

# Applied Cryptography

or: The Unofficial Notes on the Georgia Institute  
of Technology's **CS6260**: *Applied Cryptography*



George Kudrayvtsev  
[george.k@gatech.edu](mailto:george.k@gatech.edu)

Last Updated: June 3, 2020

Creation of this guide was powered entirely by caffeine in its many forms.  
☞ If you found it useful and are generously looking to fuel my stimulant  
addiction, feel free to shoot me a donation on Venmo [@george\\_k\\_btw](#) or  
PayPal [kudrayvtsev@sbcglobal.net](mailto:kudrayvtsev@sbcglobal.net) with whatever this guide was worth  
to you.

If I've shared a class with you, I might've made a guide for it as well; check  
out my [other notes](#)!

Happy studying! ☺

*Those who would give up essential Liberty, to purchase a little temporary Safety,  
deserve neither Liberty nor Safety.*

— Benjamin Franklin

<b>I</b>	<b>Symmetric Cryptography</b>	<b>5</b>
<b>0</b>	<b>Perfect Security</b>	<b>6</b>
0.1	Notation & Syntax . . . . .	6
0.2	One-Time Pads . . . . .	6
0.2.1	The Beauty of XOR . . . . .	7
0.2.2	Proving Security . . . . .	8
<b>1</b>	<b>Block Ciphers</b>	<b>10</b>
1.1	Modes of Operation . . . . .	10
1.1.1	ECB—Electronic Code Book . . . . .	11
1.1.2	CBC—Cipher-Block Chaining . . . . .	11
1.1.3	CBCC—Cipher-Block Chaining with Counter . . . . .	12
1.1.4	CTR—Randomized Counter Mode . . . . .	12
1.1.5	CTRC—Stateful Counter Mode . . . . .	13
1.2	Security Evaluation . . . . .	13
1.2.1	IND-CPA: Indistinguishability Under Chosen-Plaintext Attacks . . . . .	14
1.2.2	IND-CPA-cg: A Chosen Guess . . . . .	16
1.2.3	What Makes Block Ciphers Secure? . . . . .	17
1.2.4	Random Functions . . . . .	18
1.2.5	IND-CCA: Indistinguishability Under Chosen-Ciphertext Attacks . . . . .	23
1.3	Summary . . . . .	25
<b>2</b>	<b>Message Authentication Codes</b>	<b>26</b>
2.1	Notation & Syntax . . . . .	26
2.2	Security Evaluation . . . . .	27
2.2.1	UF-CMA: Unforgeability Under Chosen-Message Attacks . . . . .	27
2.3	Mode of Operation: CBC-MAC . . . . .	28
<b>3</b>	<b>Hash Functions</b>	<b>30</b>
3.1	Collision Resistance . . . . .	30
3.2	Building Hash Functions . . . . .	32
3.3	One-Way Functions . . . . .	34
3.4	Hash-Based MACs . . . . .	35
<b>4</b>	<b>Authenticated Encryption</b>	<b>37</b>
4.1	INT-CTXT: Integrity of Ciphertexts . . . . .	37
4.2	Generic Composite Schemes . . . . .	38
4.2.1	Encrypt-and-MAC . . . . .	38
4.2.2	MAC-then-encrypt . . . . .	39
4.2.3	Encrypt-then-MAC . . . . .	39
4.2.4	In Practice... . . . .	40
4.2.5	Dedicated Authenticated Encryption . . . . .	41
4.3	AEAD: Associated Data . . . . .	41
4.3.1	GCM: Galois/Counter Mode . . . . .	41
<b>5</b>	<b>Stream Ciphers</b>	<b>42</b>
5.1	Generators . . . . .	42
5.1.1	PRGs for Encryption . . . . .	42
5.2	Evaluating PRGs . . . . .	42
5.3	Creating Stream Ciphers . . . . .	43
5.3.1	Forward Security . . . . .	44
5.3.2	Considerations . . . . .	45

<b>6</b>	<b>Common Implementation Mistakes</b>	<b>46</b>
<b>II</b>	<b>Asymmetric Cryptography</b>	<b>48</b>
<b>7</b>	<b>Overview</b>	<b>49</b>
7.1	Notation . . . . .	49
7.2	Security Definitions . . . . .	49
<b>8</b>	<b>Number Theory</b>	<b>51</b>
8.1	Groups . . . . .	51
8.2	Modular Arithmetic . . . . .	53
8.2.1	Running Time . . . . .	53
8.2.2	Inverses . . . . .	53
8.2.3	Modular Exponentiation . . . . .	54
8.3	Groups for Cryptography . . . . .	54
8.3.1	Discrete Logarithm . . . . .	55
8.3.2	Constructing Cyclic Groups . . . . .	56
8.4	Modular Square Roots . . . . .	57
8.4.1	Square Groups . . . . .	58
8.4.2	Square Root Extraction . . . . .	59
8.5	Chinese Remainder Theorem . . . . .	60
<b>9</b>	<b>Encryption</b>	<b>61</b>
9.1	Recall: The Discrete Logarithm . . . . .	61
9.1.1	Formalization . . . . .	61
9.1.2	Difficulty . . . . .	62
9.2	Diffie-Hellman Key Exchange . . . . .	64
9.3	ElGamal Encryption . . . . .	64
9.3.1	Security: IND-CPA . . . . .	65
9.3.2	Security: IND-CCA . . . . .	67
9.4	Cramer-Shoup Encryption . . . . .	67
9.5	RSA Encryption . . . . .	68
9.5.1	Protocol . . . . .	69
9.5.2	Limitations . . . . .	70
9.5.3	Securing RSA . . . . .	71
9.6	Hybrid Encryption . . . . .	72
9.7	Multi-User Encryption . . . . .	72
9.7.1	Security Definitions . . . . .	72
9.7.2	Security Evaluation . . . . .	73
9.8	Scheme Variants . . . . .	74
<b>10</b>	<b>Digital Signatures</b>	<b>75</b>
10.1	Security Definitions . . . . .	75
10.2	RSA . . . . .	76
10.2.1	Plain RSA . . . . .	76
10.2.2	Full-Domain Hash RSA . . . . .	76
10.2.3	Probabilistic Signature Scheme . . . . .	78
10.3	ElGamal . . . . .	78
10.4	Digital Signature Algorithm . . . . .	79
10.5	Schnorr Signatures . . . . .	80
10.6	Scheme Variants . . . . .	81
10.6.1	Simple Multi-Signature Scheme . . . . .	81

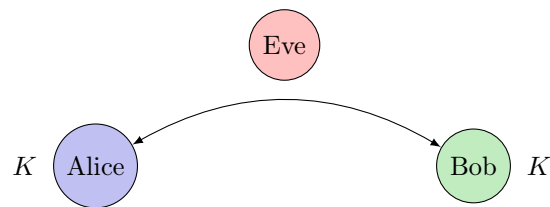
10.6.2 Simple Blind Signature Scheme . . . . .	82
10.7 Signcryption . . . . .	82
<b>11 Secret Sharing</b>	<b>84</b>
11.1 Public Key Infrastructure . . . . .	84
11.1.1 Registering Users . . . . .	84
11.1.2 Revocation . . . . .	85
11.1.3 Scaling . . . . .	85
11.1.4 An Alternative: Pretty Good Privacy . . . . .	85
11.2 Secret Sharding . . . . .	86
11.2.1 Shamir's Secret Sharing . . . . .	86
11.2.2 Threshold Schemes . . . . .	87
11.3 Man-in-the-Middle Attacks . . . . .	87
11.4 Passwords . . . . .	88
 <b>Index of Terms</b>	 <b>90</b>

# PART I

---

## SYMMETRIC CRYPTOGRAPHY

The notion of **symmetric keys** comes from the fact that both the sender and receiver of encrypted information share the same secret key,  $K$ . This secret is the only thing that separates a viable receiver from an attacker.



Symmetric key algorithms are often very efficient and supported by hardware, but their fatal flaw lies in **key distribution**. If two parties need to share a secret without anyone else knowing, how do they get it to each other without already having a secure channel? That's the job of **Part II**: asymmetric cryptography.

### Contents

<b>0</b>	<b>Perfect Security</b>	<b>6</b>
<b>1</b>	<b>Block Ciphers</b>	<b>10</b>
<b>2</b>	<b>Message Authentication Codes</b>	<b>26</b>
<b>3</b>	<b>Hash Functions</b>	<b>30</b>
<b>4</b>	<b>Authenticated Encryption</b>	<b>37</b>
<b>5</b>	<b>Stream Ciphers</b>	<b>42</b>
<b>6</b>	<b>Common Implementation Mistakes</b>	<b>46</b>

# PERFECT SECURITY

As mentioned in the ??, algorithms in symmetric cryptography rely on all members having a shared secret. Let's cover the basic notation we'll be using to describe our schemes and then dive into some.

## 0.1 Notation & Syntax

For consistency, we'll need common conventions when referring to cryptographic primitives in a scheme.

A well-described symmetric encryption scheme covers the following:

- a **message space**, denoted as the  $\text{MsgSp}$  or  $\mathcal{M}$  for short, describes the set of things which can be encrypted. This is typically unrestricted and (especially in the context of the digital world) includes anything that can be encoded as bytes.
- a **key generation algorithm**,  $\mathcal{K}$ , or the key space spanned by that algorithm,  $\text{KeySp}$ , describes the set of possible keys for the scheme and how they are created. The space typically restricts its members to a specific length.
- a **encryption algorithm** and its corresponding **decryption algorithm** ( $\mathcal{E}, \mathcal{D}$ ) describe how a message  $m \in \mathcal{M}$  is converted to and from ciphertext. We will use the notation  $\mathcal{E}(K, M)$  and  $\mathcal{E}_K(M)$  interchangeably to indicate encrypting  $M$  using the key  $K$  (and similarly for  $\mathcal{D}$ ).

A well-formed scheme *must* allow all valid messages to be en/decrypted. Formally, this means:

$$\forall m \in \text{MsgSp}, \forall k \in \text{KeySp} : \mathcal{D}(k, \mathcal{E}(k, m)) = m$$

An encryption scheme defines the message space and three algorithms:  $(\text{MsgSp}, \mathcal{E}, \mathcal{D}, \mathcal{K})$ . The key generation algorithm often just pulls a random  $n$ -bit string from the entire  $\{0, 1\}^n$  bit space; to describe this action, we use notation  $K \xleftarrow{\$} \text{KeySp}$ . The encryption algorithm is often randomized (taking random input in addition to  $(K, M)$ ) and stateful. We'll see deeper examples of all of these shortly.

## 0.2 One-Time Pads

A **one-time pad** (or OTP) is a very basic and simple way to ensure absolutely perfect encryption, and it hinges on a fundamental binary operator that will be at the heart of many of our symmetric encryption schemes in the future: **exclusive-or**.

Messages are encrypted with an equally-long sequence of random bits using XOR. With our notation, one-time pads can be described as such:

- the key space is all  $n$ -bit strings:  $\text{KeySp} = \{0, 1\}^n$
- the message space is the same:  $\text{MsgSp} = \{0, 1\}^n$
- encryption and decryption are just XOR:

$$\mathcal{E}(K, M) = M \oplus K$$

$$\mathcal{D}(K, C) = C \oplus K$$

Can we be a little more specific with this notion of “perfect encryption”? Intuitively, a secure encryption scheme should reveal nothing to adversaries who have access to the ciphertext. Formally, this notion is called being Shannon-secure (and is also referred to as **perfect security**): the probability of a ciphertext occurring should be equal for any two messages.

### DEFINITION 0.1: Shannon-secure

An encryption scheme,  $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ , is **Shannon-secure** if:

$$\begin{aligned} &\forall m_1, m_2 \in \text{MsgSp}, \forall C : \\ &\Pr[\mathcal{E}(K, m_1) = C] = \Pr[\mathcal{E}(K, m_2) = C] \end{aligned} \quad (1)$$

That is, the probability of a ciphertext  $C$  must be equally-likely for any two messages that are run through  $\mathcal{E}$ .

Note that this doesn’t just mean that a ciphertext occurs with equal probability for a *particular* message, but rather than *any* message can map to *any* ciphertext with equal probability. It’s often necessary but *not* sufficient to show that a specific message maps to a ciphertext with equal probability under a given key; additionally, it’s necessary to show that all ciphertexts can be produced by a particular message (perhaps by varying the key).

Shannon security can also be expressed as a conditional probability,<sup>1</sup> where all messages are equally-probable (i.e. independent of being) given a ciphertext:

$$\begin{aligned} &\forall m \in \text{MsgSp}, \forall C : \\ &\Pr[M = m | C] = \Pr[M = m] \end{aligned}$$

Are one-time pads Shannon-secure under these definitions? Yes, thanks to XOR.

## 0.2.1 The Beauty of XOR

XOR is the only primitive binary operator that outputs 1s and 0s with the same frequency, and that’s what makes it the perfect operation for achieving unpredictable ciphertexts.

Given a single bit, what’s the probability of the input bit?

Suppose you have some  $c = 0$  (where  $c \in \{0, 1\}^1$ ); what was the input bit  $m$ ? Well it could’ve been 1 and been XOR’d with 1 OR it could’ve been 0 and been XOR’d with 0... Knowing  $c$  gives us no new information about the input: our guess is still as good as random chance ( $\frac{1}{2} = 50\%$ ).

