathsf{E}_k(X[i])$
    $c[i] \leftarrow m[i] \oplus Y[i]$
  $c \leftarrow c[1] \| \ldots \| c[n]$
  return $c$
```

```
$\mathsf{Dec}_k^n(c)$
  Assume block length $\ell$ and $c \in \{0,1\}^{\ell \cdot n}$, $\ell < 2^{\ell/2}$, $n \in \{0,1\}^{\ell/2}$
  $(c[1], \ldots, c[n]) \leftarrow \mathsf{parse}(c)$
  for $i \in \{1, \ldots, n\}$
    $X[i] \leftarrow n \| \langle i \rangle_{\ell/2}$
    $Y[i] \leftarrow \mathsf{E}_k(X[i])$
    $m[i] \leftarrow c[i] \oplus Y[i]$
  $m \leftarrow m[1] \| \ldots \| m[n]$
  return $m$
```

Figure 2.2.: Nonce-Based Counter Mode (CTR) over a blockcipher $E = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$; key generation is that of the blockcipher

### 2.1.1. Modes of Operation

All that is left for us to do (before we can prove something useful) is to *generically* build nonce-based encryption from any blockcipher. This is done using a *mode of operation.*

Counter mode (or CTR), shown in Figure 2.2, is the most basic mode of operation given what we've already seen: use the blockcipher to expand the nonce into as many blocks of pseudorandom bits as needed, then use those as a one-time pad on the message.

It is nonce-based indistinguishable from random as long as the blockcipher it is constructed upon is pseudorandom. After a quick aside, we'll consider how to prove this.

**Other Modes of Operation** Other modes of operation exist. Some should not be used (Electronic Codebook, or ECB), some are so secure we can't even talk about their security until later in the unit (GCM, OCB), others yet are not nonce-based secure, but are secure under some additional conditions on the nonce.

The most notorious of these—for having been used in TLS, and for being used in SSH—is Cipher Block Chaining mode (CBC), which is shown in Figure 2.3. We discuss it a bit in the

```
$\mathsf{Enc}^n_k(m)$
Require $m \in \{0,1\}^{\ell \cdot n}$ and $n \in \{0,1\}^{\ell}$
- - - - - - - - - - - - - - - - - - - - - - - -
$(m[1], \ldots, m[n]) \leftarrow \mathsf{parse}(m)$
$c[0] \leftarrow n$
for $i \in [1, \ldots, n]$
    $X[i] \leftarrow m[i] \oplus c[i-1]$
    $c[i] \leftarrow \mathsf{E}_k(X[i])$
$c \leftarrow c[1] \| \ldots \| c[n]$
return $c$
```

```
$\mathsf{Dec}^n_k(c)$
Require $c \in \{0,1\}^{\ell \cdot n}$ and $n \in \{0,1\}^{\ell}$
- - - - - - - - - - - - - - - - - - - - - - - -
$(c[1], \ldots, c[n]) \leftarrow \mathsf{parse}(c)$
$c[0] \leftarrow n$
for $i \in [1, \ldots, n]$
    $X[i] \leftarrow \mathsf{D}_k(c[i])$
    $m[i] \leftarrow c[i-1] \oplus X[i]$
$m \leftarrow m[1] \| \ldots \| m[n]$
return $m$
```

Figure 2.3.: Cipher Block Chaining Mode (CBC) over a blockcipher $E = (\mathsf{Kg}, \mathsf{E}, \mathsf{D})$; key generation is that of the blockcipher

$$\begin{tikzcd}
 & & & \text{OW-PAS} \arrow[dl] \\
\text{KR-1CPA} \arrow[r] & \text{KR-1KPA} \arrow[r] & \text{KR-1KCA} \arrow[r] & \text{KR-PAS}
\end{tikzcd}$$

Figure 2.4.: Relations between one-time security notions; arrows correspond to security implications (omitting those obtained by transitivity).

problem sheet—mostly, again, destructively.

All those modes of operation—and more!—have their performance and use case advantages and drawbacks. Exploring all of them is not particularly useful for generalist cryptographers, although knowing that the variety exists is.

## 2.1.2. Reductions

Let's get back to CTR: how do we prove the earlier statement we made about its security—that if a blockcipher $E$ is pseudorandom, then CTR over $E$ is nonce-based secure?

Let's consider again the statement "if 'A' is secure against this, then 'B' is secure against that." Logically, this is equivalent to "if 'B' is not secure against that, then 'A' is not secure against this." By definition, not being secure corresponds to the existence of a successful adversary, so we can restate to "if there is a successful that-adversary against 'B', then there is a successful this-adversary against 'A'." Finally, we have arrived at a statement we can deal with constructively. We will assume the existence of some $\mathbb{A}_{\text{b,that}}$ and use it to construct an adversary $\mathbb{B}_{\text{a,this}}$ where we can relate the respective adversarial advantages.

We won't prove CTR secure just yet, but we'll illustrate the concept of a reduction by proving some relations between the one-time security notions we came across in Lecture 1. The relevant notions and their relations are summarized in Figure 2.4.

**From strong to weak powers.** To start, we keep the security goal the same and look at what happens when the adversary gets more or less power. This corresponds to the ho-

```
$\mathsf{Exp}^{\text{kr-1cpa}}_{E}(\mathbb{B}_{\text{kr-1cpa}})$
$k^* \leftarrow_\$ \mathsf{Kg}$
$\hat{k} \leftarrow_\$$
    $\mathbb{B}^{\mathcal{E}_{\text{cpa}}(\cdot)}_{\text{kr-1cpa}}$
    $\hat{k} \leftarrow_\$ \mathbb{A}^{\mathcal{E}_{\text{kpa}}()}_{\text{kr-1kpa}}$
    return $\hat{k}$

        $\mathcal{E}_{\text{kpa}}()$
        $m \leftarrow_\$ \mathcal{M}$
        $c \leftarrow \mathcal{E}_{\text{cpa}}(m)$
        return $(m, c)$

    $\mathcal{E}_{\text{cpa}}(m)$
    $c \leftarrow \mathsf{E}_{k^*}(m)$
    return $c$
```

Figure 2.5.: A reduction for KR-1CPA ⇒ KR-1KPA.

rizontal implications in Figure 2.4. Intuitively, more power should help an adversary, so security against "stronger" adversaries should imply security against "weaker" adversaries.

For this example, we will construct a reduction showing that $(t, \epsilon)$-KR-1CPA security implies $(t, \epsilon)$-KR-1KPA-security.[1] Recall the logic—so we can recall what we assume, and what we construct.

i. If $E$ is $(t, \epsilon)$-KR-1CPA secure then it is $(t, \epsilon)$-KR-1KPA secure.

ii. If $E$ is $(t, \epsilon)$-KR-1KPA *insecure* then it is $(t, \epsilon)$-KR-1CPA *insecure*

iii. If there exists a KR-1KPA adversary against $E$ that runs in time at most $t$ and wins with an advantage greater than $\epsilon$, then there exists a KR-1CPAadversary against it that runs in time at most $t$ and wins with an advantage greater than $\epsilon$.

Thus, given a KR-1KPA adversary $\mathbb{A}_{\text{kr-1kpa}}$, we need to construct a KR-1CPA adversary $\mathbb{B}_{\text{kr-1cpa}}$ in such a way that:

1. $\mathbb{B}_{\text{kr-1cpa}}$ is (roughly) as efficient as $\mathbb{A}_{\text{kr-1kpa}}$; and
2. $\mathsf{Adv}^{\text{kr-1kpa}}_{E}(\mathbb{A}_{\text{kr-1kpa}}) \leq \mathsf{Adv}^{\text{kr-1cpa}}_{E}(\mathbb{B}_{\text{kr-1cpa}})$.

Here $\mathbb{B}_{\text{kr-1cpa}}$ is called the *reduction*, and the two claims relating to efficiency and advantage are the *analysis* of the reduction. Figure 2.5 shows the reduction (in the dashed box headed $\mathbb{B}_{\text{kr-1cpa}}$). We are now left to analyse its efficiency and advantage.

[1] Note the same $t$ and $\epsilon$; this is happy land.

```
$\mathsf{Exp}_E^{\text{ow-pas}}(\mathbb{B}_{\text{ow-pas}})$
$k^* \leftarrow_\$ \mathsf{Kg}$
$m^* \leftarrow_\$ \mathcal{M}$
$c^* \leftarrow \mathsf{E}_{k^*}(m^*)$
$\widehat{m} \leftarrow_\$$
    $\mathbb{B}_{\text{ow-pas}}(c^*)$
    $\widehat{k} \leftarrow_\$ \mathbb{A}_{\text{kr-1kca}}^{\mathcal{E}_{\text{kca}}}$
    $\widehat{m} \leftarrow \mathsf{D}_{\widehat{k}}(c^*)$
    return $\widehat{m}$

        $\mathcal{E}_{\text{kca}}$
        return $c^*$
```

Figure 2.6.: Reduction for OW-PAS ⇒ KR-1KCA.

- $\mathbb{B}_{\text{kr-1cpa}}$ runs $\mathbb{A}_{\text{kr-1kpa}}$ once, with the only overhead being that of sampling a message when $\mathbb{A}_{\text{kr-1kpa}}$ makes an oracle query—so $\mathbb{B}_{\text{kr-1cpa}}$ runs (roughly) in time $t$ if $\mathbb{A}_{\text{kr-1kpa}}$ runs in time $t$;

- $\mathbb{B}_{\text{kr-1cpa}}$ and $\mathbb{A}_{\text{kr-1kpa}}$ are facing experiments with the same challenge key $k^*$, and share also their key guess, so whenever one wins, the other does as well, and we have

$$\mathsf{Adv}_E^{\text{kr-1kpa}}(\mathbb{A}_{\text{kr-1kpa}}) = \mathsf{Adv}_E^{\text{kr-1cpa}}(\mathbb{B}_{\text{kr-1cpa}})$$

**From hard to easy goals.** The same way adversary capabilities can be ranked, there is a hierarchy of goals as well. Typically,[2] indistinguishability notions are strongest, key recovery is weakest, and one-wayness sits in the middle. We will prove that $(t, \epsilon)$-OW-PAS security implies $(t, \epsilon)$-KR-1KCA security using a reduction.

First, we figure out the logic of the reduction. We are given an adversary $\mathbb{A}_{\text{kr-1kca}}$ and need to create a reduction $\mathbb{B}_{\text{ow-pas}}$. The 'skin' of the reduction is shown in the left pane of Figure 2.6: $\mathbb{B}_{\text{ow-pas}}$ must simulate $\mathbb{A}_{\text{kr-1kca}}$'s one-time KCA oracle (which gives it a valid ciphertext under the challenge key), and must somehow turn $\mathbb{A}_{\text{kr-1kca}}$'s output—a key guess—into a message guess.

The overall reduction is shown in Figure 2.6, and we can analyse it. Again, the running time of $\mathbb{B}_{\text{ow-pas}}$ is essentially that of $\mathbb{A}_{\text{kr-1kca}}$ as the overhead is minimal. Whenever $\mathbb{A}_{\text{kr-1kca}}$ wins by outputting the correct key, then $\mathbb{B}_{\text{ow-pas}}$ is guaranteed to win as well. Additionally, $\mathbb{B}_{\text{ow-pas}}$ might end up lucky even if $\mathbb{A}_{\text{kr-1kca}}$ doesn't return the correct key, so we have

$$\mathsf{Adv}_E^{\text{kr-1kca}}(\mathbb{A}_{\text{kr-1kca}}) \leq \mathsf{Adv}_E^{\text{ow-pas}}(\mathbb{B}_{\text{ow-pas}})$$

This is exactly what we needed to prove.

[2] of those we discuss in this unit

# Lecture 3 – Maths Background and Diffie-Hellman

In previous lectures we have seen how to set up secure communication *given a shared secret value.* In the next lecture, we will see how to make this communication more efficient using block ciphers. In this lecture, we'll consider how to distribute this secret value using mathematics. In the modern world, you are attempting to communicate securely with many different parties: servers on the other side of the world, family in another country, companies, governments, hospitals, the list goes on. So how can we use mathematics to share a secret value cheaply, easily, and without ever meeting? This is the premise of *public key cryptography.* This lecture will build up the mathematical foundations necessary to understand some ways in which this is done in practice und introduce the Diffie Hellman key exchange protocol.

## 3.1. Modular arithmetic

*Arithmetic* should be thought of as the basic mathematical operations such as addition, subtraction, multiplication, and division. This is something with which you are very familiar with in the sets $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$, but there are many more ways of constructing sets of numbers on which there exist consistent arithmetic laws. The arithmetic of a clock is especially interesting because we can construct a consistent set of arithmetic laws on the *finite* set of hours.

Let us start by studying 'clock addition'. If you add 4 hours to 10 o'clock, then you get 2 o'clock (rather than 14 o'clock, since we will work here with the 12-hour clock). The way that we will write this is:

$$10 + 4 \equiv 2 \pmod{12}.$$

The $\equiv$ sign should be read as 'is equivalent to', and the notation $\pmod{12}$ tells us that we should reset when we get to the number 12, or if you like that our day is split into 12 hours.

'Clock subtraction' works in much the same way. If you subtract 6 hours from 1 o'clock, then you get 7 o'clock. The way that we will write this is:

$$1 - 6 \equiv 7 \pmod{12}.$$

'Clock multiplication' can be thought of just as repeated addition, so can as be defined in a natural way. For example,

$$5 \times 3 = 5 + 5 + 5 \equiv 3 \pmod{12}.$$

Division is a little more complicated, so we will return to that later.

A natural question that arises when studying clock arithmetic is: what if the day was not split into sets of 12 hours, but some other number, like 7? We can of course set up addition, subtraction, and multiplication $\pmod 7$ in just the same way as $\pmod{12}$. Formally, we define the notation $\equiv$ and $\pmod n$ as follows.

**Definition 3.1.** Let $n \in \mathbb{Z}_{>1}$ and let $a, b \in \mathbb{Z}$. We say that

$$a \equiv b \pmod n$$

if there exists $k \in \mathbb{Z}$ such that $a = b + kn$.

We refer to basic arithmetic $\pmod n$ as *modular arithmetic*. Suppose now that we want to compute $10 \times 11 \pmod{12}$. We would like to find $a \in \mathbb{Z}$ such that $1 \leq a \leq 12$ and $10 \times 11 \equiv a \pmod{12}$. One way to do this is to first compute $10 \times 11 = 110$, and then divide 110 by 12 and take $a$ to be the remainder. Try to prove for yourself that this will give the right answer.

Finally, let us turn to division. Suppose that you want to divide 3 by 4 on our 7-hour clock. It turns out that the best way to think of this is as $3 \times 4^{-1}$– we already have a notion of multiplication (and of 3), so it remains to understand the notion of inverses[1]:

**Definition 3.2.** Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}_{>1}$. If there exists $b \in \mathbb{Z}$ such that

$$ab \equiv 1 \pmod n$$

then we say that $b \pmod n$ is the *inverse* of $a \pmod n$.

Notice the 'if there exists' part of this definition. Consider $a = n = 12$. No matter how many multiples of 12 you take, you are always going to land back at the 12 o'clock position on the clock, or more formally, for every $b \in \mathbb{Z}$ we have that $12b \equiv 12 \pmod{12}$, so in particular 12 has no inverse mod 12. In fact, since $12 \equiv 0 \pmod{12}$, this isn't so surprising, since we are used to the idea of 0 having no inverse. There are however other numbers by which we cannot divide $\pmod{12}$. Consider $a = 6$ and $n = 12$. For every $b \in \mathbb{Z}$ we have that either $6b \equiv 6 \pmod{12}$ or $6b \equiv 0 \pmod{12}$. So $6 \pmod{12}$ also has no inverse. When does an integer mod $n$ have an inverse?

To understand when the inverse exists, we first need to understand in which situations the inverse of $a \pmod b$ exist for any $a$ and $b$. Let's look at a couple of examples.

**Examples**

- The inverse of $4 \pmod 7$ is $2 \pmod 7$ because $4 \cdot 2 \pmod 7 \equiv 1 \pmod 7$.
- $4 \pmod 8$ has no inverse because for every $n \in \mathbb{Z}$ we know that

  $$4 \cdot n \pmod 8 \in \{0 \pmod 8, 4 \pmod 8\},$$

  so in particular there does not exist any $n \pmod 8$ such that $4 \cdot n \equiv 1 \pmod 8$.

[1] Technically we are introducing *multiplicative* inverses here (analogous to the additive inverse that we saw in Worksheet 1). As multiplicative inverses are clearly more interesting that additive inverses, we tend to drop the adjective.

- Exercise: generalise the above example. That is, show that if $m$ and $n$ are not coprime then $m$ does not have an inverse mod $n$.

In fact, the above exercise is also true in the reverse. That is, $a \pmod b$ is invertible if and only if $a$ and $b$ are coprime. The exercise above gives the 'only if', but what about the 'if'? For this we need *Euclid's algorithm.*[2]

## 3.2. Euclid's algorithm

**Algorithm 1 [Euclid's Algorithm]**