<sup>1</sup> See *Definition 2.3* in Katz & Lindell, pp. 29, parts of which are available on [Google Books](https://books.google.com/books?id=8187BVHqp6YC).

$x$	$y$	$x \oplus y$
1	1	0
1	0	1
0	1	1
0	0	0

**Table 1:** The truth table for XOR.

Now suppose you know that  $c = 1$ ; are your odds any better? In this case,  $m$  could've been 1 and been XOR'd with 0 OR it could've been 0 and XOR'd with 1... Again, we can't do better than random chance.

By the very definition of being Shannon-secure, if we (as the attacker) can't do better than random chance when given a ciphertext, the scheme is perfectly secure.

## 0.2.2 Proving Security

So we did a bit of a hand-wavy proof to show that XOR is Shannon-secure, but let's be a little more formal in proving that OTPs are perfect as well.

**Theorem 0.1.** *One-time pads are a perfect-security encryption scheme.*

*Proof.* We start by fixing an arbitrary  $n$ -bit ciphertext:  $C \in \{0, 1\}^n$ . We also choose a fixed  $n$ -bit message,  $m \in \text{MsgSp}$ . Then, what's the probability that a randomly-generated key  $k \in \text{KeySp}$  will encrypt that message to be that ciphertext? Namely, what is

$$\Pr[\mathcal{E}(K, m) = C]$$

for our **fixed**  $m$  and  $C$ ? In other words, how many keys can turn  $m$  into  $C$ ?

Well, by the definition of the OTP, we know that this can only be true for a single key:  $K = m \oplus C$ . Well, since every bit counts, and the probability of a single bit in the key being "right" is  $1/2$ :

$$\begin{aligned} \Pr[\mathcal{E}(K, m) = C] &= \Pr[K = m \oplus C] \\ &= \underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdots}_{n \text{ times}} \\ &= \frac{1}{2^n} \end{aligned}$$

Note that this is true  $\forall m \in \text{MsgSp}$ , which fulfills the requirement for perfect security! Every message is equally likely to result in a particular ciphertext. ■

The problem with OTPs is that keys can only be used once. If we're going to go through the trouble of securely distributing OTPs,<sup>2</sup> we could just exchange the messages themselves at that point in time...

Let's look at what happens when we use the same key across two messages. From the scheme itself, we know that  $C_i = K \oplus M_i$ . Well if we also have  $C_j = K \oplus M_j$ , then:

$$C_i \oplus C_j = (K \oplus M_i) \oplus (K \oplus M_j)$$

<sup>2</sup> One could envision a literal physical pad in which each page contained a unique bitstring; if two people shared a copy of these pads, they could communicate securely until the bits were exhausted (or someone else found the pad). Of course, if either of them lost track of where they were in the pad, everything would be gibberish from then-on...



$$\begin{aligned}
&= (K \oplus K) \oplus (M_i \oplus M_j) && \text{XOR is associative} \\
&= M_i \oplus M_j && a \oplus a = 0
\end{aligned}$$

Though this may seem like insignificant information, it actually can [reveal](#) quite a bit about the inputs, and eventually the entire key if it's reused enough times.

An important corollary of perfect security is what's known as the **impossibility result** (also referred to as the **optimality of the one-time pad** when used in that context):

**Theorem 0.2.** *If a scheme is [Shannon-secure](#), then the key space cannot be smaller than the message space. That is,*

$$|\text{KeySp}| \geq |\text{MsgSp}|$$

*Proof.* We are given an encryption scheme  $\mathcal{E}$  that is supposedly perfectly-secure. So we start by fixing a ciphertext with a specific key, ( $K_1 \in \text{KeySp}$  and plaintext message,  $m_1 \in \text{MsgSp}$ ):

$$C = \mathcal{E}(K_1, m_1)$$

We know for a fact, then, that at least one key exists that can craft  $C$ ; thus if we pick a key  $K \in \text{KeySp}$  *at random*, there's a non-zero probability that we'd get  $C$  again:

$$\Pr[\mathcal{E}(K, m_1) = C] > 0$$

Suppose then there is a message  $m_2 \in \text{MsgSp}$  which we can *never* get from decrypting  $C$ :

$$\Pr[\mathcal{D}(K, C) = m_2] = 0 \quad \forall K \in \text{KeySp}$$

By the correctness requirement of a valid encryption scheme, if a message can never be decrypted from a ciphertext, neither should that ciphertext result from an encryption of the message:

$$\Pr[\mathcal{E}(K, M) = C] = 0 \quad \forall K \in \text{KeySp}$$

However, that violates Shannon-secrecy, in which the probability of a ciphertext resulting from the encryption of *any* two messages is equal; that's not the case here:

$$\Pr[\mathcal{E}(K, M_1) = C] \neq \Pr[\mathcal{E}(K, M_2) = C]$$

Thus, our assumption is wrong:  $m_2$  cannot exist! Meaning there *must* be some  $K_2 \in \text{KeySp}$  that decrypts  $C$ :  $\mathcal{D}(K_2, C) = M_2$ . Thus, it must be the case that there are as many keys as there are messages. ■

Ideally, we'd like to encrypt long messages using short keys, yet this theorem shows that we cannot be perfectly-secure if we do so. Does that indicate the end of this chapter? Thankfully not. If we operate under the assumption that our adversaries are computationally-bounded, it's okay to relax the security requirement and make breaking our encryption schemes very, *very* unlikely. Though we won't have *perfect* secrecy, we can still do extremely well.

We will create cryptographic schemes that are computationally-secure under **Kerckhoff's principle**, which effectively states that *everything* about a scheme should be publicly-available except for the secret key(s).

# BLOCK CIPHERS

These are one of the fundamental building blocks of symmetric cryptography: a **block cipher** is a tool for encrypting short strings. Well-known examples include [AES](#) and [DES](#).

## DEFINITION 1.1: Block Cipher

Formally, a block cipher is a **function family** that maps from a  $k$ -bit key and an  $n$ -bit input string to an  $n$ -bit output string:

$$\mathcal{E} : \{0, 1\}^k \times \{0, 1\}^n \mapsto \{0, 1\}^n$$

Additionally,  $\forall K \in \{0, 1\}^k$ ,  $\mathcal{E}_K(\cdot)$  is a permutation on  $\{0, 1\}^n$ . This means its inverse is well-defined; we denote it either as  $\mathcal{E}_K^{-1}(\cdot)$  or the much more intuitive  $\mathcal{D}_K(\cdot)$ .

$$\begin{aligned} \forall M, C \in \{0, 1\}^n : \quad & \mathcal{E}_K(\mathcal{D}_K(C)) = C \\ & \mathcal{D}_K(\mathcal{E}_K(M)) = M \end{aligned}$$

In a similar vein, ciphertexts are unique, so  $\forall C \in \{0, 1\}^n$ , there exists a *single*  $M$  such that  $C = \mathcal{E}_K(M)$ .

## MATH REVIEW: Functions

A function is **one-to-one** if every input value maps to a unique output value. In other words, it's when no two inputs map to the same output.

A function is **onto** if all of the elements in the range have a corresponding input. That is,  $\forall y \exists x$  such that  $f(x) = y$ .

A function is **bijective** if it is both one-to-one and onto; it's a **permutation** if it maps a set onto itself. In our case, the set in question will typically be the set of all  $n$ -length bitstrings:  $\{0, 1\}^n$ .

## 1.1 Modes of Operation

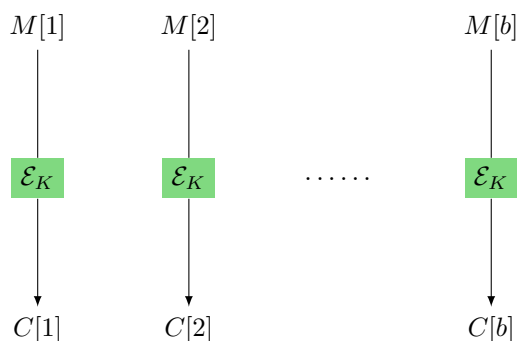
Block ciphers are limited to encrypting an  $n$ -bit string, but we want to be able to encrypt arbitrary-length strings. A **mode of operation** is a way to combine block ciphers to achieve this goal. For simplicity, we'll assume that our arbitrarily-long messages are actually a multiple of a block length; if they weren't, we could just pad them, but we'll omit that detail for brevity.

### 1.1.1 ECB—Electronic Code Book

The simplest mode of operation is ECB mode, visually described in [Figure 1.1](#). Given an  $n$ -bit block cipher  $\mathcal{E}$  and a message of length  $nb$ , we could just encrypt it block by block. The decryption is just as easy, applying the inverse block cipher on each piece individually:

$$C[i] = \mathcal{E}_K(M[i])$$

$$M[i] = \mathcal{D}_K(C[i])$$

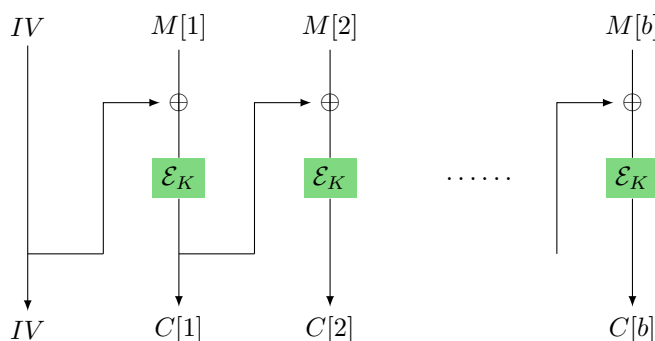


**Figure 1.1:** The ECB ciphering mode.

This mode of operation has a fatal flaw that greatly compromises its security: if two message blocks are identical, the ciphertexts will be as well. Furthermore, encrypting the same long message will result in the same long ciphertext. This mode of operation is never used, but it's useful to present here to highlight how we'll fix these flaws in later modes.

### 1.1.2 CBC—Cipher-Block Chaining

This mode of operation fixes both flaws in ECB mode and is usable in real symmetric encryption schemes. It introduces a random **initialization vector** or IV to keep each ciphertext random, and it chains the output of one block into the input of the next block.



**Figure 1.2:** The CBC ciphering mode.

Each message block is first chained via XOR with the previous ciphertext before being run through the encryption algorithm. Similarly, the ciphertext is run through the inverse then XOR'd with the previous ciphertext to decrypt. That is,

$$C[i] = \mathcal{E}_K(M[i] \oplus C[i-1])$$

$$M[i] = \mathcal{D}_K(C[i]) \oplus C[i-1]$$

(where the base case is  $C[0] = IV$ ).

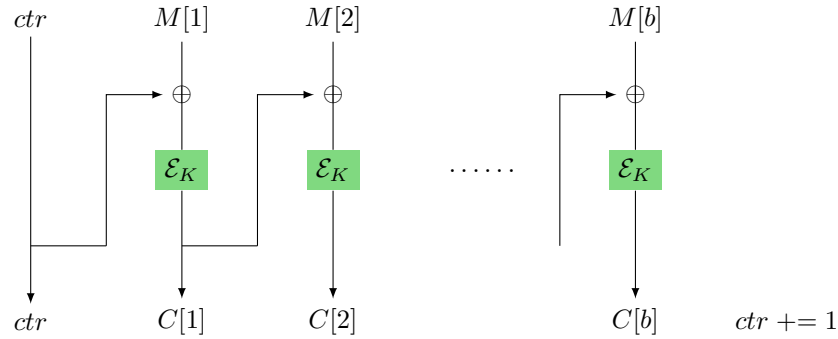
The IV can be sent out in the clear, unencrypted, because it doesn't contain any secret information in-and-of itself. If Eve intercepts it, she can't do anything useful with it; if Mallory modifies it, the decrypted plaintext will be gibberish and the recipient will know something is up.

**However**, if an initialization vector is **repeated**, there can be information leaked to keen attackers about the underlying plaintext.

### 1.1.3 CBC—Cipher-Block Chaining with Counter

In this mode, instead of using a randomly-generated IV, a counter is incremented for each new *message* until it wraps around (which typically doesn't occur, consider  $2^{128}$ ). This counter is XOR'd with the plaintext to encrypt and decrypt:

$$\begin{aligned} C[i] &= \mathcal{E}_K(M[i] \oplus ctr) \\ M[i] &= \mathcal{D}_K(C[i]) \oplus ctr \end{aligned}$$



**Figure 1.3:** The CBC ciphering mode.

The downside of these two algorithms is not a property of security but rather of performance. Because every block depends on the outcome of the previous block, both encryption and decryption must be done in series. This is in contrast with...

### 1.1.4 CTR—Randomized Counter Mode

Like ECB mode, this encryption mode encrypts each block independently, meaning it can be parallelized. Unlike all of the modes we've seen so far, though, it does not use a block cipher as its fundamental primitive.<sup>1</sup> Specifically, the encryption function does not need to be invertible. Whereas before we used  $\mathcal{E}_K$  as a mapping from a  $k$ -bit key and an  $n$ -bit string to an  $n$ -bit string (see [Definition 2.2](#)), we can now use a function that instead maps them to an  $m$ -bit string:

$$F : \{0, 1\}^k \times \{0, 1\}^l \mapsto \{0, 1\}^L$$

This is because both the encryption and decryption schemes use  $F_K$  directly. They rely on a randomly-generated value  $R$  as fuel, much like the IV in the CBC modes.<sup>2</sup> Notice that to decrypt  $C[i]$  in [Figure 1.4](#), one needs to first determine  $F_K(R + i)$ , then XOR that with the ciphertext to get

<sup>1</sup> In practice, though,  $F_K$  will generally be a block cipher. Even though this properly is noteworthy, it does not offer any additional security properties.

<sup>2</sup> In fact, I'm not sure why the lecture decides to use  $R$  instead of  $IV$  here to maintain consistency. They are mathematically the same: both  $R$  and  $IV$  are pulled from  $\{0, 1\}^n$ .

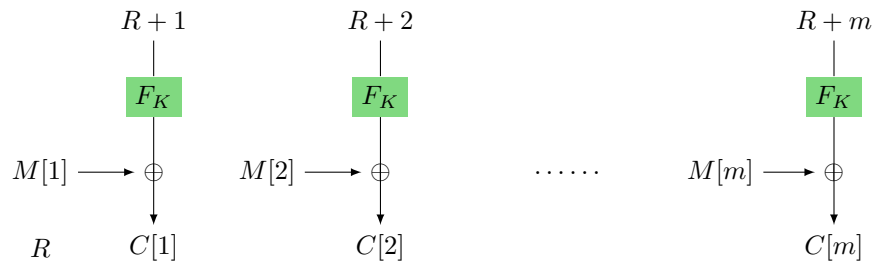


Figure 1.4: The CTR ciphering mode.

$M[i]$ . The plaintext is never run through the encryption algorithm at all; instead,  $F_K(R+i)$  is used as a **one-time pad** for  $M[i]$ . That is,

$$\begin{aligned} C[i] &= M[i] \oplus \mathcal{E}_K(R+i) \\ M[i] &= C[i] \oplus \mathcal{E}_K(R+i) \end{aligned}$$

Note that in all of these schemes, the only secret is  $K$  ( $F$  and  $\mathcal{E}$  are likely standardized and known).

### 1.1.5 CTRC—Stateful Counter Mode

Just like CBC, this mode has a variant that uses a counter rather than a randomly-generated value.

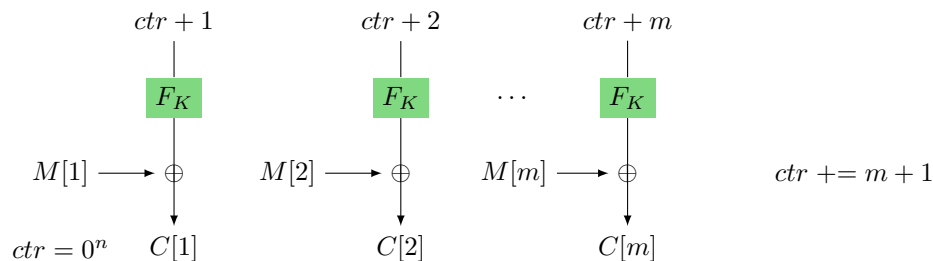


Figure 1.5: The CTRC ciphering mode.

## 1.2 Security Evaluation

Recall that we established that **Shannon-secure** schemes are impractical, and that we’re instead relying on adversaries being computationally bounded to achieve a reasonable level of security. To analyze our portfolio block ciphers, then, we need new definitions of this “computationally-bounded level of security.”

It’s easier to reverse the definition: a secure scheme is one that is not insecure. An insecure scheme allows a passive adversary that can see all ciphertexts do malicious things like learn the secret key or read any of the plaintexts. This isn’t rigorous enough, though: if the attacker can’t see any bits of the plaintext but can compute their sum, is that secure? What if they can tell when identical plaintexts are sent, despite not knowing their content?

There are plenty of possible information leaks to consider and it’s impossible to enumerate them all (especially when new attacks are still being discovered!). Dr. Boldyreva informally generalizes the aforementioned ideas:

*Informally, an encryption scheme is secure if no adversary with “reasonable” resources who sees several ciphertexts can compute any<sup>3</sup> partial information about the plaintexts, besides some a priori information.*

Though this informality is not useful enough to prove things about encryption schemes we encounter, it’s enough to give us intuition on the formal definition ahead.

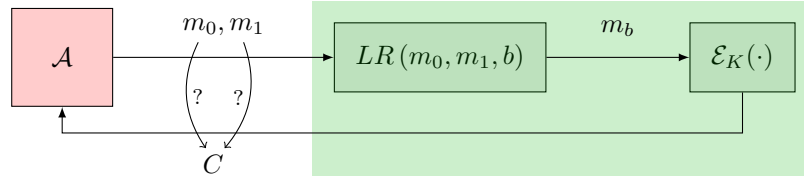
### 1.2.1 IND-CPA: Indistinguishability Under Chosen-Plaintext Attacks

It may be a mouthful, but the ability for a scheme to keep all information hidden when an attacker gets to feed their chosen inputs to it is key to a secure encryption scheme. **IND-CPA** is the formal definition of this attack.

We start with a fixed scheme and secret key,  $\mathcal{SE} = (\text{KeySp}, \mathcal{E}, \mathcal{D})$ ;  $K \xleftarrow{\$} \text{KeySp}$ .

Consider an adversary  $\mathcal{A}$  that has access to an oracle. When they provide the oracle with a pair of equal-length messages,  $m_0, m_1 \in \text{MsgSp}$ , it outputs a ciphertext.

The oracle, called the “left-right encryption” oracle, chooses a bit  $b \in \{0, 1\}^1$  to determine which of these messages to encrypt. It then passes  $m_b$  to the encryption function and outputs the ciphertext,  $C = \mathcal{E}_K(m_b)$ .



**Figure 1.6:** A visualization of the IND-CPA adversary scenario.

The adversary does not know the value of  $b$ , and thus does not know which of the messages was encrypted; it’s their goal to figure this out, given full access to the oracle. We say that an encryption scheme is secure if the adversary’s ability to determine which experiment the ciphertexts came from is no better than random chance.

#### DEFINITION 1.2: IND-CPA

A scheme  $\mathcal{SE}$  is considered secure under IND-CPA if an adversary’s **IND-CPA advantage**—the difference between their probability of guessing correctly and guessing incorrectly—is small ( $\approx 0$ ):

$$\text{Adv}^{\text{ind-cpa}}(\mathcal{A}) = \Pr[\mathcal{A} \text{ guessed 0 for experiment 0}] - \Pr[\mathcal{A} \text{ guessed 0 for experiment 1}]$$

Since we are dealing with a “computationally-bounded” adversary,  $\mathcal{A}$ , we need to be cognizant about the real-world meaning behind resource usage. At the very least, we should consider the running time of our scheme and  $\mathcal{A}$ ’s attack. After all, if the encryption function itself takes an entire year, it’s likely unreasonable to give the attacker more than a few hundred tries at the oracle before they’re time-bound.

We should likewise be cognizant of how many queries the attacker makes and how long they are.

<sup>3</sup> Any information *except* the length of the plaintexts; this knowledge is assumed to be public.

We might be willing to make certain compromises of security if, for example, the attacker needs a  $2^{512}$ -length message to gain an advantage.

With our new formal definition of security under our belt, let's take a crack at breaking the various [Modes of Operation](#) we defined. If we can provide an algorithm that demonstrates a reasonable advantage for an adversary that requires reasonable resources, we can show that a scheme is not secure under [IND-CPA](#).

### Analysis of ECB

This was clearly the simplest and weakest of schemes that we outlined. The lack of randomness makes gaining an advantage trivial: the message can be determined by having a message with a repeating and one with a non-repeating plaintext.

---

**ALGORITHM 1.1:** A simple algorithm for breaking the ECB block cipher mode.

```

 $C_1 \parallel C_2 = \mathcal{E}_K(LR(0^{2n}, 0^n \parallel 1^n, b))$ 
if  $C_1 = C_2$  then
  | return 0
end
return 1

```

---

The attack can be generalized to give the adversary perfect knowledge for any input plaintext, and it leads to an important corollary.

<p><b>Theorem 1.1.</b> <i>Any deterministic, stateless encryption scheme cannot be IND-CPA secure.</i></p>
--

*Proof.* Under deterministic encryption, identical plaintexts result in identical ciphertexts. We can always craft an adversary with an advantage. First, we associate ciphertexts with plaintexts, then by the third message we can always determine which ciphertext corresponds to which input.

---

**ALGORITHM 1.2:** A generic algorithm for breaking deterministic encryption schemes.

```

 $C_1 = \mathcal{E}_K(LR(0^n, 0^n, b))$ 
 $C_2 = \mathcal{E}_K(LR(1^n, 1^n, b))$ 
// Given knowledge of these two, we can now always differentiate between them. We can
   repeat this for any  $m \in \mathcal{MsgSp}$ .
 $C_3 = \mathcal{E}_K(LR(0^n, 1^n, b))$ 
if  $C_3 = C_1$  then
  | return 0
end
return 1

```

---

The proof holds for an arbitrary  $\mathcal{MsgSp}$ , we chose  $\{0,1\}^n$  in [algorithm 1.2](#) for convenience of representation. ■

### Analysis of CBCC

Turns out, counters are far harder to “get right” relative to random [initialization vectors](#): their predictable nature means we can craft messages that are effectively deterministic by replicating the counter state. Namely, if we pre-XOR our plaintext with the counter, the first ciphertext block functions the same way as in ECB.

The first message lets us identify the counter value. The second message lets us craft a “post-counter” message that will be equal to the third message.

---

**ALGORITHM 1.3:** A simple adversarial algorithm to break CBCC mode.

```
// First, determine the counter (can't count on  $ctr = 0$ ).
 $C_0 \parallel C_1 = \mathcal{E}_K(LR(0^n, 1^n, b))$ 

// Craft a message that'll be all-zeros post-counter.
 $M_1 = 0^n \oplus (ctr + 1)$ 
 $C_2 \parallel C_3 = \mathcal{E}_K(LR(M_1, 1^n, b))$ 

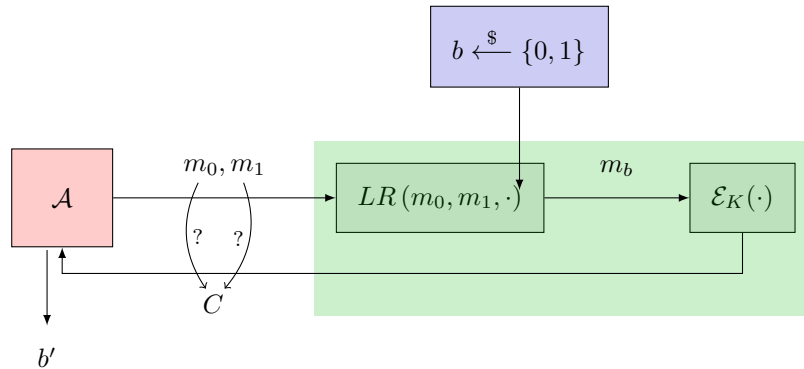
// Craft it again, then compare equality.
 $M_3 = 0^n \oplus (ctr + 2)$ 
 $C_4 \parallel C_5 = \mathcal{E}_K(LR(M_3, 1^n, b))$ 
if  $C_3 = C_5$  then
  | return 0
end
return 1
```

---

### 1.2.2 IND-CPA-cg: A Chosen Guess

It turns out that the other [modes of operation](#) are provably IND-CPA secure *if* the underlying block cipher is secure. Before we dive into those proofs, though, let's define an alternative interpretation of the [IND-CPA advantage](#); we will call this formulation **IND-CPA-cg**, for “chosen guess,” and its shown visually in [Figure 1.7](#). This formulation will be more convenient to use in some proofs.

In this version, the choice between left and right message is determined randomly at the start and encoded within  $b$ . There is now only one experiment: if the attackers guess matches ( $b' = b$ ), the experiment returns 1.



**Figure 1.7:** The “chosen guess” variant on IND-CPA security, where the attacker must guess a  $b'$ , and the experiment returns 1 if  $b' = b$ .



**DEFINITION 1.3: IND-CPA-cg**

A scheme  $\mathcal{SE}$  is still only considered secure under the “chosen guess” variant of IND-CPA if their **IND-CPA-cg advantage** is small; this advantage is now instead defined as:

$$\text{Adv}^{\text{ind-cpa-cg}}(\mathcal{A}) = 2 \cdot \Pr[\text{experiment returns 1}] - 1$$

The two variants on attacker advantage in [Definition 2.3](#) and the new [Definition 2.4](#) can be proven equal.

**Claim 1.1.**  $\text{Adv}^{\text{ind-cpa}}(\mathcal{A}) = \text{Adv}^{\text{ind-cpa-cg}}(\mathcal{A})$  for some encryption scheme  $\mathcal{SE}$ .