```
Require: $a$ and $b \in \mathbb{Z}_{>0}$; without loss of generality suppose that $a \geq b$.
Ensure: $d = \gcd(a, b)$.
Set $r_0 = a$, $r_1 = b$, and $i = 1$.
while $r_i \neq 0$ do
    $i \leftarrow i + 1$.
    Compute[a] the unique $m_i$ and $r_i \in \mathbb{Z}$ such that $0 \leq r_i < r_{i-1}$ and
        $r_{i-2} = m_i \cdot r_{i-1} + r_i.$
return $r_i$
```

[a] This is called *division-with-remainder.*

Euclid's algorithm returns the greatest common divisor (gcd) of the input numbers $a$ and $b$; if $\gcd(a, b) = 1$, then $b$ is invertible mod $a$. Moreover, from Euclid's algorithm, we can recover the inverse. This is stated and proven in the following corollary.

**Corollary 3.1** (Euclid's corollary[3]). *Let $a$ and $b$ be integers. If $d = \gcd(a, b)$ then there exist $m, n \in \mathbb{Z}$ such that*

$$am + bn = d.$$

*Proof.* This follows from Euclid's algorithm just by solving the series

$$\{r_{i-2} = m_i \cdot r_{i-1} + r_i\}_{2 \leq i \leq k}$$

of simultaneous equations occuring in Euclid's algorithm for $r_0 = a$, $r_1 = b$, and $r_k = d$. □

Now we have a new method to compute the inverse of $a$ mod $b$, as long as $a$ and $b$ are coprime: Use Euclid's algorithm to compute $m$ and $n$ such that $am + bn = 1$. Then, modulo $b$, we have

$$am \equiv 1 \pmod b,$$

or in other words $m$ is the inverse of $a$ mod $b$.

[2] Most likely not due to Euclid, but Euclid wrote about it.

[3] Often referred to just as Euclid's algorithm.

**Example.**
Let's see an example of how to use Euclid's algorithm to compute an inverse. Suppose that you want to compute the inverse of $11 \pmod{17}$.

Run Euclid's algorithm with $r_0 = 17$ and $r_1 = 11$:

$$\begin{aligned} r_0 &= 17 \\ r_1 &= 11 \\ r_2 &= 17 - 1 \cdot 11 = 6 \\ r_3 &= 11 - 1 \cdot 6 = 5 \\ r_4 &= 6 - 1 \cdot 5 = 1. \end{aligned}$$

Then reverse engineer the algorithm to get:

$$1 = r_4 = r_2 - r_3 = (r_0 - r_1) - (r_1 - r_2) = r_0 - 2r_1 + r_2 = 2r_0 - 3r_1.$$

In other words,

$$2 \cdot 17 - 3 \cdot 11 = 1,$$

so in particular

$$-3 \cdot 11 \equiv 1 \pmod{17},$$

so the inverse of $11 \pmod{17}$ is $-3 \equiv 14 \pmod{17}$.

## 3.3. Sun-Tzu's Remainder Theorem

There are many surprising constructions and consequences of modular arithmetic, and we now present a seminal theorem in modular arithmetic which turns out to be a key tool in cryptanalysis.

**Theorem 3.2** (Sun-Tzu's Remainder Theorem (SRT)[4]). *Given coprime $n, m \in \mathbb{Z}_{>1}$ and $a, b \in \mathbb{Z}$ there exist $c, d \in \mathbb{Z}$ such that*

$$cm + dn = 1 \tag{3.1}$$

*and*

$$x = bcm + adn \pmod{mn}$$

*is the only number $\pmod{mn}$ such that both*

$$x \equiv a \pmod{m} \quad \textit{and} \quad x \equiv b \pmod{n}.$$

You may have seen this before in a basic number theory course or a group theory course for example: with some mathematical machinery it is quick to prove. We won't prove uniqueness now but we will check that the given construction is valid.

[4] Most textbooks refer to this as the 'Chinese Remainder Theorem'. It is most likely not due to Sun-Tzu, but Sun-Tzu wrote about it.

*Proof of existence (constructive).* As $\gcd(n, m) = 1$, by Euclid's algorithm there exist $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1. \tag{3.2}$$

We claim that

$$x = bcm + adn \pmod{mn}$$

will work. We first check mod $n$. Note that $cm = 1 - dn$ by (3.2). So

$$x = b(1 - dn) + adn \equiv b \pmod{n}.$$

Similarly

$$x = bcm + a(1 - cm) \equiv a \pmod{m}.$$

□

**Example**
Now suppose that you are given the equations

$$x \equiv 4 \pmod{17}$$

and

$$x \equiv 3 \pmod{11}$$

and you want to find the $x \pmod{17 \cdot 11}$ that reduces mod 17 and 11 to these values. We already saw in our example of computing inverses using Euclid's algorithm that

$$2 \cdot 17 - 3 \cdot 11 = 1.$$

Now using SRT, we get

$$x = 2 \cdot 17 \cdot 3 - 3 \cdot 11 \cdot 4 = 2 \cdot 3(17 - 2 \cdot 11) = -30.$$

To get a positive representative, we can just add $17 \cdot 11 = 187$, so

$$x \equiv 157 \pmod{17 \cdot 11}.$$

## 3.4. Groups

Sets of integers equipped with addition modulo $n$ are examples of *groups*. Groups are central to the construction of public-key cryptography – we'll see how to define a secure key exchange based on a group with certain properties. This key exchange (the Diffie-Hellman key exchange) is fundamental in every widely used protocol on the internet today (TLS 1.3, Signal, etc). Here we just give the definition and some examples of groups to familiarise ourselves with the concept.

**Definition 3.3.** Let $G$ be a set and $* : G \times G \to G$ a map that takes pairs of elements in $G$ to a single element of $G$. We say that $(G, *)$ is a *group* or that $G$ *defines a group under* $*$ if the following *group axioms* are satisfied:

(G1) There exists $e \in G$ such that for every $g \in G$, $e * g = g * e = g$. *($G$ has an identity).*

(G2) For every $g \in G$, there exists $h \in G$ such that $g * h = h * g = e$. *(every element has an inverse).*

(G3) For every $a, b, c \in G$, $(a * b) * c = a * (b * c)$. *($*$ is associative).*

We often just say '$G$ is a group' instead of '$(G, *)$ is a group' if the author considers it 'obvious' which operation $*$ should be.

**Examples**

- For any integer $n \geq 2$, the set $\{0 \pmod n, 1 \pmod n, \ldots, n-1 \pmod n\}$ is a group under $+ \pmod n$.
- For any integer $n \geq 2$, the set $\{0 \pmod n, 2 \pmod n, \ldots, n-1 \pmod n\}$ is *not* a group under multiplication $\pmod n$. Reason: $0 \pmod n$ has no inverse (c.f. (G2)).
- For any composite integer $n \geq 2$, the set $\{1 \pmod n, 2 \pmod n, \ldots, n-1 \pmod n)$ is *not* a group under multiplication $\pmod n$. Reason: $n$ is composite, so there exists $0 \neq a \pmod n$ such that $\gcd(a, n) \neq 1$, which we proved above was not invertible.
- For any prime $p$, the set $\{1 \pmod p, \ldots, p-1 \pmod p\}$ is a group under multiplication $\pmod p$.

Sets of integers $\pmod p$ and $\pmod n$ will return again and again, so let us introduce some notation for this. From now on, we will write

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

and $(\mathbb{Z}/n\mathbb{Z})^*$ for the set of invertible elements of $\mathbb{Z}/n\mathbb{Z}$.

Note that for a prime $p$, that means that

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1 \pmod p, \ldots, p-1 \pmod p\};$$

we saw in our examples above that $(Z/p\mathbb{Z})^*$ is a group under multiplication $\pmod p$. This group turns out to be very useful for us, partly because it is *cyclic* for any prime $p$. That is, there exists a $g \pmod p$ such that

$$(\mathbb{Z}/p\mathbb{Z})^* = \{g \pmod p, g^2 \pmod p, \ldots, g^{p-1} \pmod p\}.$$

Another word for this is to say that $g$ *generates* $(\mathbb{Z}/p\mathbb{Z})^*$.

**Definition 3.4.** Let $(G, *)$ be a group. We say that $g \in G$ *generates* $G$ if

$$G = \{g, g * g, g \underbrace{* \cdots *}_{|G| \text{ times}} g\}.$$

We then call $g$ a *generator*.

For example, if $p = 5$ it turns out that we can choose $g = 2$:

$$(\mathbb{Z}/5\mathbb{Z})^* = \{2 \pmod 5, 4 \equiv 2^2 \pmod 5, 3 \equiv 2^3 \pmod 5, 1 \equiv 2^4 \pmod 5\}.$$

In this example, you see that the last element in the list, $g^{p-1} \pmod p$, is $1 \pmod p$. In fact, this is not a coincidence: This follows from *Fermat's Little Theorem.*

## 3.5. Diffie-Hellman key exchange

In Lecture 1 we saw the *one-time pad*, which is a secret known by multiple people which then can be used for cryptography. However, having a one-time pad with which we can work requires secure offline communication, which for most real-world scenarios is not practical and definitely not cost-effective. The cryptographic solution to this is to use a *key-exchange* algorithm, which does exactly what it says on the tin: it allows two (or more but we focus on two for now) parties who communicate over an open channel to compute a shared secret value, known only to them, which they can then use to encrypt communication between them.

The abstract idea of a key exchange is as follows: suppose Diffie and Hellman want compute a shared secret key (read: a shared value that no-one else can compute).

$$\begin{tikzcd}
\textbf{Diffie} & & \textbf{Hellman} \\
\text{generates key pair} & & \\
(sk_D, pk_D) \arrow[rrd, "pk_D"] & & \\
& & \text{generates key pair} \\
& & (sk_H, pk_H) \arrow[lld, "pk_H"'] \\
\text{computes shared secret} & & \text{computes shared secret} \\
(sk_D, pk_H) \rightsquigarrow ss & & (sk_H, pk_D) \rightsquigarrow ss.
\end{tikzcd}$$

Here is a(n overly simple) message encryption scheme built using such a key-exchange:

1. Alice and Bob compute $ss$ via a key-exchange and encode it as a bit string.
2. Alice encodes her plaintext message $m$ as a a bit string, computes the ciphertext $c = m \oplus s$, and sends $c$ to Bob.
3. Bob decrypts the ciphertext via $m = c \oplus s$.

So, how do we instantiate such a key-exchange? The most basic version of the Diffie-Hellman key exchange uses exponentiation modulo a large prime $p$. A prime $p$ and a non-zero element $g \pmod{p}$ is fixed in a public setup phase, and the key exchange is as follows:

$$\begin{tikzcd}
\textbf{Diffie} & p, g & \textbf{Hellman} \\
\text{generates key pair} & & \\
(d \in \mathbb{Z}, g^d \pmod{p}) \arrow[rrd, "g^d"] & & \\
& & \text{generates key pair} \\
& & (h \in \mathbb{Z}, g^h \pmod{p}) \arrow[lld, "g^h"'] \\
\text{computes shared secret} & & \text{computes shared secret} \\
ss \equiv (g^h)^d \pmod{p} & & ss \equiv (g^d)^h \pmod{p}.
\end{tikzcd}$$

In order for this to define a *crypto*system, we need more than just mathematical validity. We need any attack method to be much much slower than the algorithms used by the honest participants. 'Much slower' is something that we need to define in order to understand how to choose security parameters; for this we introduce a *security parameter* $\lambda$, which will be some smallish positive integer (often 128 in real-world scenarios), and which we use to discuss the amount of time an algorithm takes in terms of $\lambda$.

When we say that we want a computation to be 'fast' or *polynomial-time* we mean 'the number of basic operations for said computation is polynomial in $\lambda$', i.e., you can abstractly compute an upper bound on the number of basic operations (e.g. addition) needed as a polynomial in $\lambda$. When we say that we want a computation to be 'slow' we 'the number of basic operations is exponential or subexponential in $\lambda$', meaning that the number of basic operations for said computation is lower bounded by $O(2^\lambda)$ or $O(\lambda^\alpha \log_2(\lambda)^{1-\alpha})$ for some $\alpha \in (0,1)$ respectively[5]; these are referred to *exponential* and *subexponential* algorithms respectively. In practise, if this is true for a reasonable $\alpha$ (for example, for RSA which we will see below, $\alpha = 1/3$), we can increase $\lambda$ to a size for which polynomial time calculations are at most milliseconds and subexponential calculations would take years.

For Diffie-Hellman, if we choose $p$ so that $\lambda \approx \log_2(p)$, then exponentiation mod $p$ should be easy/fast/polynomial in $\lambda$ (more on this later) and the *discrete logarithm problem*, or computing $d^{\text{th}}$ or $h^{\text{th}}$ roots mod $p$, should be hard/slow/subexponential in $\lambda$ (more on this later too).

## 3.6. The Discrete Logarithm Problem

The Diffie-Hellman key exchange relies on certain computations being easy (fast) or hard (slow). For Diffie-Hellman, exponentiation mod $p$ should be easy/fast/polynomial-time, which we will later see is possible with square-and-multiply.

The fundamental problem that should be hard/slow/(sub)exponential-time for Diffie-Hellman is the *discrete logarithm problem*, that is, computing $d \in [0, p-1]$ given $g \pmod p$ and $g^d \pmod p$.[6]

There are instances where this might be very easy, for example if $g = p-1 \equiv -1 \pmod p$ then the only values of $g^d$ or $g^h$ that can occur are $-1$ and $1$ so finding a root is very easy. To avoid this, we want $g^d$ to be able to take as many values as possible.

For the Diffie-Hellman key exchange, our key space is $(\mathbb{Z}/p\mathbb{Z})^*$, where $p$ is a prime, which we saw last week is defined by

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1 \pmod p, \dots, p-1 \pmod p\};$$

[5] A reminder on Big-Oh notation: $x = O(f(x))$ for some function $f$ means that there exists constants $N, c > 0$ so that $x \le cf(x)$ for all $x \ge N$. In the context of algorithmic run-time, if the function $f$ is a polynomial in $x$, then we say that the algorithm runs in polynomial time.

[6] Computing $d$ given $g^d$ if $g \in \mathbb{R}$ is something you've seen before: namely computing logarithms base $g$. However, the problem turns out be fundamentally different when instead of working in a continuous solution set like $\mathbb{R}$ we are working in a discrete solution set like $\mathbb{Z}/p\mathbb{Z}$–hence the name the *Discrete Logarithm Problem*.

we saw also that $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group under multiplication $\pmod p$. That is, there exists a $g \pmod p$ such that

$$(\mathbb{Z}/p\mathbb{Z})^* = \{g \pmod p, g^2 \pmod p, \ldots, g^{p-1} \pmod p\}.$$

The reason that it is important for Diffie–Hellman that $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic is because it is possible to choose $g \pmod p \in (\mathbb{Z}/p\mathbb{Z})^*$ for which $g^d \pmod p$ takes $p-1$ different values: so that there is exactly one valid private key ($d$) for any given public key ($g^d$). In fact, the reason that we chose the letter 'g' when setting up the key exchange is because we typically choose a *generator* for the cyclic group $(\mathbb{Z}/p\mathbb{Z})^*$ (or a subgroup of that, but for simplicity we ignore that for now).

In conclusion, for our Diffie-Hellman setup, we choose $p$ prime and $g$ a generator of the group $(\mathbb{Z}/p\mathbb{Z})^*$. Observe that this choice only avoids the most obvious reason for the discrete logarithm problem (computing $d$ from $g^d \pmod p$) being easy; we'll get to other algorithms to compute discrete logarithms in due course.

# Lecture 4 – Public-Key Encryption

In a previous lecture, we learnt about modular arithmetic, Euclid's algorithm and the Diffie-Hellman Key Exchange protocol. In this lecture we'll see how to evolve the Diffie-Hellman Key Exchange protocol to an encryption algorithm (El Gamal), as well as an attack that teaches us that we need to be careful when setting parameters. We'll also learn about the RSA algorithm.

## 4.1. El Gamal encryption

A more sophisticated method to use the Diffie-Hellman key exchange to build an encrypted messaging protocol is called *El Gamal* encryption. El Gamal works as follows:

**Setup:**

1. Diffie chooses a prime $p$ and a generator $g$ of $\mathbb{Z}/p\mathbb{Z} - \{0\}$.
2. Diffie chooses a random secret $d \in \{1, \dots, p-1\}$ and computes $pk_D = g^d \pmod{p}$.
3. Diffie sends his public key $(p, g, pk_D)$ to Hellman.

**Encryption:**

1. Hellman chooses a random secret $h \in \{1, \dots p-1\}$ and computes $pk_H = g^h \pmod{p}$.
2. Hellman computes the shared secret $ss = pk_D^h \pmod{p}$.
3. Hellman computes the encrypted message $enc_m = m \cdot ss$.
4. Hellman sends the ciphertext $(pk_H, enc_m)$ to Diffie.

**Decryption:**

1. Diffie computes the shared secret $ss = pk_H^d \pmod{p}$.