*Proof.* The probability of the **cg** experiment being 1 (that is, the attacker guessing  $b' = b$  correctly) can be expressed as conditional probabilities. Remember that  $b \xleftarrow{\$} \{0, 1\}$  with uniformly-random probability.

$$\begin{aligned} \Pr[\text{experiment-cg returns 1}] &= \Pr[b = b'] \\ &= \Pr[b = b' | b = 0] \Pr[b = 0] + \Pr[b = b' | b = 1] \Pr[b = 1] \\ &= \Pr[b' = 0 | b = 0] \cdot \frac{1}{2} + \Pr[b' = 1 | b = 1] \cdot \frac{1}{2} \\ &= \frac{1}{2} \cdot \Pr[b' = 0 | b = 0] + \frac{1}{2} (1 - \Pr[b' = 0 | b = 1]) \\ &= \frac{1}{2} + \frac{1}{2} (\Pr[b' = 0 | b = 0] - \Pr[b' = 0 | b = 1]) \end{aligned}$$

Notice the expression in parentheses: the difference between the probability of the attacker guessing 0 correctly (that is, when it really is 0) and incorrectly. This is exactly [Definition 2.3](#): advantage under the normal IND-CPA definition! Thus:

$$\begin{aligned} \Pr[\text{exp-cg returns 1}] &= \frac{1}{2} + \frac{1}{2} \underbrace{\Pr[b' = 0 | b = 0] - \Pr[b' = 0 | b = 1]}_{\text{IND-CPA advantage}} \\ &= \frac{1}{2} + \frac{1}{2} \text{Adv}^{\text{ind-cpa}}(\mathcal{A}) \\ 2 \cdot \Pr[\text{exp-cg returns 1}] - 1 &= \text{Adv}^{\text{ind-cpa}}(\mathcal{A}) \\ \text{Adv}^{\text{ind-cpa-cg}}(\mathcal{A}) &= \text{Adv}^{\text{ind-cpa}}(\mathcal{A}) \end{aligned} \tag{1.1}$$

■

### 1.2.3 What Makes Block Ciphers Secure?

We can now look into the inner guts of each [mode of operation](#) and classify some block ciphers as being “secure” under IND-CPA. Refer to [Definition 2.2](#) to review the mathematical properties of a [block cipher](#). Briefly, it is a function family with a well-defined inverse that maps every message to a unique ciphertext for a specific key.

First off, it’s important to recall that we expect attackers to be computationally-bounded to a reasonable degree. This is because block ciphers—and all symmetric encryption schemes, for that matter—are susceptible to an **exhaustive key-search** attack, in which an attacker enumerates every possible  $K \in \text{KeySp}$  until they find the one that encrypts some known message to a known ciphertext. If we say  $k = |\text{KeySp}|$ , this obviously takes  $\mathcal{O}(k)$  time and requires on average  $2^{k-1}$  checks, which is why  $k$  must be large enough for this to be infeasible.

**FUN FACT: Historical Key Sizes**

Modern block ciphers like [AES](#) use *at least* 128-bit keys (though 192 and 256-bit options are available) which is considered secure from exhaustive search.

The now-outdated block cipher [DES](#) (invented in the 1970s) had a 56-bit key space, and it had a particular property that could speed up exhaustive search by a factor of two. This means exhaustive key-search on DES takes  $\approx 2^{54}$  operations which took about 23 years on a 25MHz processor (fast at the time of DES' inception). By 1999, the key could be found in only 22 hours.

The improved triple-DES or 3DES block cipher used 112-bit keys, but it too was abandoned in favor of AES for performance reasons: doing three DES computations proved to be too slow for efficient practical use.

Obviously a block cipher is not necessarily secure just because exhaustive key-search is not feasible. We now aim to define some measure of security for a block cipher. Why can't we just use [IND-CPA](#)? Well a block cipher is deterministic *by definition*, and we saw in [Theorem 1.1](#), a deterministic scheme cannot be IND-CPA secure. Thus our definition is too strong! We need something weaker for block ciphers that is still lets us avoid all possible information leaks: nothing about the key, nothing about the plaintexts (or some property of the plaintexts), etc. should be revealed.

We will say that a block cipher is secure if its output ciphertexts “look” random; more precisely, it'd be secure if an attacker can't differentiate its output from a random function. Well... that requires a foray into random functions.

**1.2.4 Random Functions**

Let's say that  $\mathcal{F}(l, L)$  defines the set of ALL functions that map from  $l$ -bit strings to  $L$ -bit strings:

$$\forall f \in \mathcal{F}(l, L) \quad f : \{0, 1\}^l \mapsto \{0, 1\}^L$$

A random function  $g$  is then just a random function from that set:  $g(\cdot) \xleftarrow{\$} \mathcal{F}(l, L)$ . Now because picking a function at random is the same thing as picking a bitstring at random,<sup>4</sup> we can define  $g$  in pseudocode as a deterministic way of picking bitstrings (see [algorithm 1.4](#)).

---

**ALGORITHM 1.4:**  $g(x)$ , a random function.

```

Define a global array  $T$ 
if  $T[x]$  is not defined then
    |  $T[x] \xleftarrow{\$} \{0, 1\}^L$ 
end
return  $T[x]$ 

```

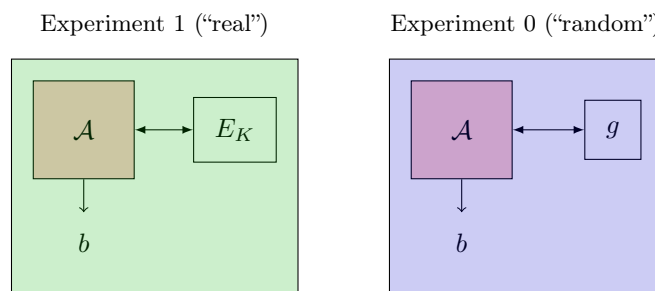
---

A function family is a [pseudorandom function](#) family (a PRF) if the input-output behavior of a random instance of the family is computationally indistinguishable from a truly-random function. This input-output behavior is defined by [algorithm 1.4](#) and is hidden from the attacker.

<sup>4</sup> Any function we pick will map values to  $L$ -bit strings. Concatenating all of these output bitstrings together will result in some  $nL$ -bit string, with a  $L2^L$  bitstring being longest if the function maps to *every* bitstring. Each chunk of this concatenated string is random, so we can just pick some random  $L2^L$ -length bitstring right off the bat to pick  $g$ .

## PRF Security

The security of a block cipher depends on whether or not an attacker can differentiate between it and a random function. Like with **IND-CPA**, we have two experiments. In the first experiment, the attacker gets the output of the block cipher  $E$  with a fixed  $K \in \text{KeySp}$ ; in the second, it's a random function  $g$  chosen from the PRF matching the domain and range of  $E$ .



The attacker outputs their guess,  $b$ , which should be 1 if they think they're being fed outputs from the real block cipher and 0 if they think it's random. Then, their "advantage" is how much more often the attacker can guess correctly.

### DEFINITION 1.4: Block Cipher Security

A block cipher is considered **PRF secure** if an adversary's **PRF advantage** is small (near-zero), where the advantage is defined as the difference in probabilities of the attacker choosing

$$\text{Adv}^{\text{prf}}(\mathcal{A}) = \Pr[\mathcal{A} \text{ returns 1 for experiment 1}] - \Pr[\mathcal{A} \text{ returns 1 for experiment 0}]$$

For **AES**, the PRF advantage is very small and its *conjectured* (not proven) to be PRF secure. Specifically, for running time  $t$  and  $q$  queries,

$$\text{Adv}_{\text{AES}}^{\text{prf}}(\mathcal{A}) \leq \underbrace{\frac{ct}{T_{\text{AES}}} \cdot 2^{-128}}_{\text{exhaustive key-search}} + \underbrace{q^2 \cdot 2^{-128}}_{\text{birthday paradox}} \quad (1.2)$$

We will use this as an upper bound when calling a function  $F$  PRF secure.

The second term comes from an interesting attack that can be applied to *all* block ciphers known as the birthday paradox. Recall that block ciphers are permutations, so for distinct messages, you always get distinct ciphertexts. The attack is simple: if you feed the PRF security oracle  $q$  distinct messages and get  $q$  distinct ciphertexts, you output  $b = 1$ ; otherwise, you output  $b = 0$ . The only way you get  $< q$  distinct ciphertexts is from a  $g$  that isn't one-to-one. The probability of this happening is the probability of [algorithm 1.4](#) picking the same bitstring for two  $xs$ , so  $2^{-L}$ .

**FUN FACT: The Birthday Paradox**

Suppose you're at a house party with 50 other people. What're the chances that two people at that party share the same birthday? Turns out, it's really, *really* high: 97%, in fact!

The **birthday paradox** is the counterintuitive idea despite the fact that YOU are unlikely to share a birthday with someone, the chance of ANY two people sharing a birthday is actually extremely high.

In the context of cryptography, this means that as the number of outputs generated by a random function  $g$  increases, the probability of SOME two inputs resolving to the same output increases much faster.

**Proving Security: CTRC** Recall the **CTRC—Stateful Counter Mode** mode of operation. Armed with the new definition of **block cipher** security, we can prove that this mode is secure. We start by assuming that the underlying cryptographic primitives are secure (in this case, this is the block cipher). Then, we can leverage the **contrapositive** to prove it. Starting with the implication:

**If** a scheme  $\mathcal{T}$  is  $y$ -secure,  
**then** a scheme  $\mathcal{S}$  is  $x$ -secure.

(for some fill-in-the-blank  $x, y$ s like “IND-CPA” or “PRF”), we instead aim to prove the contrapositive:

**If** a scheme  $\mathcal{S}$  is NOT  $x$ -secure,  
**then** a scheme  $\mathcal{T}$  is NOT  $y$ -secure.

To bring this into context, we will show that our mode of operation  $\mathcal{S}$  being insecure implies that the block cipher  $\mathcal{T}$  is *not* PRF-secure. More specifically, using our definitions of security, we're trying to show that: there existing an  $x$ -adversary  $\mathcal{A}$  that can break  $\mathcal{S}$  implies that there exists a  $y$ -adversary  $\mathcal{B}$  that can break  $\mathcal{T}$ :

- We assume  $\mathcal{A}$  exists, then construct  $\mathcal{B}$  using  $\mathcal{A}$ .
- Then, we show that  $\mathcal{B}$ 's  $y$ -advantage is not “too small” if  $\mathcal{A}$ 's  $x$ -advantage is not “too small” ( $\approx 0$ ).

**LOGIC REVIEW: Contrapositive**

The **contrapositive** of an implication is its inverted negation. Namely, for two given statements  $p$  and  $q$ :

$$\begin{aligned} &\text{if } p \implies q \\ &\text{then } \neg q \implies \neg p \end{aligned}$$

With that in mind, let's prove CTRC's security. To be verbose, the statements we're aiming to prove are:

$$\underbrace{\text{the underlying blockcipher is secure}}_P \implies \underbrace{\text{CTRC is a secure mode of operation}}_Q$$

However, since we're approaching this via the contrapositive, we'll instead prove

$$\begin{array}{c} \underbrace{\text{CTR} \text{ is not a secure mode of operation}}_{\neg Q} \\ \implies \text{only when } \underbrace{\text{the underlying blockcipher is not secure}}_{\neg P} \end{array}$$

**Theorem 1.2.** *CTR is a secure mode of operation if its underlying block cipher is secure.*

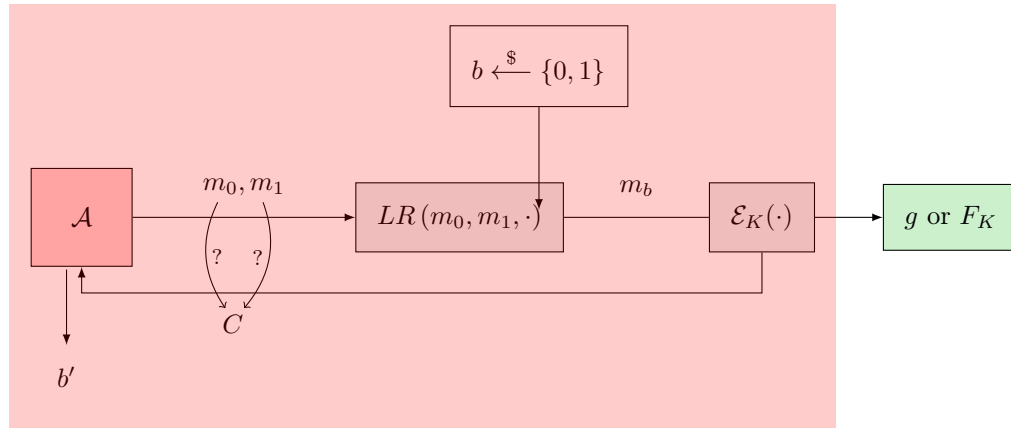
More formally, for any efficient adversary  $\mathcal{A}$ ,  $\exists \mathcal{B}$  with similar efficiency such that the *IND-CPA advantage* of  $\mathcal{A}$  under CTR mode is less than double the *PRF advantage* of  $\mathcal{B}$  under a secure block cipher  $F$ :

$$\text{Adv}_{\text{CTR}}^{\text{ind-cpa}}(\mathcal{A}) \leq 2 \cdot \text{Adv}_F^{\text{prf}}(\mathcal{B})$$

where we know an example of a secure block cipher  $F = \text{AES}$  that any  $\mathcal{B}$ 's advantage will be very small (see (1.2)).

*Proof.* Let  $\mathcal{A}$  be an *IND-CPA-cg* adversary attacking CTR. Then, we can present the PRF adversary  $\mathcal{B}$ .

- We construct  $\mathcal{B}$  so that it can act as the very left-right oracle that  $\mathcal{A}$  uses to query and attack the CTRC scheme.



adversary  $\mathcal{B} \longrightarrow 1$  iff  $b' = b$  else 0

- Namely,  $\mathcal{B}$  lets  $\mathcal{A}$  make oracle queries to CTRC until it guesses  $b$  correctly. This is valid because  $\mathcal{B}$  still delegates to a PRF oracle which is choosing between a random function  $g$  and the block cipher  $F_K$  (where  $K$  is still secret) for the actual block cipher; everything else is done exactly as described for CTRC in Figure 1.5.
- This construction lets us leverage the fact that  $\mathcal{A}$  knows how to break CTRC-encrypted messages, but we don't need to know how. For the pseudocode describing this process, refer to algorithm 1.5.

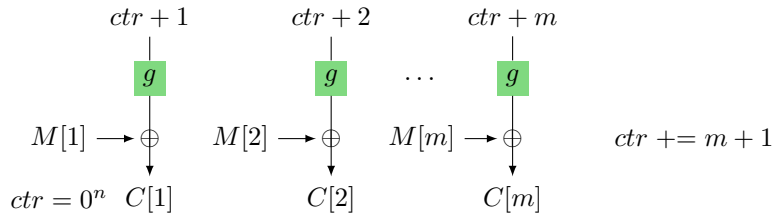
Now let's analyze  $\mathcal{B}$ , expressing its PRF advantage over  $F$  in terms of  $\mathcal{A}$ 's IND-CPA advantage over CTRC.<sup>5</sup> The ability for  $\mathcal{B}$  to differentiate between  $F$  and some random function  $g \in \text{Func}(\ell, L)$

<sup>5</sup> The syntax  $\mathcal{X} \rightarrow n$  means the adversary  $\mathcal{X}$  outputs the value  $n$ , and the syntax  $\text{Exp}_m^n$  refers to the experiment  $n$  under some parameter or scheme  $m$ , for shorthand.

depends *entirely* on  $\mathcal{A}$ 's ability to differentiate between CTRC with an actual block cipher  $F$  and a truly-random function  $g$ . Thus,

$$\begin{aligned} \text{Adv}_F^{\text{prf}}(\mathcal{B}) &= \Pr[\mathcal{B} \rightarrow 1 \text{ in } \text{Exp}_F^{\text{prf-0}}] - \Pr[\mathcal{B} \rightarrow 1 \text{ in } \text{Exp}_F^{\text{prf-1}}] && \text{definition} \\ &= \Pr[\text{Exp}_{\text{CTRC}[F]}^{\text{ind-cpa-cg}} \rightarrow 1] - \Pr[\text{Exp}_{\text{CTRC}[g]}^{\text{ind-cpa-cg}} \rightarrow 1] && \mathcal{B} \text{ depends only on } \mathcal{A} \\ &= \frac{1}{2} \cdot \text{Adv}_{\text{CTRC}[F]}^{\text{ind-cpa}}(\mathcal{A}) + \frac{1}{2} - \frac{1}{2} \cdot \text{Adv}_{\text{CTRC}[g]}^{\text{ind-cpa}}(\mathcal{A}) - \frac{1}{2} && \text{IND-CPA is equal to IND-CPA-cg via (1.1)} \end{aligned}$$

Next, we'll show that  $\text{Adv}_{\text{CTRC}[g]}^{\text{ind-cpa}}(\mathcal{A}) = 0$ . That is, we will show that  $\mathcal{A}$  has absolutely no advantage in breaking the scheme when using  $g$ —a truly-random function—as the block cipher. Consider the visualization of the CTRC scheme again:



Notice that the inputs to  $g$  are all distinct points, and by definition of a truly-random function its outputs are truly-random bitstrings. These are then XOR'd with messages... sound familiar? The outputs of  $g$  are distinct **one-time pads** and thus each  $C[i]$  is **Shannon-secure**, meaning an advantage is simply impossible by definition.

The theorem claim can then be trivially massaged out:

$$\begin{aligned} \text{Adv}_F^{\text{prf}}(\mathcal{B}) &= \frac{1}{2} \cdot \text{Adv}_{\text{CTRC}[F]}^{\text{ind-cpa}}(\mathcal{A}) + \frac{1}{2} - \frac{1}{2} \cdot \underbrace{\text{Adv}_{\text{CTRC}[g]}^{\text{ind-cpa}}(\mathcal{A})}_{=0} - \frac{1}{2} \\ &= \frac{1}{2} \cdot \text{Adv}_{\text{CTRC}}^{\text{ind-cpa}}(\mathcal{A}) \\ 2 \cdot \text{Adv}_F^{\text{prf}}(\mathcal{B}) &= \text{Adv}_{\text{CTRC}}^{\text{ind-cpa}}(\mathcal{A}) \end{aligned}$$

■

**Proving Security: CTR** Recall that the difference between CTRC (which we just proved was secure) and standard CTR is the use of a random IV rather than a counter (see [Figure 1.4](#)). It's also provably **PRF secure**, but we'll state its security level without proof.<sup>6</sup>

**Theorem 1.3.** *CTR is a secure mode of operation if its underlying block cipher is secure. More formally, for any efficient adversary  $\mathcal{A}$ ,  $\exists \mathcal{B}$  with similar efficiency such that:*

$$\text{Adv}_{\text{CTR}}^{\text{ind-cpa}}(\mathcal{A}) \leq 2 \cdot \text{Adv}_F^{\text{prf}}(\mathcal{B}) + \frac{\mu_A^2}{\ell 2^\ell}$$

where  $\mu$  is the total number of bits  $\mathcal{A}$  sends to the oracle.

<sup>6</sup> Feel free to refer to the [lecture video](#) to see the proof. In essence, the fact the value is chosen randomly means it's possible that for enough  $R$ s and  $M$ s there will be overlap for some  $R_i + m$  and  $R_j + n$ . This will result in identical "one-time pads," though thankfully it occurs with a very small probability (it's related to the [birthday paradox](#)).

---

**ALGORITHM 1.5:** Constructing an adversary  $\mathcal{B}$  that uses another adversary  $\mathcal{A}$  to break a higher-level symmetric encryption scheme.

**Input:** An adversary  $\mathcal{A}$  that executes oracle queries.

**Result:** 1 if  $\mathcal{A}$  succeeds in breaking the emulated scheme, 0 otherwise.

Let  $g \xleftarrow{\$} F(\ell, L)$  where  $F$  is a **PRF**, which  $\mathcal{B}$  will use as a **block cipher**.

Let  $\mathcal{E}_f(\cdot)$  be an encryption function that works like  $\mathcal{A}$  expects (for example, a CTRC scheme).

Choose a random bit:  $b \xleftarrow{\$} \{0, 1\}^1$

**repeat**

    Get a query from  $\mathcal{A}$ , some  $(M_1, M_2)$

$C \xleftarrow{\$} \mathcal{E}_f(M_b)$

    return  $C$  to  $\mathcal{A}$

**until**  $\mathcal{A}$  outputs its guess,  $b'$

**return** 1 iff  $b = b'$ , 0 otherwise

---

It's still secure because  $\ell \geq 128$  for secure block ciphers, making the extra term near-zero. Proving bounds on security is very useful: we can see here that CTRC mode is better than CTR mode because there is no additional constant.

There is a similar theorem for CBC mode (see Figure 1.2), the last mode of operation whose security we haven't formalized.

**Theorem 1.4.** *CBC is a secure mode of operation if its underlying block cipher is secure. More formally, for any efficient adversary  $\mathcal{A}$ ,  $\exists \mathcal{B}$  with similar efficiency such that:*

$$\text{Adv}_{CBC}^{\text{ind-cpa}}(\mathcal{A}) \leq 2 \cdot \text{Adv}_F^{\text{prf}}(\mathcal{B}) + \frac{\mu_A^2}{n^2 2^n}$$

where  $\mu$  is the total number of bits  $\mathcal{A}$  sends to the oracle.

We can see that  $n^2 > \ell$  when comparing CBC to CTR, meaning the term will be smaller for the same  $\mu$ . Thus, CTRC is more secure than CBC is more secure than CTR. The constant again comes from the [birthday paradox](#).

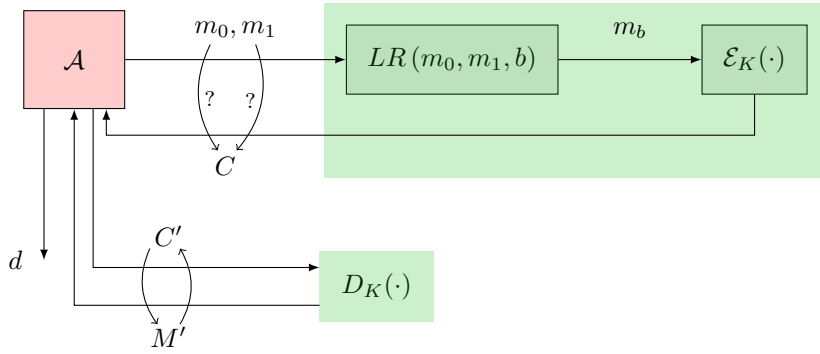
### 1.2.5 IND-CCA: Indistinguishability Under Chosen-Ciphertext Attacks

Is the intuition behind a scheme being both **IND-CPA** and **PRF secure** sufficient? Does IND-CPA take into account all of the possible attack vectors? Well, it limits attackers to choosing *plaintexts* and using only their ciphertext results to make learn information about the scheme and see ciphertexts. What if the attacker could instead attack a scheme by choosing *ciphertexts* and learn something about the scheme from the resulting plaintexts?

This isn't a far-fetched possibility,<sup>7</sup> and it has historic precedent in being a viable attack vector. Since **IND-CPA** does not cover this vector, we need a stronger definition of security: the attacker needs more power. With **IND-CCA**, the adversary  $\mathcal{A}$  has access to *two* oracles: the left-right encryption oracle, as before, and a decryption oracle.

---

<sup>7</sup> Imagine reverse-engineering an encrypted messaging service like iMessage to fully understanding its encryption scheme, and then control the data that gets sent to Apple's servers to "skip" the encryption step and control the ciphertext directly. If you control both endpoints, you can see what the ciphertext decrypts to!



**Figure 1.8:** A visualization of the IND-CCA security definition. The adversary  $\mathcal{A}$  submits two messages,  $(m_0, m_1)$ , to an encryption oracle that (consistently) chooses one of them based on a bit  $b$  and returns  $m_b$ 's ciphertext. The adversary can also submit any  $C'$  that hasn't been submitted to  $LR$  to a decryption oracle and see the resulting plaintext.

The only restriction on the attacker is that they cannot query the decryption oracle on ciphertexts returned from the encryption oracle (obviously, that would make determining  $b$  trivial) (in Figure 1.8, this means  $C \neq C'$ ). As before, a scheme is considered IND-CCA secure if an adversary's advantage is small.

### DEFINITION 1.5: IND-CCA

A scheme  $\mathcal{SE}$  is considered secure under IND-CCA if an adversary's **IND-CCA advantage**—the difference between their probability of guessing correctly and guessing incorrectly—is small ( $\approx 0$ ):

$$\text{Adv}^{\text{ind-cca}}(\mathcal{A}) = \Pr[\mathcal{A} \text{ guessed } 0 \text{ for experiment } 0] - \Pr[\mathcal{A} \text{ guessed } 0 \text{ for experiment } 1]$$

Note that since IND-CCA is stronger than IND-CPA, the former implies the latter. This is trivially-provable by reduction, so we won't show it here.

Unfortunately, none of our IND-CPA schemes are also secure under IND-CCA.

### Analysis of CBC

Recall from Figure 1.2 the way that message construction works under CBC with random **initialization vectors**.

Suppose we start by encrypting two distinct, two-block messages. They don't have to be the ones chosen here, but it makes the example easier. We pass these to the left-right oracle:

$$IV \parallel c_1 \parallel c_2 \xleftarrow{\$} \mathcal{E}_K(LR(0^{2n}, 1^{2n}))$$

From these ciphertexts alone, we've already shown that the adversary can't determine which of the input messages was encrypted. However, suppose we send just the first chunk to the decryption oracle?

$$m = \mathcal{D}_K(IV \parallel c_1)$$



This is legal since it's not an *exact* match for any encryption oracle outputs. Well since our two blocks were identical, and  $c_2$  has no bearing in the decryption of  $IV \parallel c_1$  (again, refer to the visualization in [Figure 1.2](#)), the plaintext  $m$  will be all-zeros in the left case and all-ones in the right case!