2. Diffie computes the ciphertext $m = enc_m \cdot ss^{-1} = enc_m \cdot pk_H^{p-1-d} \pmod{p}$.

Let us make some observations about El Gamal encryption:

- Note that $ss^{-1} = pk_H^{p-1-d}$ as

$$ss \cdot pk_H^{p-1-d} = g^{dh} \cdot g^{h \cdot (p-1-d)} = (g^{p-1})^h = 1 \pmod{p}.$$

- If $m$ is known, the shared secret $ss$ can be recovered from the ciphertext, so use a new secret $h$ for each message.

For both the Diffie-Hellman Key Exchange protocol and the El Gamal encryption algorith, we require that the Discrete Logartithm problem is hard for our group of choice. This means that we need to make a choice of $p$ that ensures that the Discrete Logarithm problem is hard.

## 4.2. SRT vs. the Discrete Logarithm Problem

Later in the course we will look at the some of best known classical (that is 'not quantum') algorithms to attack the Discrete Logarithm Problem, – i.e. computing $d \in [0, p-1]$ given $g^d \pmod{p}$. In some contexts though we already have the tools we need: Sun-Tzu's Remainder Theorem!

We define the *order* $d$ of an element $g$ of a group $G$ with group operation $*$ and identity $id$ as the minumum positive integer $d$ such that $\underbrace{g * \cdots * g}_{d \text{ times}} = id$. Of course in our context this looks like the minumum positive integer $d$ such that $g^d \equiv 1 \pmod{p}$. In particular, the generator $g$ that we've been using in Diffie-Hellman and El Gamal has order $p-1$ (if you don't immediately see why, look forward to Fermat's Little Theorem and take some time to think about this).

Going back to the example with $p = 7$, you can hopefully now spot that $3^2 \equiv 2 \pmod{7}$ is an element of order 3, and $3^3 \equiv 6 \pmod{7}$ is an element of order 2. These are examples of a more general concept: if $g$ generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ (so has order $p-1$) and $\ell$ divides $p-1$, then $g^{\frac{p-1}{\ell}} \pmod{p}$ has order $\ell$ (exercise: prove this); this gives an easy way of finding elements of a given order–this is going to be very useful in the following example.

**Example** Suppose that we want to solve the following Discrete Logarithm Problem: find $a \in \mathbb{Z}$ such that $2^a \equiv 17 \pmod{37}$, and you are given that 2 is a generator of the multiplicative group $(\mathbb{Z}/37\mathbb{Z})^*$. Then as $a$ is in the exponent, it suffices to compute $a \pmod{36}$ (because of Fermat's Little Theorem). If we want to compute $a \pmod{36}$, by Sun-Tzu's Remainder Theorem it suffices to compute $a \pmod{4}$ and $a \pmod{9}$. This is something we can just do by brute force and observation, but there is a more effiicient way using the group theory above:

- To compute $a \pmod{4}$, we first compute $a \pmod{2}$. Write $a = a_0 + 2a_1$ where $a_0 \in \{0, 1\}$. By the above, we know that $2^{(p-1)/2} = 2^{18}$ is an element of order 2. Substituting for $a, a_0$, and $a_1$, we get the following equalities mod 37

  $$-1 \equiv 17^{18} \equiv (2^a)^{18} \equiv 2^{18a_0+36a_1} \equiv (2^{18})^{a_0} \cdot (2^{36})^{a_1} \equiv (-1)^{a_0},$$

  from which we can read off that $a_0 = 1$, so $a \equiv 1 \pmod{2}$.

- Now we compute $a \pmod{4}$. We know that $a \equiv 1 \pmod{2}$, so there exist $a_1, a_2$ with $a_1 \in \{0, 1\}$ such that $a = 1 + 2a_1 + 4a_2$. By the above, we know that $2^{(p-1)/4} = 2^9$ is an element of order 4. Substituting for $a, a_1, a_2$, we get the following equalities mod 37

$$31 \equiv 17^9 \equiv (2^{1+2a_1+4a_2})^9 \equiv 2^9 \cdot (2^{18})^{a_1} \cdot (2^{36})^{a_2} \equiv 6 \cdot (-1)^{a_1},$$

from which we can read off that $a_1 = 1$, so $a \equiv 3 \pmod 4$.

- To compute $a \pmod 9$, we first compute $a \pmod 3$. Write $a = a_0 + 3a_1$ where $a_0 \in \{0, 1, 2\}$. By the above, we know that $2^{(p-1)/3} = 2^{12}$ is an element of order 3. Substituting for $a, a_0, a_1$ we get the following equations mod 37

$$26 \equiv 17^{12} \equiv (2^a)^{12} \equiv 2^{12a_0+36a_1} \equiv (2^{12})^{a_0} \cdot (2^{36})^{a_0} \equiv 26^{a_0},$$

from which we can read off that $a_0 = 1$.

- Now we compute $a \pmod 9$. We know that $a \equiv 1 \pmod 3$, so there exist $a_1, a_2$ with $a_1 \in \{0, 1, 2\}$ such that $a = 1 + 3a_1 + 9a_2$. By the above, we have that $2^{(p-1)/9} = 2^4$ is an element of order 9. Substituting for $a, a_1, a_2$, we get the following equations mod 37

$$12 \equiv 17^4 \equiv (2^a)^4 \equiv 2^{4+12a_1+36a_2} \equiv 2^4 \cdot (2^{12})^{a_1} \cdot (2^{36})^{a_2} \equiv 16 \cdot 26^{a_1},$$

from which we can read off that $a_1 = 2$, so $a \equiv 7 \pmod 9$.

- Now we know that $a \equiv 3 \pmod 4$ and $a \equiv 7 \pmod 9$, which by CRT we know uniquely defines $a \pmod{36}$. We can compute this via Euclid's algorithm as we've done before, giving $a \equiv 7 \pmod{36}$.

The above method is in no way specific to the numbers we have chosen (37, 4, 9, etc): it is in fact an example of an algorithm that works in general. This trick of using Sun-Tzu's Remainder Theorem to attack the Discrete Logarithm Problem is due to Pohlig and Hellman and is therefore referred to as the Pohlig-Hellman algorithm.

Next we'll look at another encryption algorithm: RSA. However, we must first introduce a vital mathematical theorem.

## 4.3. Fermat's Little Theorem

Fermat's Little Theorem comes up again and again when dealing with modular arithmetic. It is a useful identity in its own right, but it is also another way of computing inverses mod $n$, as well as fundamental in the construction of RSA.

**Definition 4.1.** Let $n \in \mathbb{Z}_{>0}$. The *Euler $\varphi$-function* of $n$ is

$$\varphi(n) := \#\{m \in \mathbb{Z} : 0 < m < n, \gcd(m, n) = 1\}.$$

**Examples.** 1. $\varphi(7) = \#\{1, 2, 3, 4, 5, 6\} = 6$.

2. $\varphi(8) = \#\{1, 3, 5, 7\} = 4$.

Exercise: prove that for $p \neq q$ prime,

$$\varphi(p) = p - 1$$

and

$$\varphi(pq) = (p-1)(q-1).$$

**Theorem 4.1** (Fermat's Little Theorem)**.** *For every $a \in \mathbb{Z}$ and squarefree $n \in \mathbb{Z}_{>1}$,*

$$a^{\varphi(n)+1} \equiv a \pmod{n}.$$

Note in particular that is $n = p$ is prime, then this identity becomes

$$a^{p-1} \equiv 1 \pmod{p},$$

which is what we observed in the example above. This also means that for any $a$ coprime to $p$, the inverse of $a$ can be computed by repeated exponentiation as $a^{p-2} \pmod{p}$.

## 4.4. RSA

This section is about RSA, named after Rivest, Shamir, and Ademan. RSA was the first public key encryption (PKE) and signature system, and is still in wide use today.

The basic RSA public key encryption system consists of 3 steps: a setup phase for key generation by the user (**KeyGen**), encryption of a message $m$ by a second party (**Encrypt**), and decryption of the message $m$ by the user (**Decrypt**). The algorithms for these steps are below. We have coloured the users secrets in red and the public values in green.

### KeyGen

1. Pick primes $p \neq q$ of bit length $\lambda$.
2. Compute $n = p \cdot q$ and $\varphi(n) = (p-1)(q-1)$.
3. Pick $e$ coprime to $\varphi(n)$.
4. Compute $d = e^{-1} \pmod{\varphi(n)}$.
5. Generate key pair

$$\mathsf{pk}, \mathsf{sk} = (e, n), (d, n).$$

### Encrypt

1. Pick $m \in \mathbb{Z}_{[0,n-1]}$.
2. Compute $c \equiv m^e \pmod{n}$.
3. Send $c$.

**Decrypt**

1. Compute $c^d \equiv m \pmod{n}$.

In order for this to be a valid system, there are two steps that don't obviously mathematically check out: step 4 of **KeyGen** (does the inverse exist?) and step 1 of **Decrypt**.

Recall from last week that $a \pmod{b}$ is invertible if and only if $a$ and $b$ are coprime, so step 4 of **KeyGen** is valid.

For the **Decrypt** step, note that if $m$ *is* invertible mod $n$ then Fermat's Little Theorem implies that $m^{\varphi(n)} \equiv 1 \pmod{n}$. Let $k \in \mathbb{Z}$ be such that $ed = 1 + k\varphi(n)$. Then

$$c^d \equiv (m^e)^d \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot 1^k \equiv m \pmod{n},$$

so the decrypt step is valid (if $m$ is invertible mod $n$, actually also if it's not but that requires some more steps).

For RSA to work as a cryptosystem:

- Step 2 of **KeyGen** needs to be fast, i.e., we need to be able to multiply fast.
- Step 4 of **KeyGen** needs to be fast, i.e., we need to be able to compute inverses mod $\varphi(n)$ fast.
- In Step 5 of **KeyGen**, an attacker shouldn't be able to compute $d$ from $(e, n)$. Because Step 4 is fast, if the attacker knows $\varphi(n)$ then they can compute $d$ fast. So computing $\varphi(n)$ from $n$ should be slow. As $\varphi(n)$ is easy to compute from $p$ and $q$, factoring $n$ should be slow.
- Step 2 of **Encrypt** needs to be fast, i.e., we need to be able to exponentiate mod $n$ fast.
- An attacker should also not be able to recover $m$ from $c$, so computing $e^{\text{th}}$ roots mod $n$ should be slow.

## 4.5. Fast multiplication, exponentiation, and inversion

Let us first focus on the computations we want to be fast in RSA. To multiply fast, we use a method called 'double-and-add'. To see how this works let's consider how we would multiply $p$ by $q$ in Step 2 of **KeyGen**. We first write $q$ in binary as $(q_\lambda, \ldots, q_0)$, or in other words

$$q = \sum_{i=0}^{\lambda} q_i 2^i,$$

where $q_i \in \{0, 1\}$. In particular

$$pq = \sum_{i=0}^{\lambda} q_i \cdot (2^i p).$$

We can compute the $2^i p$ terms just by repeated doubling – each doubling is just one addition (which is a basic operation) so this is very efficient.

- **Double**: Compute

$$\begin{aligned}
2^0 \cdot p = p &\rightarrow \text{ 0 additions} \\
2^1 \cdot p = p + p &\rightarrow \text{ 1 addition} \\
2^2 \cdot p = 2p + 2p &\rightarrow \text{ 1 addition} \\
&\cdots \\
2^\lambda \cdot p = 2^{\lambda-1}p + 2^{\lambda-1}p &\rightarrow \text{ 1 addition.}
\end{aligned}$$

  The doubling step costs $\lambda$ additions, so is polynomial in $\lambda$ i.e. fast.

Now to get $p \cdot q$ we just have to add together the terms $2^i \cdot p$ together for which $q_i = 1$. So, let $q_{i_0}, \dots, q_{i_k}$ be the non zero coefficients of the binary expansion of $q$.

- **Add**: Compute

$$p \cdot q = (2^{i_0} \cdot p) + \cdots + (2^{i_k} \cdot p).$$

  The adding step costs $k \leq \lambda$ additions.

In total, the double-and-add method costs at most $2\lambda$ basic operations, so is 'fast'.

To exponentiate mod $n$ fast, we play the same game. To see how this works let's consider computing $m^e \pmod{n}$ as we do in **Encrypt**. This time we use the binary expansions of $e = (e_\lambda, \dots, e_0)$, so in other words

$$e = \sum_{i=0}^{\lambda} e_i 2^i,$$

where $e_i \in \{0, 1\}$. In particular

$$m^e = m^{\sum_{i=0}^{\lambda} e_i 2^i} = \prod_{i=0}^{\lambda} (m^{2^i})^{e_i}.$$

We can compute the $m^{2^i} \pmod{n}$ terms just by repeated squaring – each squaring is at most one multiplication (maybe you get some cancellation so it could be less), each of which we just saw is at most $2\lambda$ basic operations.

- **Square**: Compute

$$\begin{aligned}
m^{2^0} &\equiv m \pmod{n} \rightarrow \text{ 0 squarings} \\
m^{2^1} &\equiv m^2 \pmod{n} \rightarrow \text{ 1 squaring} \\
m^{2^2} &\equiv (m^2)^2 \pmod{n} \rightarrow \text{ 1 squaring} \\
&\cdots \\
m^{2^\lambda} &\equiv (m^{2^{\lambda-1}})^2 \pmod{n} \rightarrow \text{ 1 squaring.}
\end{aligned}$$

  The squaring step costs $\lambda$ squarings, so is at most $2\lambda^2$ basic operations.

Note that the fact that we are doing the computations $\pmod{n}$ here is very important - for large $\lambda$ you would quickly run into memory problems otherwise. All that remains now to get $m^e \pmod{n}$ is to multiply together the terms $m^{2^i} \pmod{n}$ together for which $e_i = 1$. So, let $e_{i_0}, \ldots, e_{i_k}$ be the non zero coefficients of the binary expansion of $e$.

- **Multiply**: Compute

$$m^e \pmod{n} \equiv m^{2^{i_0}} \times \cdots \times m^{2^{i_k}} \pmod{n}.$$

The multiplication step then costs $k \leq \lambda$ multiplications, so $\leq 2\lambda^2$ basic operations.

From these calculations, we see that square-and-multiply can always be performed in $\leq 4\lambda^2$ basic operations, so is polynomial time, i.e. 'fast'. In practise you can do quite a bit better! But we won't go into that now.

If we look now at our list of computations that we want to be fast for RSA to work as a cryptosystem, we've tackled multiplication and exponentiation, and only inversion $\pmod{\varphi(n)}$ remains.

We saw two algorithms for computing inverses last week: Euclid's corollary and Fermat's Little Theorem. However, as $\varphi(n)$ may have many small factors the most efficient algorithm here would be to combine Euclid's corollary with Sun-Tzu's Remainder Theorem–see the exercise sheet for this week.

# Lecture 5 – Symmetric Authentication

So far, we have focused on protecting the *confidentiality* of messages. But our motivation is to protect the *security* of messages. In particular, we have done nothing at all to prevent our adversaries from modifying messages: in fact, in most of the schemes and constructions we studied in depth, the adversary can cause a predictable change in the plaintext by modifying a ciphertext.

We'll focus on integrity and authenticity—two related notions with subtle differences we won't really explore here, and consider security definitions and constructions for Message Authentication Codes (MACs)—keyed functions allowing a sender and recipient with a shared secret to protect messages against modification; and we will consider security notions for hash functions—*public* functions meant to ensure a similar property in the presence of a separate high integrity channel.

We will then see how to combine confidentiality and integrity by defining a security notion for *authenticated encryption*, and considering some generic ways of composing nonce-based encryption schemes and MACs to obtain secure authenticated encryption. These are as close as we will get to a true secure channel, and will work as long as we know how to securely establish short secrets from public information. (For example, using Diffie-Hellman.)

## 5.1. Message Authentication Codes

Message authentication codes operate by producing—from the key and message—an *authentication tag* (or simply a tag) that can be used—jointly with the key and message—to verify that the message was not modified since the tag was computed.

### 5.1.1. Syntax and Security

We define the syntax and correctness of those schemes formally as follows.

**Definition 5.1** (Message Authentication Code (MAC)). A *message authentication code* $\textsc{Mac} = (\mathsf{Kg}, \mathsf{Tag}, \mathsf{Vfy})$, where $\mathsf{Kg}$ randomly generates a key $k \in \mathcal{K}$, $\mathsf{Tag}$ takes a key $k$ and a message $m \in \mathcal{M}$ to output tag $t \leftarrow \mathsf{Tag}_k(m) \in \mathcal{T}$, and $\mathsf{Vfy}$ takes a key $k$ and a message–tag pair $(m, t)$ to output either $\top$ (valid) or $\bot$ (invalid).

The MAC scheme is *correct* iff, for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$, $\mathsf{Vfy}_k(m, \mathsf{Tag}_k(m)) = \top$.

Definition 5.1 leaves open the possibility that the Tag algorithm is probabilistic. For most practical schemes, Tag is in fact deterministic, and verification simply recomputes the tag: $\mathsf{Vfy}_k(m, t)$ outputs $\top$ exactly when $\textsc{Mac}_k(m) = t$. In this unit, we consider only schemes that use this type of verification, and leave the Vfy algorithm unspecified from now on. Still, it is