It should be fairly clear that this is an efficient attack, and that the adversary's advantage is optimal (exactly 1). For posterity,

$$\begin{aligned}\text{Adv}_{\text{CBC}}^{\text{ind-cca}}(\mathcal{A}) &= \Pr[\mathcal{A} \rightarrow 0 \text{ for } \text{Exp}^{\text{ind-cca-0}}] - \Pr[\mathcal{A} \rightarrow 0 \text{ for } \text{Exp}^{\text{ind-cca-1}}] \\ &= 1 - 0 = \boxed{1}\end{aligned}$$

The attack time  $t$  is the time to compare  $n$  bits, it requires  $q_e = q_d = 1$  query to each oracle, and message lengths of  $\mu_e = 4n$  and  $\mu_d = 2n$ . Thus, CBC is not IND-CCA secure. ■

Almost identical proofs can be used to break both CTR and CTRC, our final bastions of hope in the [Modes of Operation](#) we've covered.

### Analysis of CBC: Anotha' One

(or, *Kicking 'em While They're Down*)

We can break CBC (and the others) in a different way. This is included here to jog the imagination and offer an alternative way of thinking about showing insecurity under IND-CCA.

In this attack, one-block messages will be sufficient:

$$IV \parallel c_1 \xleftarrow{\$} \mathcal{E}_K(LR(0^n, 1^n))$$

This time, there's nothing to chop off. However, what if we try decrypting the ciphertext with a flipped IV?

$$m = \mathcal{D}_K(\overline{IV} \parallel c_1)$$

Well, according to [Figure 1.2](#), the output from the blockcipher will be XOR'd with the flipped IV, and thus result in a flipped message, so  $m = \overline{0^n} = 1^n$  in the left case, and  $m = 0^n$  in the right case!

Again, this is trivially computationally-reasonable (in fact, it's even *more* reasonable than before) and breaks IND-CCA security. ■

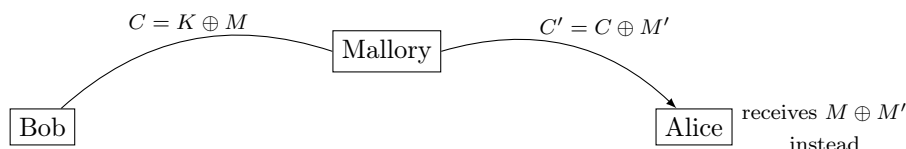
## 1.3 What Now?

We started off by enumerating a number of ways to create ciphertexts from plaintexts using [block ciphers](#). It was critical to follow that with several definitions of what "security" means, and we showed that some of the modes of operation (namely ECB and CBC) were not secure under [Definition 1.2](#), IND-CPA security. Then, we dug deeper to study the underlying block ciphers and what it meant for *those* to be [PRF secure](#): it must be hard to differentiate them from a [random function](#) (see [Definition 1.3](#)). Finally, we gave the attacker more power under the last, strictest metric of security: IND-CCA, and showed that our remaining modes of operation (that is, CBC, CTR, and CTRC) broke under this adversarial scheme.

We definitely want a way to achieve IND-CCA security. Does hope remain? Thankfully, it does, but it will first require a foray into the other realms of cryptography: [integrity](#) and [authenticity](#).

# MESSAGE AUTHENTICATION CODES

Data privacy and [confidentiality](#) is not the only goal of cryptography, and a good encryption method does not make any guarantees about anything beyond confidentiality. In the [one-time pad](#) (which is *perfectly* secure), an active attacker Mallory can modify the message in-flight to ensure that Alice receives something other than what Bob sent:



If Mallory knows that the first 8 bits of Bob's message corresponds to the number of dollars that Alice needs to send Bob (and she does, according to [Kerckhoff's principle](#)), such a manipulation will have catastrophic consequences for Alice's bank account. Clearly, we need a way for Alice to know that a message came from Bob himself.

Let's discuss ways to ensure that the recipient of a message can validate that the message came from the intended sender ([authenticity](#)) *and* was not modified on the way ([integrity](#)).

## 2.1 Notation & Syntax

A [message authentication code](#) (or MAC) is a fundamental primitive in achieving data authenticity under the symmetric cryptography framework. Much like in an encryption scheme, a well-defined MAC scheme covers the following:

- a **message space**, denoted as the  $\mathcal{MsgSp}$  or  $\mathcal{M}$  for short, describes the set of things which can be authenticated.
- a **key generation algorithm**,  $\mathcal{K}$ , or the key space spanned by that algorithm,  $\mathcal{KeySp}$ , describes the set of possible keys for the scheme and how they are created.
- the MAC algorithm itself,  $\mathcal{MAC}$  (also called a **tagging** or **signing algorithm**) defines the way some  $m \in \mathcal{MsgSp}$  is authenticated and returns a tag.
- the MAC's corresponding **verification algorithm**,  $\mathcal{VF}$ , describes how a message should be validated, given a (supposedly) authenticated message and its tag, outputting a Boolean value indicating their validity.

Succinctly, we say that  $\Pi = (\mathcal{K}, \mathcal{MAC}, \mathcal{VF})$ , and by definition

$$\forall k \in \mathcal{KeySp}, \forall m \in \mathcal{MsgSp} : \quad \mathcal{VF}(k, m, \mathcal{MAC}(k, m)) = 1$$

If a MAC algorithm is deterministic, then  $\mathcal{VF}$  does not need to be explicitly defined, since running

the MAC on the message again and comparing the resulting tags is sufficient.

An important thing to remember in this chapter is that *we don't care about confidentiality*: the messages and their tags are sent in the clear. Our only concern is now **forging**—can Mallory pretend that a message came from Bob?

## 2.2 Security Evaluation

As before, with the **Security Evaluation** of a **block cipher** or its **mode of operation**, we need a way to model practical, strong adversaries and their attacks on MACs.

To start, we can imagine that an adversary can see some number of  $(\text{message}, \text{tag})$  pairs. To mimic **IND-CCA**, perhaps s/he can also force the tagging of messages and check the verification of specific pairs. Obviously, they shouldn't be able to compute the secret key, but more importantly, they should *never* be able to compose a message and tag pairing that is considered valid.

### ATTACK VECTOR: Pay to (Re)Play

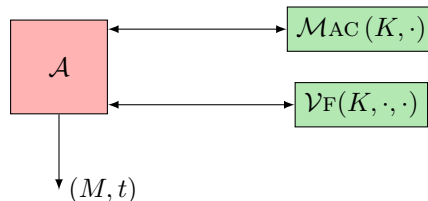
A **replay attack** is one where an adversary uses valid messages from the past that they captured to duplicate some action.

For example, imagine Bob sends an encrypted, authenticated message “You owe my friend Mallory \$5.” to Alice that everyone can see. Alice knows this message came from Bob, so she pays her dues. Then, Mallory decides to just... send Alice that message again! It's again deemed valid, and Alice again pays her dues.

Protection against replay attacks requires some more-sophisticated construction of a secure scheme, so we'll ignore them for now as we discuss MAC schemes.

### 2.2.1 UF-CMA: Unforgeability Under Chosen-Message Attacks

Let's formalize these intuitions: the adversary  $\mathcal{A}$  is given access to two oracles that run the tagging and verification algorithms respectively, and s/he must output a message-tag pair  $(M, t)$  for which  $t$  is a valid tag for  $M$  (that is,  $\mathcal{V}_F(K, M, t) = 1$ ) and  $M$  was never an input to the tagging oracle.<sup>1</sup>



#### DEFINITION 2.1: UF-CMA

A message authentication code scheme  $\Pi$  is considered to be **UF-CMA** secure if the **UF-CMA advantage** of any adversary  $\mathcal{A}$  is near-zero, where the advantage is defined

<sup>1</sup> This lone restriction on the adversary is exactly like the one for **IND-CCA**, where it's trivial to get a perfect advantage if you're allowed to decrypt messages you've encrypted.

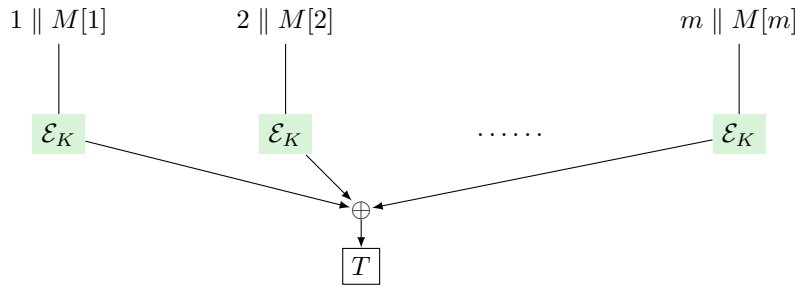
by the probability of the oracle mistakenly verifying a message:

$$\text{Adv}^{\text{uf-cma}}(\mathcal{A}) = \Pr[\mathcal{V}_F(k, m, t) = 1 \text{ and } m \text{ was not queried to the oracle}]$$

The latter part of the probability lets us ignore **replay attacks** and trivial breaks of the scheme.

### A Toy Example

Suppose we take a simple MAC scheme that prepends each message block with a counter, runs this concatenation through a block cipher, and XORs all of the ciphertexts (see [Figure 2.1](#)).



**Figure 2.1:** A simple MAC algorithm.

This can be broken easily if we realize that XORs can cancel each other out. Consider tags for three pairs of messages and what they expand to

$$\begin{aligned} T_1 &= \mathcal{MAC}(X_1 \parallel Y_1) &\longrightarrow &\mathcal{E}_K(1 \parallel X_1) \oplus \mathcal{E}_K(2 \parallel Y_1) \\ T_2 &= \mathcal{MAC}(X_1 \parallel Y_2) &\longrightarrow &\mathcal{E}_K(1 \parallel X_1) \oplus \mathcal{E}_K(2 \parallel Y_2) \\ T_3 &= \mathcal{MAC}(X_2 \parallel Y_1) &\longrightarrow &\mathcal{E}_K(1 \parallel X_2) \oplus \mathcal{E}_K(2 \parallel Y_1) \end{aligned}$$

If we combine these three tags, we can actually derive the tag for a new pair of messages!

$$\begin{aligned} T_1 \oplus T_2 \oplus T_3 &= \boxed{\mathcal{E}_K(1 \parallel X_1)} \oplus \boxed{\mathcal{E}_K(2 \parallel Y_1)} \oplus \\ &\quad \boxed{\mathcal{E}_K(1 \parallel X_1)} \oplus \mathcal{E}_K(2 \parallel Y_2) \oplus \\ &\quad \mathcal{E}_K(1 \parallel X_2) \oplus \boxed{\mathcal{E}_K(2 \parallel Y_1)} \\ &= \mathcal{E}_K(2 \parallel Y_2) \oplus \mathcal{E}_K(1 \parallel X_2) && \text{cancel duplicate XORs (highlighted)} \\ &= \mathcal{MAC}(X_2 \parallel Y_2) \end{aligned}$$

Since we haven't queried the tagging algorithm with this particular message, it becomes a valid pairing that breaks the scheme. It's also trivially a reasonable attack, requiring only  $q_t = 3$  queries to the tagging algorithm,  $\mu = 3$  messages, and the time it takes to perform 3 XORs (if we don't count the internals of  $\mathcal{MAC}$ ).

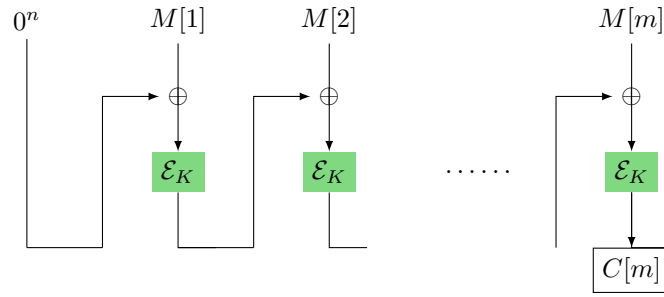
## 2.3 Mode of Operation: CBC-MAC

We state an important fact without proof; it acts as our inspiration for this section:

**Theorem 2.1.** *Any PRF function yields a UF-CMA secure MAC.*

This means that any secure blockcipher (like AES) can be used as a MAC. However, they only operate on short input messages. Can we extend our [Modes of Operation](#) to allow MACs on arbitrary-length messages?

Enter CBC-MAC, which looks remarkably like CBC mode for encryption (see [subsection 1.1.2](#)) but disregards all but the last output ciphertext. Given an  $n$ -bit block cipher,  $\mathcal{E} : \{0, 1\}^k \times \{0, 1\}^n \mapsto \{0, 1\}^n$ , the output message space is  $\text{MsgSp} = \{0, 1\}^{mn}$ , **fixed**  $m$ -block messages (obviously  $m \geq 1$ ).



**Figure 2.2:** The CBC-MAC authentication mode.

To reiterate, this scheme is secure under [UF-CMA](#) only for a fixed message length across all messages. That is, we can't send messages that are longer or shorter than some predefined multiple of  $n$  bits.

**Theorem 2.2.** *The CBC-MAC authentication scheme is secure if the underlying blockcipher is secure.*

*More specifically, for any efficient adversary  $\mathcal{A}$ , there exists an adversary  $\mathcal{B}$  with similar resources such that  $\mathcal{A}$ 's advantage is worse than  $\mathcal{B}$ 's:*

$$\text{Adv}_{\text{CBC-MAC}}^{\text{uf-cma}}(\mathcal{A}) \leq \text{Adv}_E^{\text{prf}}(\mathcal{B}) + \frac{m^2 q_A^2}{2^{n-1}}$$

*(the last term is an artifact of the [birthday paradox](#))*

This is an important limitation, and it will be enlightening for the reader to determine why variable-length messages break the CBC-MAC authentication scheme. There *are*, however, ways to extend CBC-MAC to allow variable-length messages, such as by prepending the length as the first message block.

# HASH FUNCTIONS

An essential part of modern cryptography, **hash functions** transform arbitrary-length input data to a short, fixed-size **digest**:

$$\mathcal{H} : \{0, 1\}^{<2^{64}} \mapsto \{0, 1\}^n$$

Some examples of modern hash functions include those in [Table 3.1](#). They should be pretty familiar: SHA-1 is used by `git` and SHA-3 is used by various cryptocurrencies like Ethereum.<sup>1</sup> They are used as building blocks for encryption, hash-maps, [blockchains](#), key-derivation functions, password-storage mechanisms, and more.

Function	Digest Size	Secure?
MD4	128	×
MD5	128	×
SHA-1	160	×
SHA-256	256	✓
SHA-3	224, 256, 384, 512	✓

**Table 3.1:** A list of some modern hash functions and their output digest length.

## 3.1 Collision Resistance

Not all hash functions are created equal. For example, here’s a valid hash function: just output the first  $n$  bits of the input as the digest. A **good** hash function tries to distribute its potential inputs uniformly across the output space to minimize *hash collisions*. In fact, most of the functions in [Table 3.1](#) above are considered **broken** from a cryptography perspective: collisions have been found.

Formally, a collision is a pair of messages from the domain,  $m_1 \neq m_2$ , such that  $H(m_1) = H(m_2)$ . Obviously, if the domain is larger than the range, there *must* be collisions (by the [pigeonhole principle](#)), but from the perspective of security, we want the probability of *creating* a collision to be very small. This is called being [collision resistant](#).

As we’ve done several times before, let’s formalize the notion of collision resistance with an experiment. If we try to approach this in the traditional way—define an oracle that outputs hashes, and defined some “collision resistance advantage” as the probability of finding two inputs that output the same hash—we immediately run into problems:

<sup>1</sup> Technically, Ethereum uses the KECCAK-256 hash function, which is the pre-standardized version of SHA-3. There are some interesting theories on the difference between the two: though the standardized version changes a [padding rule](#)—allegedly to allow better variability in digest lengths—its underlying algorithm was [weakened to improved performance](#), casting doubts on its general-purpose security.

1. Since hash algorithms are public, there isn't really a key to keep secret and thus no oracle to construct.
2. Hash functions have collisions *by definition*, so the probability of finding one is *always* one. Even if we, as humans, don't *know* how to find the collision, this is a separate issue.

To get around this, we'll instead consider experiments on *families* of hash functions, where a “key” acts as a selector of specific instances from the family.

*This is unfortunate in some ways, because it distances us from concrete hash functions like SHA1. But no alternative is known.*

— Introduction to Modern Cryptography (pp. 141)

Formally, we define a family of hash functions as being:

$$\mathcal{H} : \{0, 1\}^k \times \{0, 1\}^m \mapsto \{0, 1\}^n$$

Then, the key is chosen randomly ( $k \xleftarrow{\$} \{0, 1\}^k$ ) and provided to the adversary (to enable actually running the hash functions), who tries to find two inputs that map to the same output.

### DEFINITION 3.1: Collision Resistance

A family of hash functions  $\mathcal{H}$  is considered **collision resistant** if an adversary's **cr-advantage**—the probability of finding a collision—on a randomly-chosen instance  $\mathcal{H}_k$  is small ( $\approx 0$ ).

$$\text{Adv}_{\mathcal{H}}^{\text{cr}}(\mathcal{A}) = \Pr[\mathcal{H}_k(x_1) = \mathcal{H}_k(x_2)] \quad \text{where } x_1 \neq x_2$$

This avoids the aforementioned problem of adversaries hard-coding *a priori*-known collisions to specific instances. There's still a bit of a gap between this theoretical security definition and practice, since hash functions still typically don't have keys.

**Practice: Find a Collision** Do blockciphers make good hash functions? By their very nature (being a permutation), their output is collision resistant. However, they don't accept arbitrary-length inputs, and a **mode of operation** will still render arbitrary-length *outputs* while we need a fixed-size digest as a result.

Consider a simple way of combining AES inputs: XOR the individual output blocks. For simplicity, we'll limit ourselves to two AES blocks, so our function family is:

$$\mathcal{H} : \{0, 1\}^k \times \{0, 1\}^{256} \mapsto \{0, 1\}^{128}$$

Is  $\mathcal{H}$  collision resistant?

Obviously not. It's actually quite trivial to get the exact same digest, since  $x_1 \oplus x_1 = 0$ . That is, we pass the same 128-bit block in twice:

$$\begin{aligned} \text{Let } x &\xleftarrow{\$} \{0, 1\}^{128} \text{ and } m = x \parallel x : \\ \mathcal{H}_k(m) &= \text{AES}(x) \oplus \text{AES}(x) \\ &= c \oplus c = 0 \end{aligned}$$

Notice that this is extremely general-purpose, finding  $2^{128}$  messages that all collide to the same value of zero.

## 3.2 Building Hash Functions

Suppose we had a hash function that compressed short inputs into even-shorter outputs:

$$\mathcal{H}_s : \{0, 1\}^k \times \{0, 1\}^{b+n} \mapsto \{0, 1\}^n$$

where  $b$  is relatively small. We can use a technique called the **Merkle-Damgård transform** to create a new compression function that operates on *much* larger inputs, on an arbitrary domain  $D$ :

$$\mathcal{H}_\ell : \{0, 1\}^k \times D \mapsto \{0, 1\}^n$$

The algorithm is straightforward and is formalized in [algorithm 3.1](#). It's used by many modern hash function families, including the MD and SHA families. Visually, it looks like [Figure 3.1](#): each “block” of the input message is concatenated with the hashed version of its previous block, then hashed again.



**Figure 3.1:** A visualization of the Merkle-Damgård transform.

---

**ALGORITHM 3.1:** The Merkle-Damgård transform, building an arbitrary-length compression function using a limited compression function.

**Input:**  $h(\cdot)$ , a limited-range compression function operating on  $b$ -bit inputs.

**Input:**  $M$ , the arbitrary-length input message to compress.

**Result:**  $M$  compressed to an  $n$ -bit digest.

```

 $m := \|M\|_b$                                      // the number of  $b$ -bit blocks in  $M$ 
 $M[m+1] := \langle M \rangle$                              // the last block is message size
 $V[0] := 0^n$ 
for  $i = 1, \dots, m+1$  do
   $V[i] := h(M[i] \| V[i-1])$ 
end
return  $V[m+1]$ 

```

---

The [good news](#) of this transform is the following:

**Theorem 3.1.** *If a short compression function  $\mathcal{H}_s$  is [collision resistant](#), then a longer compression function  $\mathcal{H}_\ell$  composed from the [Merkle-Damgård transform](#) will also be collision resistant.*

This means we can build up complex hash functions from simple primitives, as long as those primitives can make promises about collision resistance. Can they, though?

**Birthday Attacks** Recall the [birthday paradox](#): as the number of samples from a range increases, the probability of any two of those samples being equal grows rapidly (there’s a 95% chance that two people at a 50-person party will have the same birthday).

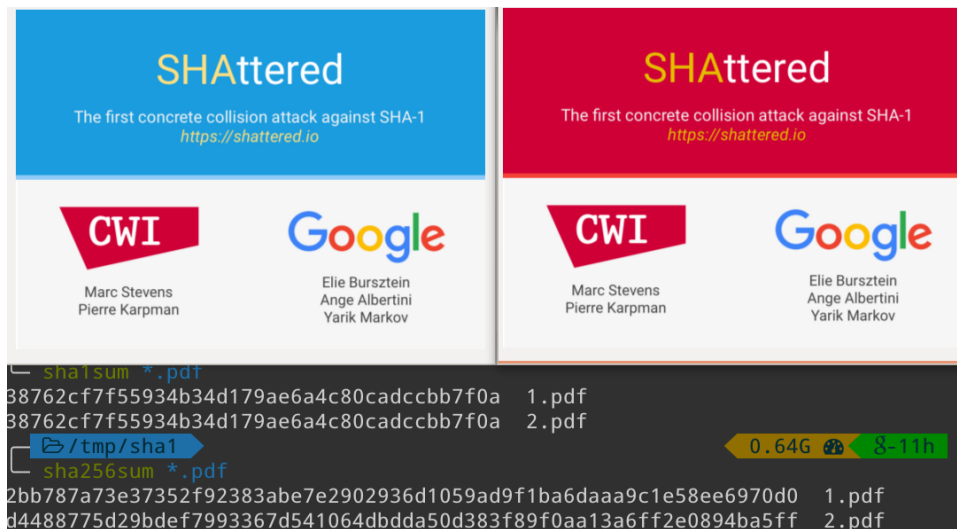
A hash function is **regular** if every range point has the same number of pre-images (that is, if every output has the same number of possible inputs). For such a function, the “birthday attack” finds a



collision in  $\approx 2^{n/2}$  trials. For a hash function that is *not* regular, such an attack could succeed even sooner.

Thorough research into the modern hash functions (for which  $n \geq 160$ , large-enough to protect against birthday attacks) suggests that they are “close to regular.”<sup>2</sup> Thus, we can safely use them as building blocks for Merkle-Damgård.

**Attacks in Practice: SHAttered** A collision for the SHA-1 hash was found in February of 2017, breaking the hash function in practice after it was broken theoretically in 2005: two PDFs resolved to the same digest. The attack took  $2^{63} - 1$  computations; this is 100,000 faster than the birthday attack.



**Figure 3.2:** The two PDFs in the SHAttered attack and their resulting, identical digests. More details are available on the attack’s [site](https://shattered.io) (because no security attack is complete without a trendy title and domain name).

## QUICK MAFFS: Function Nomenclature

Because mathematicians like to use opaque terminology, it’s worth expanding upon the nomenclature for clarity.

The **domain** and **range** of a function should be familiar to us: the domain is a set of inputs and the range (also confusingly called the **codomain** sometimes) is the set of possible outputs for that input. Formally,

$$R = \{f(d) : d \in D\}$$

**Example** If the domain of  $f(x) = x^2$  is all real numbers ( $D = \mathbb{R}$ ), its range is all positive reals  $R = \mathbb{R}^+$  (we’ll treat 0 as a positive number for brevity).

These terms refer to the function as a whole; we chose the input domain, and the range is the corresponding set of outputs. However, it’s also useful to examine subsets of the

<sup>2</sup> Much like the conjecture that AES is **PRF** secure, this is thus far unproven. As we’ll see later, neither are the security assumptions behind asymmetric cryptography (e.g. “factoring is hard”). Overall, these conjectures on top of conjectures unfortunately do not inspire much confidence in the overall state of security, yet it’s the best we can do.

range and ask the inverse question. That is, what's the domain that corresponds to a particular set of values?

**Example** Given  $f(x) = x$  (the diagonal line passing through the origin), for what subset of the domain is  $f(x) > 0$ ? Obviously, when  $x > 0$ .

This subset is called the **preimage**. Namely, given a subset of the range,  $S \subseteq R$ , its preimage is the set of inputs that corresponds to it:

$$P = \{x \mid f(x) \in S\}$$

In summary, a preimage of some outputs of a function is **the set of inputs that result** in that output.

### 3.3 One-Way Functions

Hash functions that are viable for cryptographic use must being **one-way functions**: they must be easy to compute in one direction, but (very) hard to compute in reverse. That is, given a hash, it should be hard to figure out what input resulted in that hash. Informally, a hash function is one-way if, given  $y$  and  $k$ , it is infeasible to find  $y$ 's preimage under  $h_k$ .

As usual, we'll define this notion formally with an experiment. Given a hash function family,  $\mathcal{H} : \{0, 1\}^k \times D \mapsto \{0, 1\}^n$ , we'll have an oracle randomly-select a key and an input value, providing the adversary with its resulting hash and the key (so they can run hash computations).

$$\text{Let } k \xleftarrow{\$} \{0, 1\}^k \text{ and } x \xleftarrow{\$} D : \\ y = \mathcal{H}_k(x)$$

Now, the adversary wins if they can produce a  $x'$  for which  $\mathcal{H}_k(x') = y$ . Note that  $x'$  does not need to be  $x$ , only in the preimage of  $y$ , so this security definition somewhat-includes being **collision resistant**.

#### DEFINITION 3.2: One-Way Function

A family of hash functions  $\mathcal{H}$  is considered **one-way** if an adversary's **ow-advantage**—the probability of finding the randomly-chosen input,  $x$ , from its digest  $y$ —on a randomly-chosen instance  $\mathcal{H}_k$  is small ( $\approx 0$ ).

$$\text{Adv}_{\mathcal{H}}^{\text{ow}}(\mathcal{A}) = \Pr[\mathcal{H}_k(x') = y]$$

Given our two security properties for a hash function, do either of them imply the other? That is, are either of these true?

$$\begin{aligned} \text{collision resistance} &\implies \text{one-wayness} \\ \text{one-wayness} &\implies \text{collision resistance} \end{aligned}$$

**CR  $\implies$  OW** For functions in general (not necessarily hash functions), collision resistance **does not** imply one-wayness. Consider the trivial identity function: since it's one-to-one, it's collision resistant by definition, but it's obviously not one-way. However, for functions that compress their input (e.g. hash functions), it **does** imply it!