```
$\mathsf{Exp}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A})$
$k \leftarrow_\$ \mathsf{Kg}$
$(\widehat{m}, \widehat{t}) \leftarrow_\$ \mathbb{A}^{\mathcal{T}_{\text{cma}}}$

    $\mathcal{T}_{\text{cma}}(m)$
    $t \leftarrow \mathsf{Tag}_k(m)$
    return $t$
```

```
$\mathsf{Exp}^{\text{uuf-cma}}_{\textsc{Mac}}(\mathbb{A})$
$k \leftarrow_\$ \mathsf{Kg}$
$m^* \leftarrow_\$ \mathcal{M}$
$\widehat{t} \leftarrow_\$ \mathbb{A}^{\mathcal{T}_{\text{cma}}}(m^*)$

    $\mathcal{T}_{\text{cma}}(m)$
    require $m \neq m^*$
    $t \leftarrow \mathsf{Tag}_k(m)$ return $t$
```

Figure 5.1.: Two different unforgeability experiments for Mac

sometimes useful to refer to the act of "verifying a tag with respect to a message and a key"', so we will keep the terminology and notation.

**Definition 5.2** (Existential Unforgeability under Chosen Message Attack)**.** The *advantage of an adversary $\mathbb{A}$ in existentially forging a Mac tag under chosen message attack* is defined as follows—where $\mathsf{Exp}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A})$ is defined in Figure 5.1, and a message being *fresh* in a given run means that it was never used as an input to the $\mathcal{T}_{\text{cma}}$ oracle.

$$\mathsf{Adv}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A}) : \mathsf{Vfy}_k(\widehat{m}, \widehat{t}) = \top \wedge \ \widehat{m} \text{ is fresh}\right]$$

We say that Mac is $(t, q, \epsilon)$*-existentially unforgeable under chosen message attack* if, for every $\mathbb{A}$ that runs in time at $t$ and makes at most $q$ queries to their $\mathcal{T}_{\text{cma}}$ oracle, we have $\mathsf{Adv}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A}) \leq \epsilon$.

**Definition 5.3** (Universal Unforgeability under Chosen Message Attack)**.** The *advantage of an adversary $\mathbb{A}$ in existentially forging a Mac tag under chosen message attack* is defined as follows—where $\mathsf{Exp}^{\text{uuf-cma}}_{\textsc{Mac}}(\mathbb{A})$ is defined in Figure 5.1.

$$\mathsf{Adv}^{\text{uuf-cma}}_{\textsc{Mac}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\text{euf-cma}}_{\textsc{Mac}}(\mathbb{A}) : \mathsf{Vfy}_k(\widehat{m}, \widehat{t}) = \top\right]$$

We say that Mac is $(t, q, \epsilon)$*-universally unforgeable under chosen message attack* if, for every $\mathbb{A}$ that runs in time at $t$ and makes at most $q$ queries to their $\mathcal{T}_{\text{cma}}$ oracle, we have $\mathsf{Adv}^{\text{uuf-cma}}_{\textsc{Mac}}(\mathbb{A}) \leq \epsilon$.

**Guessing attack and lower bound on insecurity.** As with previous definitions, let us first consider the best we could hope to do. Let us consider the weakest notion we can think of: UUF-PAS (universal unforgeability under passive attack). In that case, the very best even a clever adversary can do is guess a tag, which succeeds with probability at least $1/\left|\mathcal{T}\right|$. Here again, this implies that tags should be long enough *at least* for you to be happy with the level of insecurity implied by this best case attack.

```
$\text{CBC-MAC}_k(m)$
$(m[1], \ldots, m[n]) \leftarrow \mathsf{parse}(m)$
$X[0] \leftarrow 0^\ell$
for $i \in [1, \ldots, n]$
    $Y[i] \leftarrow X[i-1] \oplus m[i]$
    $X[i] \leftarrow \mathsf{E}_k(Y[i])$
return $X[n]$
```

```
$\text{C}^*\text{-MAC}_{k_1,k_2}(m)$
$(m[1], \ldots, m[n]) \leftarrow \mathsf{pad}(m)$
$X[0] \leftarrow 0^\ell$
for $i \in [1, \ldots, n]$
    $Y[i] \leftarrow X[i-1] \oplus m[i]$
    $X[i] \leftarrow \mathsf{E}_{k_1}(Y[i])$
$t \leftarrow \mathsf{F}_{k_2}(X[n])$
return $t$
```

Figure 5.2.: CBC-MAC: the vanilla version for $\mathcal{M} = \{0,1\}^{\ell \cdot n}$ in the left panel; the usual template for dealing with $\mathcal{M} = \{0,1\}^*$ in the right panel.

### 5.1.2. CBC-MAC

A popular way of building MACs is to use a blockcipher in CBC mode, retaining only the last block of ciphertext. The resulting construction, CBC-MAC, is shown in Figure 5.2. We can prove it is EUF-CMA secure as long as the underlying blockcipher is IND secure and as long as the length of messages is fixed *a priori* to some $n \cdot \ell$ (where $\ell$ is the block size).

To make it secure in practice, some form of post-processing is needed. One simple form of post-processing is shown in the right pane of Figure 5.2, and consists in running the tag through an independent blockcipher (or the same blockcipher with an independent key) before output. CMAC, a standard based on this, is slightly more involved because it attempts to minimise padding—which we discuss now.

### 5.1.3. Padding: Dealing with Arbitrary-Length Messages

So far, we've defined our constructions only on well-behaved messages that could easily be parsed into blocks. We need to explain how this parsing can be done in practice, in a way that doesn't weaken security.[1]

The most pervasive way of allowing arbitrary-length inputs is *padding*, which consists in defining an *injective* function pad $\in \{0,1\}^* \rightarrow (\{0,1\}^\ell)^*$—that is, a function that turns a string of bits into a string of blocks in—at least theoretically—invertible way.

For MACs, the inverse does not need to be efficiently computable for correctness (but it might need to be efficiently computable for security proofs to make sense). For encryption, the inverse does need to be efficiently computable for correctness, as well as for security proofs.

One widely-used padding scheme is the $10^*$ padding scheme (or some byte-level variant), which involves padding the message with at least one $1$ bit, followed by as many zeroes as needed to align with the block length. It can easily inverted by looking back from the end of the padded string for the last $1$ bit, and dropping it and all following bits. (Note that this is partial! It is important that "unpadding" can fail.)

[1] We had scope to discuss padding in relation to encryption as well, but there, getting it wrong in the normal ways only threatens correctness, which we don't care overmuch about in this unit.

```
$\mathsf{Exp}^{\mathsf{cr}}_{\mathsf{H}}(\mathbb{A})$
$k \leftarrow_{\$} \mathcal{K}$
$(\widehat{m_1}, \widehat{m_2}) \leftarrow_{\$} \mathbb{A}(k)$
```

$$\mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{cr}}_{\mathsf{H}}(\mathbb{A}) : \begin{array}{l} \widehat{m_1} \neq \widehat{m_2} \\ \wedge\ \mathsf{H}_k(m_1) = \mathsf{H}_k(m_2) \end{array}\right]$$

```
$\mathsf{Exp}^{\mathsf{pr}}_{\mathsf{H}}(\mathbb{A})$
$k \leftarrow_{\$} \mathcal{K}$
$m^* \leftarrow_{\$} \mathcal{M}$
$d^* \leftarrow \mathsf{H}_k(m^*)$
$\widehat{m} \leftarrow_{\$} \mathbb{A}(k, d^*)$
```

$$\mathsf{Adv}^{\mathsf{pr}}_{\mathsf{H}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{pr}}_{\mathsf{H}}(\mathbb{A}) : \mathsf{H}_k(\widehat{m}) = d^*\right]$$

```
$\mathsf{Exp}^{\mathsf{pr2}}_{\mathsf{H}}(\mathbb{A})$
$k \leftarrow_{\$} \mathcal{K}$
$m^* \leftarrow_{\$} \mathcal{M}$
$\widehat{m} \leftarrow_{\$} \mathbb{A}(k, m^*)$
```

$$\mathsf{Adv}^{\mathsf{pr2}}_{\mathsf{H}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{pr2}}_{\mathsf{H}}(\mathbb{A}) : \begin{array}{l} \widehat{m} \neq m^* \\ \wedge\ \mathsf{H}_k(\widehat{m}) = \mathsf{H}_k(m^*) \end{array}\right]$$

Figure 5.3.: Hash function security notions: collision resistance (top), preimage resistance (bottom left), and second preimage resistance (bottom right)

## 5.2. Cryptographic Hash Functions

MACs are powerful, but require that the sender and recipient share a key and trust each other to not misuse it. This is not useful if a single sender wants to send to multiple recipients: anyone who can verify the tag can also compute it! Cryptography offers a public alternative—*hash functions*—which provide some form of integrity protection, and often also serve as a building block in many other constructions. For technical reasons, we must define hash functions as keyed functions instead.

### 5.2.1. Syntax and Security

**Definition 5.4** (Hash Function)**.** A hash function is a $\mathcal{K}$-indexed family of algorithms $\mathsf{H}_k : \mathcal{M} \to \mathcal{D}$ that take as input a message $m \in \mathcal{M}$ and outputs a *digest* $d \in \mathcal{D}$.

In order to speak of a hash function we require that the function *compresses*, that is $|\mathcal{M}| > |\mathcal{D}|$.

Typically, the cardinality $|\mathcal{M}|$ of the message space is a *whole lot* larger than that of the digest space $|\mathcal{D}|$. For instance, $\mathcal{D} = \{0,1\}^d$ for say $d = 256$, yet $\mathcal{M}$ consists of all bitstrings up to length $2^{64}$.

Cryptographic hash functions are typically expected to have three security properties: collision resistance, preimage resistance, and second preimage resistance. We define the corresponding experiments and advantages in Figure 5.3, without formally defining the detailed notions.

```
$\mathsf{Exp}^{\text{ae-real}}_{\mathrm{ENC}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}^{\mathcal{E}(\cdot,\cdot),\mathcal{D}(\cdot,\cdot)}$

    $\mathcal{E}(n, m)$
        no repeated nonces
        $c \leftarrow \mathsf{Enc}^{n}_{k}(m)$
        return $c$

    $\mathcal{D}(n, c)$
        $c$ not output by $\mathcal{E}(n, \cdot)$
        $m \leftarrow \mathsf{Dec}^{n}_{k}(c)$
        return $m$
```

```
$\mathsf{Exp}^{\text{ae-ideal}}_{\mathrm{ENC}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}^{\mathcal{E}(\cdot,\cdot),\mathcal{D}(\cdot,\cdot)}$

    $\mathcal{E}(n, m)$
        no repeated nonces
        $c \leftarrow_{\$} \mathcal{C}(|m|)$
        return $c$

    $\mathcal{D}(n, c)$
        $c$ not output by $\mathcal{E}(n, \cdot)$
        return $\perp$
```

Figure 5.4.: All-in-one AE security experiment

As always, generic attacks help us pick parameters such as the digest length. Collision resistance is vulnerable to birthday attacks and are the main constraint on digest length: for the same level of security, collision resistance requires digests to be twice as long as preimage resistance or second preimage resistance.

## 5.3. Authenticated Encryption

### 5.3.1. Syntax and Security

**Definition 5.5** ((Nonce-Based) Authenticated Encryption)**.** A nonce-based authenticated encryption scheme $\mathrm{E} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of algorithms where $\mathsf{Kg}$ randomly generates a key $k \in \mathcal{K}$, $\mathsf{Enc}$ takes a key $k$, a nonce $n \in \mathcal{N}$ and a message $m \in \mathcal{M}$ to output ciphertext $c \leftarrow \mathsf{Enc}^{n}_{k}(m) \in \mathcal{C}$, and $\mathsf{Dec}$ takes a ciphertext $c$, a nonce $n$ and a key $k$ to output a message $m$ or $\perp$ (denoting a decryption failure).

The authenticated encryption scheme is correct iff, for all $k \in \mathcal{K}$, $n \in \mathcal{N}$ and $m \in \mathcal{M}$, it holds that $\mathsf{Dec}^{n}_{k}(\mathsf{Enc}^{n}_{k}(m)) = m$.

All notations here assume that $\perp$ is not a valid message (that is, $\perp \notin \mathcal{M}$) so we can safely denote with $m$ the representation of $m$ in the extended set $\mathcal{M} \cup \{\perp\}$. (In practice, how to encode errors is one of those pitfalls that even experienced cryptography engineers get caught in.)

**Definition 5.6** (Authenticated Encryption Security)**.** The *advantage of an adversary* $\mathbb{A}$ *in distinguishing an authenticated encryption scheme* $\mathsf{Enc}$ *from an ideal encryption scheme* is defined as follows, where experiments $\mathsf{Exp}^{\text{ae-real}}_{\mathsf{Enc}}(\mathbb{A})$ and $\mathsf{Exp}^{\text{ae-ideal}}_{\mathsf{Enc}}(\mathbb{A})$ are defined in Figure 5.4.

```
$\mathsf{Exp}^{\text{(n)ind-cca-real}}_{\textsc{Enc}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}^{\mathcal{E}(\cdot,\cdot),\mathcal{D}(\cdot,\cdot)}$

    $\mathcal{E}(n, m)$
        no repeated nonces, and
        $m$ not output by $\mathcal{D}(n, \cdot)$
        $c \leftarrow \mathsf{Enc}^n_k(m)$
        return $c$

    $\mathcal{D}(n, c)$
        $c$ not output by $\mathcal{E}(n, \cdot)$
        $m \leftarrow \mathsf{Dec}^n_k(c)$
        return $m$
```

```
$\mathsf{Exp}^{\text{(n)ind-cca-ideal}}_{\textsc{Enc}}(\mathbb{A})$
    $k \leftarrow_{\$} \mathsf{Kg}$
    $\widehat{b} \leftarrow_{\$} \mathbb{A}^{\mathcal{E}(\cdot,\cdot),\mathcal{D}(\cdot,\cdot)}$

    $\mathcal{E}(n, m)$
        no repeated nonces, and
        $m$ not output by $\mathcal{D}(n, \cdot)$
        $c \leftarrow_{\$} \mathcal{C}(|m|)$
        return $c$

    $\mathcal{D}(n, c)$
        $c$ not output by $\mathcal{E}(n, \cdot)$
        $m \leftarrow \mathsf{Dec}^n_k(c)$
        return $m$
```

$$\mathsf{Adv}^{\text{(n)ind-cca}}_{\textsc{Enc}}(\mathbb{A}) = \left|\Pr\left[\mathsf{Exp}^{\text{(n)ind-cca-real}}_{\textsc{Enc}}(\mathbb{A}) : \widehat{b} = 1\right] - \Pr\left[\mathsf{Exp}^{\text{(n)ind-cca-ideal}}_{\textsc{Enc}}(\mathbb{A}) : \widehat{b} = 1\right]\right|$$

Figure 5.5.: The (N)IND-CCA experiment and security notion

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{Enc}}(\mathbb{A}) = \left|\Pr\left[\mathsf{Exp}^{\text{ae-real}}_{\mathsf{Enc}}(\mathbb{A}) : \widehat{b}\right] - \Pr\left[\mathsf{Exp}^{\text{ae-real}}_{\mathsf{Enc}}(\mathbb{A}) : \widehat{b}\right]\right|$$

An authenticated encryption scheme Enc is said to be $(t, q_{\mathcal{E}}, q_{\mathcal{D}}, \epsilon)$*-AE-secure* if, for every $\mathbb{A}$ that runs in time at most $t$, and makes at most $q_{\mathcal{E}}$ queries to its encryption oracle, and at most $q_{\mathcal{D}}$ queries to its decryption oracle, we have $\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{Enc}}(\mathbb{A}) \leq \epsilon$.

It might be an interesting exercise to show that the EUF-CMA notion we defined on MACs is equivalent to an indistinguishability notion inspired by the decryption oracle in the above. The security notion we use is in fact equivalent to being (N)IND-secure and being EUF-CMA secure (seeing the encryption algorithm as Tag, and the decryption algorithm as Vfy).

## 5.3.2. Chosen Ciphertext Attacks

We now have a security definition that allows the adversary to not only ask for encryptions of chosen plaintexts, but also for decryptions of chosen ciphertexts. This kind of threat model is in fact also useful for non-authenticated encryption, where the decryption oracle might in fact leak more than success. We describe the security experiments for nonce-based indistinguishability under chosen ciphertext attacks (IND-CCA) in Figure 5.5, without further formally defining the security notion. (Which goes as usual.)

It should be intuitively clear that any AE-secure scheme is (N)IND-CCA secure.

```
$\textsc{MtE}^{n}_{k_a,k_e}(m)$
    $t \leftarrow \mathsf{Tag}_{k_a}(n, m)$
    $c \leftarrow \mathsf{Enc}_{k_e}(n, m \| t)$
    **return** $c$
```

```
$\textsc{EtM}^{n}_{k_a,k_e}(m)$
    $c \leftarrow \mathsf{Enc}_{k_e}(n, m)$
    $t \leftarrow \mathsf{Tag}_{k_a}(n, c)$
    **return** $c \| t$
```

```
$\text{E+M}^{n}_{k_a,k_e}(m)$
    $c \leftarrow \mathsf{Enc}_{k_e}(n, m)$
    $t \leftarrow \mathsf{Tag}_{k_a}(n, m)$
    **return** $c \| t$
```

Figure 5.6.: Generic composition for authenticated encryption: mac-then-encrypt (left), encrypt-then-mac (middle), and encrypt-and-mac (right)

### 5.3.3. Constructing AE: Generic Composition

We can construct an AE-secure scheme from an (N)IND-secure encryption scheme and an EUF-CMA-secure MAC scheme. Figure 5.6 shows the three natural ways of doing this.

All three are AE-secure under reasonable assumptions on the encryption and MAC schemes, but Encrypt-then-MAC (EtM) is the most widely used because it is harder to implement insecurely: for MtE and E+M, it is very easy to leak more information than success or failure upon decryption failures, which will reveal more information than safe about the plaintext.

# Lecture 6 – The Discrete Logarithm Problem and Digital Signatures

## 6.1. Pohlig-Hellman

We have already seen an example of how to use the SRT to solve discrete logarithm problems. This is in fact an example of an algorithm due to Pohlig and Hellman, which we will now state in full.

The Pohlig-Hellman algorithm is a method to solve a discrete logarithm problem: given a prime $p$ and $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ of order $p-1$, and given $g^a \pmod{p}$, find $a$.

Now as, by Fermat's Little Theorem, for any $k \in \mathbb{Z}$ we have that $g^{a+(p-1)k} \equiv g^a \pmod{p}$, it suffices to find $a \pmod{p-1}$. So just as in the example above, we factorise $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

where the $q_i$ are prime. Then we use Sun-Tzu's Remainder Theorem to compute $a \pmod{p-1}$ from $a \pmod{q_1^{e_1}}, \ldots, a \pmod{q_r^{e_r}}$. The algorithm is as follows:

1. Factorise $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

where the $q_i$ are prime.

2. For each $i = 1, \ldots, r$,

   (i) Write $a = a_0 + a_1 q_i + a_2 q_i^2 + \cdots$, with $a_j \in [0, q_i - 1]$.

   (ii) Compute $a_0$, ie., $a \pmod{q_i}$: Note that

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^a \equiv (g^{\frac{p-1}{q_i}})^{a_0} \cdot (g^{p-1})^{(\cdots)} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod{p},$$

   so in particular

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod{p},$$

   and the values in red are all things we can compute. Just checking the $q_i$ options for $a_0$ gives us $a_0$, and hence $a \pmod{q_i}$.

(iii) For $k = 1, \ldots, e_i - 1$:
Given $a_0, \ldots a_{k-1}$, ie., $a \pmod{q_i^k}$, compute $a_k$, i.e., compute $a \pmod{q_i^{k+1}}$: Note that

$$(g^a)^{\frac{p-1}{q_i^{k+1}}} \equiv (g^{\frac{p-1}{q_i^{k+1}}})^a \equiv (g^{\frac{p-1}{q_i^{k+1}}})^{a_0+q_i a_1+\cdots+q_i^{k-1}a_{k-1}} \cdot (g^{\frac{p-1}{q_i}})^{a_k} \cdot (g^{p-1})^{(\cdots)} \pmod{p},$$

so in particular

$$\color{red}{(g^a)^{\frac{p-1}{q_i^{k+1}}}} \equiv \color{red}{(g^{\frac{p-1}{q_i^{k+1}}})^{a_0+q_i a_1+\cdots+q_i^{k-1}a_{k-1}}} \cdot \color{red}{(g^{\frac{p-1}{q_i}})}^{a_k} \pmod{\color{red}{p}},$$

and the values in red are all things we can compute. Just checking the $q_i$ options for $a_k$ gives us $a_k$, and hence $a \pmod{q_i^{k+1}}$.

3. Using Euclid's corollary, compute $a \pmod{p-1}$ from the values $a \pmod{q_1^{e_1}}, \ldots, a \pmod{q_r^{e_r}}$.
Note: You'll need to first compute $a \pmod{q_1^{e_1} \cdot q_2^{e_2}}$, then $a \pmod{(q_1^{e_1}q_2^{e_2}) \cdot q_3^{e_3}}$, etc., until you have the full product.

The *complexity* of this algorithm will depend on $\ell$, where $\ell$ is the largest prime dividing $(p-1)$, or more precisely the number of basic operations for this algorithm will be a polynomial in $\ell$.

Of course this attack can therefore be thwarted by choosing a prime $p$ such that there is at least one large prime dividing $p-1$.

## 6.2. Digital signatures

In the second half of this lecture we turn back to constructive cryptography, rather than attacks. There are many weird and wonderful things that one can achieve in the world of privacy and security from (public-key) cryptography, most of which we won't get to in this course, but there are two that are fundamental and universal on the internet: key exchange, which we have already covered in some detail, and *digital signatures*.

On an abstract level, a digital signature has the following basic setup: Assume that you, the signer, have already generated a key pair $(sk, pk)$ and published your public key $pk$ as your identity and there is a message $m$ (already in the form of a bit string) that you wish you sign. It is then a basic two-step process:

**Sign:** You use a signing function $(sk, m) \rightsquigarrow sig$ and send the signed message together with your identity $(sig, pk)$ to the verifier.

**Verify:** The verifier uses a verifier function $(sig, pk) \rightsquigarrow m$ to check the signature matches your identity.

Note that unlike message encryption, the important functionality here is that nobody can impersonate the signer: so nobody should be able to compute $sig$ or $sk$ given $pk$ and $m$.

### 6.2.1. ElGamal signatures

We can construct a discrete-logarithm-based signature to fill in these wiggly arrows as follows:

**Setup**

1. Choose a prime $p$ and an element $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ that generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ as a multiplicative group. (Remember, that means that
$$\mathbb{Z}/p\mathbb{Z} - \{0\} = \{g \pmod p, g^2 \pmod p, \ldots, g^{p-1} \pmod p\}.)$$
2. The signer Alice generates a (Diffie-Hellman-style) key pair $(sk, pk) = (a, g^a \pmod p)$, where $a \in [0, p-1]$ is an integer, and publishes $pk$ as her identity.
3. The verifier (or anyone) generates a message $m \pmod{p-1}$ to be signed.

**Sign**

1. Pick a random integer nonce (*number that you use once*) $k \in [0, p-1]$ such that $\gcd(k, p-1) = 1$.
2. Compute $r = g^k \pmod p$.
3. Compute $sig \equiv k^{-1}(m - ar) \pmod{p-1}$.
4. Publish signed message $(r, sig)$.

**Verify**

1. The verifier checks that $g^m \equiv pk^r \cdot r^{sig} \pmod p$.

**Observations**

- Note that the verification step works out just by unrolling all the notation:
$$pk^r \cdot r^{sig} = g^{ar} \cdot g^{k \cdot k^{-1}(m-ar)} = g^m.$$
- Observe that, unlike any of the messages in the other protocols we've seen, we defined our message $m \pmod{p-1}$, not mod $p$. This is because the message appears in the *exponent* in this protocol. Think about when two messages $m$ and $m'$ are equivalent in the verification: that is when $g^m \equiv g^{m'} \pmod p$, which is exactly when $m$ and $m'$ differ by a multiple of $p-1$ (because $g$ has order $p-1$; we'll recall what order means in this context just below). That is, we only need to know the integer $m$ modulo $p-1$.
- Observe that if an attacker knows the nonce $k$, they can recover the secret key $a$ and consequently could imitate the signer, breaking the protocol. Remember that $r, sig, p$, and $m$ are public values, so rearranging the equation defining $sig$ in step 3 of 'Sign' will give the secret key $a$.

- You may be wondering why we use $k$ only once, and not twice (or more times). The reason is that if you use $k$ more than once then the attacker can actually recover $k$ and hence also the secret key $a$ by the previous point. To illustrate this, suppose that you sign messages $m_1$ and $m_2$ using the same nonce $k$. This will give signatures $(r, sig_1)$ and $(r, sig_2)$, where

  $$sig_1 \equiv k^{-1}(m_1 - ar) \pmod{p-1}$$

  and

  $$sig_2 \equiv k^{-1}(m_2 - ar) \pmod{p-1}.$$

  Solving these simultaneous equations then gives

  $$k \equiv \frac{m_1 - m_2}{sig_1 - sig_2} \pmod{p-1},$$

  so never reuse your nonce!

### 6.2.2. RSA signatures

We can also create a factoring-based digital signature scheme as follows:

**Setup**:

1. Signer: Generate an RSA key pair $sk, pk = (d, n), (e, n)$ and publish your identity $(e, n)$.
2. Verifier/anyone: Generate a message $m \pmod{n}$ to be signed.

**Sign**:

1. Compute $sig \equiv m^d \pmod{n}$.
2. Send $(sig, (e, n))$ to the verifier.

**Verify**:

1. Check that $m \equiv sig^e \pmod{n}$.

As before, this is a mathematically consistent scheme because of Fermat's Little Theorem:

$$sig^e \equiv (m^d)^e \equiv m^{de} \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \pmod{n}.$$

Also, the only way to send the signed message corresponding to a given public key $(e, n)$ is to know the secret $(d, n)$, so this scheme is as secure as RSA.

# Lecture 7 – Cryptanalysis of the DLP

## 7.1. Baby-Step-Giant-Step

We have already seen that if you want to find discrete logarithms in $\mathbb{Z}/p\mathbb{Z} - \{0\}$ and $p-1$, the size of the multiplicative group $\mathbb{Z}/p\mathbb{Z} - \{0\}$, has only small factors, you can do this very effectively using Pohlig-Hellman.

However, if you choose a prime $p$ such that there is a large prime $\ell$ dividing $p-1$, then you can also find an element $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ of order $\ell$ where Pohlig-Hellman won't help you. So the question is: can we do better than brute-force? Below we will see two algorithms, Baby-Step-Giant-Step and Pollard-$\rho$, that are essentially clever methods for brute-forcing.

As always, we want to break the discrete logarithm problem, so suppose you have $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ of order $\ell$ (not necessarily prime). Then, given $g$ and $g^a$, find $a$. Remember that changing $a$ by adding a multiple of $\ell$ amounts to multiplying $g^a$ by $g^\ell = 1$, so it suffices to compute $a \pmod{\ell}$.

The algorithm is as follows:

1. For $i$ from 0 to $\lfloor\sqrt{\ell}\rfloor$, compute and save $b_i = g^i$.
2. For $j$ from 0 to $\lfloor\sqrt{\ell}+1\rfloor$, compute $c_j = g^a \cdot g^{-\lfloor\sqrt{\ell}\rfloor \cdot j}$; break if there exists an $i$ such that $c_j = b_i$.
3. Return $a = i + \lfloor\sqrt{\ell}\rfloor \cdot j$.

**Example** Compute $a$ such that $4^a \equiv 83 \pmod{107}$. The order of 4 in $\mathbb{Z}/107\mathbb{Z} - \{0\}$ is equal to 53, so $\ell = 53$ and $\lfloor\sqrt{\ell}\rfloor = 7$. From step 1, the 'baby step', we get a table of values

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $b_i = 4^i \pmod{107}$ | 1 | 4 | 16 | 64 | 42 | 61 | 30 | 13 |

Note that computing this table costs $7 = \lfloor\sqrt{\ell}\rfloor$ multiplications.

For step 2, the 'giant step', $c_0 = g^a$ is easy. For $c_1 = 4^a \cdot 4^{-7}$, we have to compute one inversion and exponentiate to get the 7th power, but we save the values $c_1$ and $4^{-7}$. For $c_2 = 4^a \cdot 4^{-14}$, we first observe that $c_2 = c_1 \cdot 4^{-7}$, and we saved the values $c_1$ and $4^{-7}$, so this just costs one multiplication, as will the computation of every $c_j$ after this by a similar argument. So we get

$$c_0 = 83,$$
$$c_1 = 64$$

at which point we stop because $c_1 = b_3$, or in other words

$$4^a \cdot 3^{-7} \equiv 4^3 \pmod{107},$$

so $a = 10$.

As is illustrated in the example, the baby-step-giant-step algorithm costs at most $2\sqrt{\ell}$ multiplications plus some set up costs for the inversion and exponentiation to the $\lfloor\sqrt{\ell}\rfloor^{\text{th}}$ power (for example using square-and-multiply. As $\ell$ grows, these setup costs become negligible, so we say that 'the complexity of baby-step-giant-step is about $O(\sqrt{\ell})$' – any constants multiplying the $\sqrt{\ell}$ or added to it disappear in the big $O$.

This gives a square root speed up on just brute forcing – if your $\ell$ has 2000 bits then $\sqrt{\ell}$ only has 1000 bits – but at the expense of a pretty serious memory assumption: baby-step-giant-step also requires $O(\sqrt{\ell})$ storage. The next algorithm solves that problem.

## 7.2. Pollard's $\rho$ method

Again, we want to solve the discrete logarithm problem: given $g$ and $g^a \in \mathbb{Z}/p\mathbb{Z} - \{0\}$, find $a$.

The aim of Pollard's $\rho$ method is to output integers $b, c, b', c' \in \{1, \ldots, \ell\}$ such that $c \neq c'$ and

$$g^b (g^a)^c = g^{b'} (g^a)^{c'}. \tag{7.1}$$

Why does this solve our problem? Well, suppose that the order of $g$ in $\mathbb{Z}/p\mathbb{Z} - \{0\}$ is $\ell$. Then, as before, adding $\ell$ to the exponent is equivalent to multiplying by $g^\ell = 1$, so taking logarithms of our equation (7.1) gives us:

$$b + ac \equiv b' + ac' \pmod{\ell}.$$

Rearranging this equation then gives us the secret $a$:

$$a \equiv \frac{b - b'}{c' - c} \pmod{\ell},$$

thus solving the discrete logarithm problem.

So, how exactly do we find such $b, c, b'$, and $c'$? To do this, we define a *graph* $G$ with vertices $G_i \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ such that for each $i$ there exists $b_i$ and $c_i$ such that $G_i = g^{b_i} g^{ac_i}$. We define $G_i$, $b_i$, and $c_i$ iteratively, and once we've found $i \neq j$ with $G_i = G_j$, we have candidates for $b, c, b', c'$ satisfying (7.1), namely $b = b_i$, $c = c_i$, $b' = b_j$, $c' = c_j$.

The iterative sequence that turns out to be the most efficient to do this is:

$$G_0 = g,\ b_0 = 1,\ c_0 = 0,$$

and

$$(G_{i+1}, b_{i+1}, c_{i+1}) = \begin{cases} (G_i \cdot g, b_i + 1, c_i) & G_i \equiv 0 \pmod{3}, \\ (G_i \cdot g^a, b_i, c_i + 1) & G_i \equiv 1 \pmod{3}, \\ (G_i^2, 2b_i, 2c_i) & G_i \equiv 2 \pmod{3}. \end{cases}$$

You might think the $\pmod 3$ looks a bit random, but this is basically just a way of making sure the choice of how to iterate changes around a bit.

**Example.** Compute $a$ such that $3^a \equiv 7 \pmod{17}$. The order of 3 in $\mathbb{F}_{17}$ is 16, so $\ell = 16$. The algorithm above outputs a list:

$$\begin{aligned}
G_0, b_0, c_0 &= 3, 1, 0\\
G_1, b_1, c_1 &= 9, 2, 0\\
G_2, b_2, c_2 &= 10, 3, 0\\
G_3, b_3, c_3 &= 2, 3, 1\\
G_4, b_4, c_4 &= 4, 6, 2\\
G_5, b_5, c_5 &= 11, 6, 3\\
G_6, b_6, c_6 &= 2, 12, 6,
\end{aligned}$$

at which point we terminate because $G_6 = G_3$. In particular, this means that

$$g^{b_6} \cdot (g^a)^{c_6} = g^{b_3} \cdot (g^a)^{c_3},$$

which plugging in the values gives

$$3^3 \cdot (3^a) \equiv 3^{12} \cdot (3^a)^6,$$

giving $a \equiv 9 \cdot (-5)^{-1} \equiv 11 \pmod{16}$.

Pollard's $\rho$ algorithm is expected to terminate after about $\sqrt{\frac{\pi}{2}\ell}$ steps, so also costs $O(\sqrt{\ell})$ multiplications. If we store all the vertices $G_i$'s, we do not improve on the memory requirements of baby-step-giant-step. The trick to avoid incurring this problem is to use Floyd's cycle-finding algorithm, which we are not going to discuss in this lecture.

Both baby-step-giant-step and Pollard $\rho$ are what we refer to as 'generic' algorithms: they're not using anything particular about the structure of the group $\mathbb{Z}/p\mathbb{Z} - \{0\}$ or the choice of $g$ for example–both are essentially just same methods for brute forcing.

In our context, that is in finite fields, there is another algorithm that beats both of these generic algorithms.

## 7.3. Index calculus

Again, we want to solve the discrete logarithm problem: given $g$ and $g^a \in \mathbb{Z}/p\mathbb{Z} - \{0\}$, find $a = \log_g(g^a)$. This is the last algorithm we will see to attack this problem (and also the most efficient in this setting). We will first look at an example and then work out how to write down a general algorithm from that example.

**Example** Suppose you are given that 17 has order 106 in $\mathbb{Z}/107\mathbb{Z} - \{0\}$, and that $17^a \equiv 91 \pmod{107}$, and you want to compute $a$, i.e., you want to compute $\log_{17}(91)$. With the index calculus algorithm, the first thing you do is choose a *factor base* $\mathcal{F}$, which can contain any (and as many) primes (as) you like; here we will choose

$$\mathcal{F} = \{2, 3, 5\}.$$

We then compute $\log_{17}(n)$ for every $n \in \mathcal{F}$ in the following way: compute and factorise $17^i \pmod{107}$ for increasing $i$ until you have found $3 = |\mathcal{F}|$ equations for the $\log_{17}(n)$. In this example, we get

$$17^2 \equiv 3 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$2 \equiv \log_{17}(3) + 2\log_{17}(5) \pmod{106}, \tag{7.2}$$

then $17^3, \dots, 17^8$ all have factors which are not in $\mathcal{F}$, but

$$17^9 \equiv 2^2 \cdot 5 \pmod{107},$$

which taking logs gives

$$9 \equiv 2\log_{17}(2) + \log_{17}(5) \pmod{106}, \tag{7.3}$$

and finally

$$17^{11} \equiv 2 \pmod{107},$$

which taking logs gives

$$11 \equiv \log_{17}(2) \pmod{106}. \tag{7.4}$$

Solving the three simultaneous equations (7.2), (7.3), and (7.4) gives

$$\begin{aligned} \log_{17}(2) &\equiv 11 \pmod{106}, \\ \log_{17}(3) &\equiv 28 \pmod{106}, \\ \log_{17}(5) &\equiv 93 \pmod{106}. \end{aligned}$$

So now we've found these values, what do we do with them? We want to be able to write the discrete log we're actually interested in, namely $\log_{17}(91)$, in terms of the discrete logs we now know, namely $\log_{17}(n)$ for $n \in \mathcal{F}$, and of course $\log_{17}(17^j)$ $(= j)$ for small values of $j$. We can play the same game as above: try multiplying 91 with $17^j$ for small values of $j$ and factorizing until we find a number with only factors from our factor base. Doing this we see that $17^0 \cdot 91, \dots, 17^4 \cdot 91$ yields nothing but

$$17^5 \cdot 91 \equiv 2^2 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$5 + \log_{17}(91) \equiv 2\log_{17}(2) + 2\log_{17}(5) \pmod{106},$$

and plugging in the values above this gives us that

$$a = \log_{17}(91) = 97.$$

So, let's summarize our method into a more general algorithm. Suppose you are given $g$ and $g^a \in \mathbb{F}_p$ and you want to compute $a$. Then

1. Choose your factor base $\mathcal{F} = \{p_1, \ldots, p_n\}$.
2. Compute, for each $i = 1, \ldots, n$, the value $\log_g(p_i)$:
   (a) For increasing $j \geq 1$, factorise $g^j$. Break when you have found $n$ values of $j$ for which all the factors of $g^j$ are in $\mathcal{F}$.
   (b) Take logs of the $n$ equations for values $g^j$ with all factors in $\mathcal{F}$ to get $n$ simultaneous equations for $\log_g(p_1), \ldots, \log_g(p_n)$.
   (c) Solve your $n$ simultaneous equations to get $\log_g(p_1), \ldots, \log_g(p_n)$.
3. For increasing $j \geq 0$, factoring $g^j \cdot g^a$. Break when all factors of $j$ are in $\mathcal{F}$.
4. Take logs of the equation from the previous step, and solve for $a$.

This algorithm is by far the most efficient known algorithm for this setting of the discrete logarithm problem. To write down the complexity, we recall the notation:

$$L_N(\alpha, c) = e^{c \log N^{\alpha} \log\log N^{1-\alpha}},$$

where $\alpha \in [0, 1]$ Recall also that the closer $\alpha$ is to 0, the closer an algorithm is to being polynomial time, and the closer it is to 1, the closer an algorithm is to being exponential time.

The most optimized version of the index calculus algorithm (containing many many details not covered here) has complexity $L_p(1/3, c)$, where the constant $c$ depends very heavily on the conditions, so that's closer to the polynomial time end than the exponential end, but the different is still (asymptotically) big enough for powers of large primes that it is possible to make use of finite fields in cryptography by scaling up the numbers. In particular, scaling up $p$ to at least 3000 bits for 128-bit security, meaning that it should take about $2^{128}$ bit operations to break the protocol. Compare this to our Pollard $\rho$ algorithm which takes about $\sqrt{p}$ bit operations to break the protocol–so we would need $p$ to be about 256 bits, and you see how much different the index calculus makes.

There are other examples of groups in which the index calculus is less or not at all effective. These groups, namely elliptic curve groups, are currently the most commonly used in practise–the size of your group can indeed be only about 256 bits instead of 3000. But that is beyond the scope of this course!

# Lecture A – Cryptography in Finite Fields

These notes are **Additional Content**, and are only intended for students going for the 90-100 range. Students who prefer to skip the additional content should skip these notes.

## A.1. Finite fields

The Pohlig-Hellman attack on the discrete logarithm problem in $\mathbb{F}_p^*$, for $p$ prime, can be thwarted by choosing a prime $p$ such that there is at least one large prime dividing $p - 1$.

But, this is a bit of a problem for the ideas we had for efficient computations mod $p$: remember we were also using Sun-Tzu's Remainder Theorem to make our computations more efficient for encryption.

So, we need a bit more choice in how to set up our cryptosystems: Instead of just using exponentiation mod $p$ we can use exponetiation in some more general contexts. To figure out which contexts will work, we let's first enumerate what we're actually using in for example Diffie-Hellman and ElGamal.

- We need to be able to exponentiate efficiently.
- We need to be able to multiply and add elements together (efficiently).
- We need elements to have inverses that we can compute.
- We need an element (we've been calling it $g$) of finite order.

The first three properties in the list are all true in any *field*, and the last property will hold if our field is finite. So, instead of just using integers mod $p$, we want to extend our cryptographic algorithms to be for more general finite fields. What are these exactly?

**Definition A.1.** A set $k$ is a *field* with respect to binary operations

$$\cdot : k \times k \to k$$

and

$$+ : k \times k \to k$$

if the following axioms are satisfied:

(F1) $(k, +)$ is an abelian group.

(F2) $(k - \{0\}, \cdot)$ is an abelian group.

(F3) For every $a, b \in k$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

**Examples**

- $\mathbb{Z}/p\mathbb{Z}$ for $p$ prime.
- $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$.

**Non-examples**

- $\mathbb{Z}$ (e.g. $2$ has no multiplicative inverse).
- $\mathbb{Z}/4\mathbb{Z}$ (e.g. multiplication is not a binary operation on $\mathbb{Z}/4\mathbb{Z} - \{0\}$: $2 \cdot 2 \equiv 0 \pmod 4 \notin (\mathbb{Z}/4\mathbb{Z}) - \{0\}$.)
- $\mathbb{Z}/n\mathbb{Z}$ for composite $n$. (Try to extend the reasoning for $\mathbb{Z}/4\mathbb{Z}$ to this case).

If we look at our list of examples above, given that we need a field to be finite, we're left with only $\mathbb{Z}/p\mathbb{Z}$, that we were using already. So how do we construct more examples? To see this, consider for a moment how you first constructed $\mathbb{C}$ from $\mathbb{R}$.

$$\mathbb{C} = \mathbb{R} + i\mathbb{R} = \{a + ib : a, b \in \mathbb{R}\},$$

where is abstractly defined as a number such that $i^2 + 1 = 0$.

We can use the same trick to construct extensions of the fields $\mathbb{Z}/p\mathbb{Z}$. We first see an example.

**Example**. Define

$$(\mathbb{Z}/2\mathbb{Z}) + \alpha(\mathbb{Z}/2\mathbb{Z}) = \{n + \alpha m : n, m \in \mathbb{Z}/2\mathbb{Z}\},$$

where $\alpha^2 + \alpha + 1 = 0$. This set contains four elements:

$$(\mathbb{Z}/2\mathbb{Z}) + \alpha(\mathbb{Z}/2\mathbb{Z}) = \{0, 1, \alpha, 1 + \alpha\},$$

and we claim that it is a field. Let us first write out an addition table to see why it is an additive group.

| + | 0 | 1 | $\alpha$ | $1+\alpha$ |
|---|---|---|---|---|
| 0 | 0 | 1 | $\alpha$ | $1+\alpha$ |
| 1 | 1 | 0 | $1+\alpha$ | $\alpha$ |
| $\alpha$ | $\alpha$ | $1+\alpha$ | 0 | 1 |
| $1+\alpha$ | $1+\alpha$ | $\alpha$ | 1 | 0 |

From this table we can read off the desired (nonobvious) group properties: the sum of any two elements lands back in the desired set, every element has an additive inverse (since every element has a 0 in its column), and it is abelian since the table is symmetric about the diagonal.

Now we do the same to check that $((\mathbb{Z}/2Z) + \alpha(\mathbb{Z}/2\mathbb{Z})) - \{0\}$ is a multiplicative group.

| $\cdot$ | $1$ | $\alpha$ | $\alpha+1$ |
|---|---|---|---|
| $1$ | $1$ | $\alpha$ | $\alpha+1$ |
| $\alpha$ | $\alpha$ | $1+\alpha$ | $1$ |
| $1+\alpha$ | $1+\alpha$ | $1$ | $\alpha$ |

Again, from this table we can read off the desired (nonobvious) group properties: the product of any two nonzero elements lands in the set of nonzero elements, every element has a multiplicative inverse (since every element has a 1 in its column), and it is abelian since the table is symmetric about the diagonal.

You can also check distributivity (F3) but we leave that as an exercise. So here we have a field with 4 elements. It is certainly not the same thing as $\mathbb{Z}/4\mathbb{Z}$ since that is not a field, so we have successfully constructed a new field. We can construct more examples by including higher degrees of $\alpha$ that satisfy different polynomials, and of course using different primes $p$. However, such a construction won't always work, let's see an example where this goes wrong.

**Non-example** Let

$$L = \mathbb{Z}/2\mathbb{Z} + \alpha\mathbb{Z}/2\mathbb{Z} + \alpha^2\mathbb{Z}/2\mathbb{Z} + \alpha^3\mathbb{Z}/2\mathbb{Z} = \{a + b\alpha + c\alpha^2 + d\alpha^3 : a, b, c, d \in \mathbb{Z}/2\mathbb{Z}\},$$

where $\alpha^4 + \alpha^2 + 1$. This set has 16 elements:

$$\begin{aligned} L = \{&0, 1, \alpha, 1+\alpha \\ &\alpha^2, \alpha^2+1, \alpha^2+\alpha, \alpha^2+\alpha+1 \\ &\alpha^3, \alpha^3+1, \alpha^3+\alpha, \alpha^3+\alpha+1 \\ &\alpha^3+\alpha^2, \alpha^3+\alpha^2+1, \alpha^3+\alpha^2+\alpha, \alpha^3+\alpha^2+\alpha+1\}. \end{aligned}$$

In this case $L - \{0\}$ is *not* a multiplicative group, since for example

$$(1+\alpha+\alpha^2)\cdot(1+\alpha+\alpha^2) \equiv 1+\alpha^2+\alpha^4 = 0 \pmod 2 \notin L - \{0\}.$$

What goes wrong in this example that didn't go wrong for the first example? The problem is our defining equation $\alpha^4+\alpha^2+1$ can be factorized, giving nonzero elements that can be multiplied to give 0; these elements will also not have multiplicative inverses. To make this more formal we first introduce some notation.

**Notation** We notate the set $\mathbb{Z}/p\mathbb{Z} + \alpha\mathbb{Z}/p\mathbb{Z} + \cdots + \alpha^{n-1}\mathbb{Z}/p\mathbb{Z}$, where $\alpha$ is a root of the degree $n$ polynomial $f(x) \in \mathbb{Z}/p\mathbb{Z}[x]$, by

$$(\mathbb{Z}/p\mathbb{Z})[x]/(f(x)).$$

(Recall: a degree $n$ polynomial $f(x)$ with coefficients in $\mathbb{Z}/p\mathbb{Z}$ is given by $f(x) = c_nx^n + c_{n-1}x^{n-1} + \cdots + c_0$, where the coefficients $c_i$ are integers mod $p$ and $c_n \neq 0$.)

In this notation, our first example above would be written

$$(\mathbb{Z}/2\mathbb{Z})[x]/(x^2+x+1).$$

To check if something is a field, we can use the following theorem:

**Theorem A.1.** *Let $p$ be a prime and $f(x) \in (\mathbb{Z}/p\mathbb{Z})[x]$ a polynomial. Then $(\mathbb{Z}/p\mathbb{Z})[x]/(f(x))$ is a field if and only if $f(x)$ is irreducible. It has $p^{deg(f)}$ elements.*

This theorem is actually part of a bigger theorem called the *classification of finite fields*, which we won't go into, but just for your enjoyment, here are some more facts about this construction:

- Every finite field is of the form given in the theorem above.
- For every prime $p$ and $n \in \mathbb{Z}_{>0}$ there exists a finite field of order $p^n$, and it is unique up to isomorphism.

This (semi) uniqueness of a finite field of a certain order hopefully motivates the following notation.

**Notation** A finite field of order $p^n$ is denoted by $\mathbb{F}_{p^n}$. The *multiplicative group* $\mathbb{F}_{p^n} - \{0\}$ associated to such a field is denoted by $\mathbb{F}^*_{p^n}$.

So, when confronted with the notation $\mathbb{F}_{p^n}$ and asked to do calculations in that field, your first thought should be: which irreducible degree $n$ polynomial $f(x)$ defines this field? Once you know that, you can write down elements of your field and do calculations with them.

Going back to the use of finite fields in cryptography, you can hopefully see now that you can get a large computation space even with small primes if your take your $n$ to be very large: even $\mathbb{F}_{2^n}$ can work if $n$ is large enough. We can then take a $g \in \mathbb{F}^*_{2^n}$ of large prime order and perform our Diffie-Hellman with this $g$. Because $\mathbb{F}_{2^n}$ can be viewed as a vector space over $\mathbb{F}_2$, you then can do a lot of your additions just in $\mathbb{Z}/2\mathbb{Z}$ in parallel and this speeds up the computations massively. However, it turns out if you choosen small $p$ (e.g. $p = 2$) and large $n$, then index calculus is extra effective – in fact in this particular instance it is polynomial time!

# Lecture B – More on Reductions

This bonus chapter expands a little[1] on reductions, explaining not just what they look like, but how to come up with them in the first place. We will do this based on examples that will be discussed in depth. *In an assessment setting, there is no need to discuss* how *you get to a reduction in depth: you only need to give the reduction and analyse it, as done in the main body of Chapter 2.* (Doing that both correctly and intelligibly is convincing enough for publication—on more complex examples—in leading cryptography research venues, so we use it as a solid approximation of your understanding of and ability to apply the methods we mean to assess.)

We will start with the reductions shown in the main lecture notes.

## B.1. KR-1CPA ⇒ KR-1KPA

In this instance, the security goal (hardness of key recovery) stays the same. For the purpose of the reduction, this means that the *shape* of the security experiment is also the same, and the only thing that changes is which oracles the adversary has access to. Let's expand on this a bit, re-framing security definitions slightly.

Figure B.1 presents the definitions of key recovery under one-time chosen plaintext attacks and under one-time known plaintext attack in an oracle-free style (for clarity, and hopefully building more solid understanding). Side by side, it is obvious that the only thing that changes is that the CPA adversary has more *control* over the generation of the challenge ciphertext.

```
$\mathsf{Exp}_E^{\mathsf{kr\text{-}1cpa}}(\mathbb{B})$
    $k^* \leftarrow_\$ E.\mathsf{Kg}$
    $m^* \leftarrow_\$ \mathbb{B}_1()$
    $c^* \leftarrow E.\mathsf{Enc}_k(n, m)$
    $\widehat{k} \leftarrow_\$ \mathbb{B}_2(c^*)$
```

```
$\mathsf{Exp}_E^{\mathsf{kr\text{-}1kpa}}(\mathbb{A})$
    $k^* \leftarrow_\$ E.\mathsf{Kg}$
    $m^* \leftarrow_\$ m$
    $c^* \leftarrow E.\mathsf{Enc}_k(n, m)$
    $\widehat{k} \leftarrow_\$ \mathbb{A}(c^*)$
```

Figure B.1.: Experiments defining KR-1CPA and KR-1KPA.

### B.1.1. Reduction

Now recall the contrapositive reasoning for this reduction: to prove that any scheme that is $(t', \epsilon')$-KR-1CPA secure is also $(t, \epsilon)$-KR-1KPA secure, we write a reduction that turns any

[1] lol

```
$\mathbb{B}_1^R()$                        $\mathbb{B}_2^R(c^*$
$m \leftarrow_\$ \mathcal{M}$             $\widehat{k} \leftarrow_\$ \mathbb{A}(m, c^*)$
return $m$                                return $\widehat{k}$
```

Figure B.2.: A reduction from KR-1KPA to KR-1CPA.

$(t, \epsilon)$-KR-1KPA adversary into a $(t', \epsilon')$-KR-1CPA adversary. In short, and ignoring the resource and advantage constraints for now—they are really outputs of the reduction, we need to turn an adversary (say, $\mathbb{A}$) that wins *without* choosing the challenge plaintext into one (say, $\mathbb{B}$) that wins by choosing the challenge plaintext. To come up with the reduction in the first place, the parameters (unless they are given concretely!) can safely be ignored, keeping in mind that our goal is to keep the running time and advantage of $\mathbb{B}$ as close as possible to those of $\mathbb{A}$.

In constructing this reduction, *we* decide how $\mathbb{B}$ chooses its challenge plaintext. It seems an obvious first step to consider what happens if we make it choose its challenge plaintext in the exact same way that the KPA oracle chooses the challenge plaintext. This gives us the definition of $\mathbb{B}$ shown in Figure B.2.

### B.1.2. Analysis

All that remains to do is analyse this reduction: consider its running time $t$ (as a function of the running time $t'$ of $\mathbb{A}$ and its advantage $\epsilon$ (as a function of the advantage $\epsilon'$ of $\mathbb{A}$.

**Running time.** In the experiment, $\mathbb{B}_1^R$ runs once—simply sampling a message uniformly at random in $\mathcal{M}$, then $\mathbb{B}_2^R$ runs once, and simply runs $\mathbb{A}$. So if $\mathbb{A}$ runs in time at most $t$, then $\mathbb{B}^R$ runs in time at most $t' = t + t_{\mathcal{M}}$, where $t_{\mathcal{M}}$ is the cost of sampling a message uniformly at random.

**Advantage.** Consider the programme $\mathsf{Exp}_E^{\text{kr-1cpa}}(\mathbb{B}^R)$, and inline the code of $\mathbb{B}_1^R$ and $\mathbb{B}_2^R$. The result is exactly $\mathsf{Exp}_E^{\text{kr-1kpa}}(\mathbb{A})$ and we therefore have $\mathsf{Adv}_E^{\text{kr-1cpa}}(\mathbb{B}^R) = \mathsf{Adv}_E^{\text{kr-1kpa}}(\mathbb{A})$.

As a consequence, for any scheme $E$, if all KR-1CPA adversaries running in time at most $t$ have a KR-1CPA advantage at most $\epsilon$ against $E$, then all KR-1KPA adversaries running in time at most $t - t_{\mathcal{M}}$ have a KR-1KPA advantage at most $\epsilon$ against $E$.

## B.2. KR-CPA $\Rightarrow$ KR-KPA

With the above understood, we can now move on to a similar proof, but with oracles: the security goal (security against key recovery) is once again fixed, and we once again want to show that any scheme that is secure against a particular kind of adversary is also secure against an adversary that has *less* power. We once again consider a CPA adversary as our "strong" baddie, and a KPA adversary as our "weak" baddie, but this time we allow them to ask for multiple encipherings, moving to an oracle-based definition.

Figure B.3 presents the definitions of key recovery under chosen plaintext attacks and under known plaintext attack. As discussed in lectures, note that the "shape" of the experiment and advantage is determined by the security goal (key recovery here), whereas the adversary's capabilities (CPA or KPA) only changes the kind of oracles the adversary can interact with. [2]

Here again, seeing these side-by-side should make it clear that the CPA adversary has more control over their interaction with the key (via their oracles): they get to pick the plaintexts that get enciphered by the oracle instead of simply seeing them.

```
$\mathsf{Exp}_E^{\mathsf{kr\text{-}cpa}}(\mathbb{B})$
    $k^* \leftarrow_\$ E.\mathsf{Kg}$
    $\widehat{k} \leftarrow_\$ \mathbb{B}^{\mathcal{E}_{\mathsf{cpa}}(\cdot)}()$
```

```
$\mathcal{E}_{\mathsf{cpa}}(m)$
    $c \leftarrow E.\mathsf{Enc}_{k^*}(m)$
    return $c$
```

```
$\mathsf{Exp}_E^{\mathsf{kr\text{-}kpa}}(\mathbb{A})$
    $k^* \leftarrow_\$ E.\mathsf{Kg}$
    $\widehat{k} \leftarrow_\$ \mathbb{A}^{\mathcal{E}_{\mathsf{kpa}}()}()$
```

```
$\mathcal{E}_{\mathsf{kpa}}()$
    $m \leftarrow_\$ \mathcal{M}$
    $c \leftarrow E.\mathsf{Enc}_{k^*}(m)$
    return $(m, c)$
```

Figure B.3.: Experiments defining KR-CPA and KR-KPA.

### B.2.1. Reduction

As before, recall the contrapositive reasoning for this reduction: to prove that any scheme that is $(t', q, \epsilon')$-KR-CPA secure is also $(t, q, \epsilon)$-KR-KPA secure, we write a reduction that turns any $(t, q, \epsilon)$-KR-KPA adversary into a $(t', q, \epsilon')$-KR-CPA adversary. In short, and ignoring the resource and advantage constraints for now—they are really outputs of the reduction, we need to turn an adversary (say, $\mathbb{A}$) that wins *without* getting to choose the messages whose encipherings they get to observe into one (say, $\mathbb{B}$) that wins by picking those plaintexts.

Here, there is a bit of a shift in thinking that needs to happen: it is not just about figuring out how to take the inputs of $\mathbb{B}$ and turn them into well-chosen inputs for $\mathbb{A}$ to produce something that we can use in crafting a solution for the problem $\mathbb{B}$ is trying to solve. Now we are in a setting where the problem $\mathbb{A}$ and $\mathbb{B}$ (including all of their inputs and outputs!) are actually the same, but they get to *interact* with different oracles: $\mathbb{A}$ expects to interact with a black box that reads no inputs (just a signal to go) and spits out a plaintext and ciphertext as outputs; $\mathbb{B}$ expects to interact with a black box that reads a plaintext and spits out a ciphertext.

[2] Formally defining oracles is a long and arduous process that requires either properly defining interactive Turing machines or properly defining an imperative probabilistic programming language with procedure calls… You should think of them as black boxes with a slot big enough for a piece of paper: the adversary writes some inputs on a piece of paper (this costs the adversary some time), slips it in the slot (this is free), and immediately gets back a new piece of paper with the answer written down (reading it costs the adversary some time). The answer is derived (instantly, and without cost to the adversary) following the oracle's specification.)

```
$\mathbb{B}_0^{\mathcal{E}_{\mathsf{cpa}}(\cdot)}()$
    $k \leftarrow_\$ \mathbb{A}^{\mathcal{E}_R()}()$
    return $k$

$\mathcal{E}_R()$
    $m \leftarrow_\$ \mathcal{M}$
    $c \leftarrow \mathcal{E}_{\mathsf{cpa}}(m)$
    return $c$
```

Figure B.4.: A reduction from KR-KPA to KR-CPA.

So if $\mathbb{B}$ wants to be able to run (or simulate) $\mathbb{A}$, she needs to use her own inputs and her own oracle to present $\mathbb{A}$ with a black box that behaves as he expects. In other words, $\mathbb{B}$ has a box that expects an input and produces one output, and needs to present $\mathbb{A}$ with a box that takes no inputs and produces two outputs. Here again, the solution is simple: when $\mathbb{A}$ feeds a blank piece of paper to its oracle (which $\mathbb{B}$ is now implementing), $\mathbb{B}$ picks an input however she wants, then feeds it to her own black box, and writes both the input she picked and the result she got to the piece of paper that goes out to $\mathbb{A}$.

### B.2.2. Analysis

All that remains to do is analyse this reduction: consider its running time $t$ (as a function of the running time $t'$ of $\mathbb{A}$ and its advantage $\epsilon$ (as a function of the advantage $\epsilon'$ of $\mathbb{A}$.

**Running time and query count.** $\mathbb{B}_0$ runs $\mathbb{A}$ once and simply forwards its output. For each query $\mathbb{A}$ makes to its KPA oracle, $\mathbb{B}_0$ samples a message and makes a query to its CPA oracle. So, if $\mathbb{A}$ runs in time at most $t$ and makes at most $q$ queries, then $\mathbb{B}_0$ runs in time at most $t' = t + q \cdot t_{\mathcal{M}}$, where $t_{\mathcal{M}}$ is the cost of sampling a message uniformly at random, and makes at most $q$ oracle queries.

**Advantage.** Consider the programme $\mathsf{Exp}_E^{\mathsf{kr\text{-}cpa}}(\mathbb{B}_0)$, and inline the code of $\mathbb{B}_0$ and of the CPA oracle. The result is exactly (up to inlining artefacts) $\mathsf{Exp}_E^{\mathsf{kr\text{-}kpa}}(\mathbb{A})$ and we therefore have $\mathsf{Adv}_E^{\mathsf{kr\text{-}}cpa}(\mathbb{B}_0) = \mathsf{Adv}_E^{\mathsf{kr\text{-}kpa}}(\mathbb{A})$.

As a consequence, for any scheme $E$, if all KR-CPA adversaries running in time at most $t$ and making at most $q$ CPA queries have a KR-CPA advantage at most $\epsilon$ against $E$, then all KR-KPA adversaries running in time at most $t - q \cdot t_{\mathcal{M}}$ and making at most $q$ KPA queries have a KR-KPA advantage at most $\epsilon$ against $E$.

## B.3. (N)IND-CPA ⇒ (IV)IND-CPA

We now consider the generic proof that any encryption scheme that is nonce-based secure is also IV-based secure. (See Problem Sheet 2.) There are obviously some issues that arise here that we have not yet had to consider: here, the very syntax of the schemes we consider in the security definitions is different—so there are slight differences that appear in both the shape of the experiments and the input and output types of the oracles despite their similarities.

In addition, as we will see, the reduction here is not perfect: there are some failures of the encryption scheme as an IV-based encryption scheme that cannot be adequately explained by its failures as a nonce-based encryption scheme. This implies a small *security loss*—our goal is typically to keep it as small as possible. When we can exhibit an attack that succeeds with exactly the security loss, we say that the reduction is *tight*—this is not important in this unit but is an important concept in giving reductions a practical meaning.

TBC

## B.4. (N)IND-CPA security of CTR

To finish this note off, we consider the security of counter mode (CTR) as a nonce-based encryption scheme.

TBC

# Bibliography

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:161–191, 2 1883.

[Sha49] Claude Elwood Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949.