**OW**  $\Rightarrow$  **CR** This implication **does not** hold in all cases. We can easily do a “disproof by counterexample”: suppose we have a one-way hash function  $g$ . We construct  $h$  to hash an  $n$ -bit string by delegating to  $g$ , sans the last input bit:

$$h(x_1x_2 \cdots x_n) = g(x_1x_2 \cdots x_{n-1})$$

Since  $g$  was one-way,  $h$  is also one-way. However, it’s obviously not collision resistant, since we know that when given any  $n$ -bit input  $m$

$$h(m_1m_2 \cdots m_{n-1}0) = h(m_1m_2 \cdots m_{n-1}1)$$

### 3.4 Hash-Based MACs

We’ve come full-circle. Can we use a **cryptographically-secure hash function**—a hash function that is both **collision resistant** and **one-way**—to do authentication? Obviously, we can’t use hash functions directly since there is no key.

However, can we somehow include a key within our hash input? More importantly, can we devise a *provably*-secure hash-based **message authentication code**? Enter the aptly-named **HMAC**, visualized below in [Figure 3.3](#).

First, some definitions.  $H$  is our hash function instance that maps from an arbitrary domain to an  $n$ -bit digest:

$$H : \mathcal{D} \mapsto \{0, 1\}^n$$

Then, we have a secret key  $K$ , and we denote  $B \geq n/8$  as the *byte*-length of the message block<sup>3</sup>. The HMAC is computed using a nested structure, mixing the key with some constants. Namely, we define

$$K_o = \text{opad} \oplus K \parallel 0^{8B-n} \quad K_i = \text{ipad} \oplus K \parallel 0^{8B-n}$$

where opad and ipad are hex-encoded constants:

$$\text{opad} = 0x \underbrace{5C5C5C \dots}_{\text{repeated } B \text{ times}} \quad \text{ipad} = 0x \underbrace{363636 \dots}_{\text{repeated } B \text{ times}}$$

Then, the final tag is a simple combination of the transformed keys:

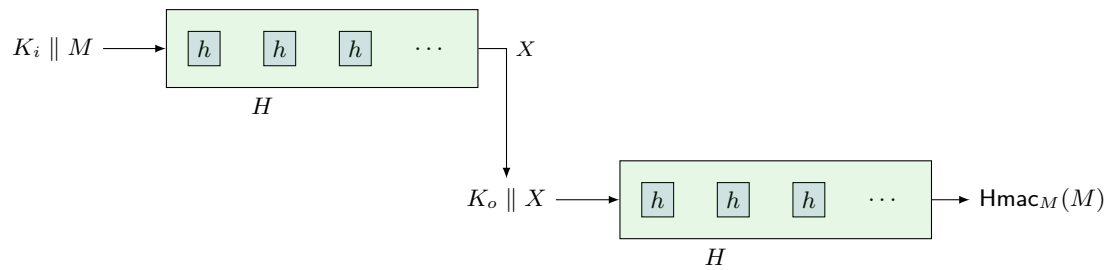
$$\text{Hmac}_K(K) = H(K_o \parallel H(K_i \parallel M))$$

The specific constants are chosen to simplify the proof of security, having no bearing on the security itself.

HMAC is easy to implement and fast to compute; it is a core part of many standardized cryptographic constructs. Its useful both as a **message authentication code** and as a **key-derivation function** (which we’ll discuss later in asymmetric cryptography).

**Theorem 3.2.** *HMAC is a PRF assuming that the underlying compression function  $H$  is a PRF.*

<sup>3</sup> Typically,  $B = 64$  for modern hash functions like MD5, SHA-1, SHA-256, and SHA-512.



**Figure 3.3:** A visualization of the two-tiered structure of HMAC, the standard keyed message authentication code scheme.

# AUTHENTICATED ENCRYPTION

In an ideal world, we would be able to ensure message [confidentiality](#), message [integrity](#), and sender [authenticity](#) all at once. This is the goal of [authenticated encryption](#) (AE) within the realm of symmetric cryptography. Until this point, we’ve seen how to achieve data privacy and confidentiality ([IND-CPA](#) and [IND-CCA](#) security) as well as authenticity and integrity ([UF-CMA](#) security) *separately*.

The syntax and notation for authenticated encryption schemes is almost identical to those we’ve been using previously; simply refer to [section 2.1](#) to review that. We have a message space, a key generation algorithm, and encryption/decryption algorithms. The only difference is that now it’s possible for the decryption algorithm to reject an input entirely. We’ll use this symbol:  $\perp$ , in algorithms and such to indicate this.

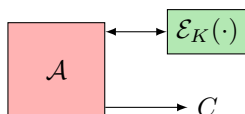
## 4.1 INT-CTXT: Integrity of Ciphertexts

Though the confidentiality definitions from before still apply, we’ll need a new UF-CMA-equivalent for encryption, since MACs make no guarantees about encryption. The intuition will still be same, except there’s the additional requirement that the adversary produces a valid ciphertext.

### DEFINITION 4.1: INT-CTXT Security

A scheme  $\mathcal{SE} = (\text{KeySp}, \mathcal{E}, \mathcal{D})$  is considered secure under INT-CTXT if an adversary’s [INT-CTXT advantage](#)—the probability of producing a valid, forged ciphertext—is small ( $\approx 0$ ):

$$\text{Adv}_{\mathcal{SE}}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathcal{A} \rightarrow C : \mathcal{D}_K(C) \neq \perp \text{ and } C \text{ wasn't received from } \mathcal{E}_K(\cdot)]$$



In one sentence, in the [INT-CTXT](#) experiment the adversary  $\mathcal{A}$  is tasked with outputting a valid ciphertext  $C$  that was never received from the encryption oracle. An authenticated encryption scheme will thus be secure from forgery under INT-CTXT (integrity) and secure from snooping under IND-CCA (confidentiality).

Thankfully, we can abuse the following fact to make constructing such a scheme much easier:

[Module 7](#)

**Theorem 4.1.** *If a symmetric encryption scheme is secure under IND-CPA and INT-CTXT, it is also secure under IND-CCA.*

We can build a secure authentication encryption scheme by composing the basic encryption and MAC schemes we’ve already seen.

[Bellare & Namprempre, ‘00](#)

## 4.2 Generic Composite Schemes

[Wikipedia](#)

Given a symmetric encryption scheme and a [message authentication code](#), we can combine them in a number of ways:

- **MAC-then-encrypt**, in which you first MAC the plaintext, then encrypt the combined plaintext and MAC; this technique is used by the SSL protocol.
- **encrypt-and-MAC**, in which you encrypt the plaintext and then MAC the original *plaintext*; this is done in SSH.
- **encrypt-then-MAC**, in which you encrypt the plaintext and then MAC the *resulting ciphertext*; this is done by IPSec.

Analyzing the security of these approaches is a little involved. Ideally, much like the [Merkle-Damgård transform](#) guarantees collision resistance (see [Theorem 3.1](#)), it'd be nice if the security of the underlying components of a composite scheme made guarantees about the scheme as a whole. More specifically, can we build a composite authenticated encryption scheme  $\mathcal{AE}$  when given an IND-CPA symmetric encryption scheme  $\mathcal{SE} = (\mathcal{K}', \mathcal{E}', \mathcal{D}')$  and a PRF  $F$  that can act as a MAC (recall from [Theorem 2.1](#) that PRFs are UF-CMA secure).

**Key Generation** Keeping keys for [confidentiality](#) and [integrity](#) separate is incredibly important. This is called the **key separation principle**: one should always use distinct keys for distinct algorithms and distinct modes of operation. It's possible to do authenticated encryption without this, but it's far more error-prone.<sup>1</sup>

Thus our composite key generation algorithm will generate two keys:  $K_e$  for encryption and  $K_m$  for authentication.

$$\begin{aligned}\mathcal{K} : \quad K_e &\xleftarrow{\$} \mathcal{K}' \\ K_m &\xleftarrow{\$} \{0, 1\}^k \\ K &:= K_e \parallel K_m\end{aligned}$$

### 4.2.1 Encrypt-and-MAC

In this composite scheme, the plaintext is both encrypted and authenticated; the full message is the concatenated ciphertext and tag.

---

**ALGORITHM 4.1:** The encrypt-and-MAC encryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_m}, \mathcal{D}_{K_m})$

$C' \xleftarrow{\$} \mathcal{E}'_K(M)$   
 $T = F_{K_m}(M)$   
**return**  $C' \parallel T$

---



---

**ALGORITHM 4.2:** The encrypt-and-MAC decryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_m}, \mathcal{D}_{K_m})$

$M = \mathcal{D}'_K(C')$   
**if**  $T = F_{K_m}(M)$  **then**  
  | **return**  $M$   
**end**  
**return**  $\perp$

---

<sup>1</sup> The unique keys can still be derived from a single key via a [pseudorandom generator](#), such as by saying  $K_1 = F_K(0)$  and  $K_2 = F_K(1)$  for a [PRF secure](#)  $F$ . The main point is to keep them separate beyond that.

We want this scheme to be both IND-CPA and INT-CTXT secure. Unfortunately, **it provides neither**. Remember, PRFs are deterministic: by including the plaintext's MAC, we hurt the confidentiality definition and can break IND-CPA (via [Theorem 1.1](#)).<sup>2</sup>

### 4.2.2 MAC-then-encrypt

In this composite scheme, the plaintext is first tagged, then the concatenation of the tag and the plaintext is encrypted.

How's the security of this scheme? There's no longer a deterministic component, so it is IND-CPA secure; however, it does not guarantee integrity under INT-CTXT. We can prove this by counterexample if there are some *specific* secure building blocks that lead to valid forgeries.

---

**ALGORITHM 4.3:** The MAC-then-encrypt encryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$

$T = F_{K_m}(M)$

$C' \xleftarrow{\$} \mathcal{E}'_{K_e}(M \parallel T)$

**return**  $C'$

---



---

**ALGORITHM 4.4:** The MAC-then-encrypt decryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$

$M \parallel T = \mathcal{D}'_{K_e}(C')$

**if**  $T = F_{K_m}(M)$  **then**

**return**  $M$

**end**

**return**  $\perp$

---

The counterexample for this is a little bizzare and worth exploring; it gives us insight into how hard it truly is to achieve security under these rigorous definitions. We'll first define a new IND-CPA encryption scheme:

$$\mathcal{SE}'' = \{\mathcal{K}', \mathcal{E}'', \mathcal{D}''\}$$

Then, we'll define  $\mathcal{SE}'$  as an encryption scheme that is *also* IND-CPA secure, that *uses*  $\mathcal{SE}''$ , but enables trivial forgeries by appending an ignorable bit to the resulting ciphertext:

$$\mathcal{SE}' = \{\mathcal{K}', \mathcal{E}', \mathcal{D}'\}$$

$$\mathcal{E}'_K(M) = \mathcal{E}''_K(M) \parallel 0$$

$$\mathcal{D}'_K(M \parallel b) = \mathcal{D}''_K(M)$$

Obviously, now both  $C \parallel 0$  and  $C \parallel 1$  decrypt to the same plaintext, and this means that an adversary can easily create forgeries. Weird, right? This example, silly as though it may be, is enough to demonstrate that MAC-then-encrypt cannot make guarantees about INT-CTXT security *in general*.

### 4.2.3 Encrypt-then-MAC

In our last hope for a generally-secure scheme, we will encrypt the plaintext, then add a tag based on the resulting ciphertext.

---

<sup>2</sup> Specifically, consider submitting two queries to the left-right oracle:  $LR(0^n, 1^n)$  and  $LR(0^n, 0^n)$ . The *tags* for the  $b = 0$  case would match.

**CAVEAT: Timing Attacks**

Notice the key, extremely important nuance of the decryption routine in [algorithm 4.6](#): the message is decrypted regardless of whether or not the tag is valid. From a performance perspective, we would ideally check the tag first, right? Unfortunately, this leads to the potential for an advanced **timing attack**: decryption of invalid messages now *takes less time* than valid ones, and this lets the attacker to learn secret information about the scheme. Now, they can differentiate between an invalid tag and an invalid ciphertext.

With this scheme, we get *both* security under IND-CPA and INT-CTXT, and by [Theorem 4.1](#), also under IND-CCA.

**Theorem 4.2.** *Encrypt-then-MAC is the **only** generic composite scheme that provides confidentiality under [IND-CCA](#) as well as integrity under [INT-CTXT](#) regardless of the underlying cryptographic building blocks provided that they are secure. Namely, it holds as long as the base encryption is [IND-CPA](#) secure and  $F$  is [PRF](#) secure.*

A common combination of primitives is AES-CBC (which we proved to be secure in [Theorem 1.4](#)) and HMAC-SHA-3 (which is conjectured to be a [cryptographically-secure hash function](#)).

---

**ALGORITHM 4.5:** The encrypt-then-MAC encryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$

**Input:**  $M$ , an input plaintext message.

$C \xleftarrow{\$} \mathcal{E}'_{K_e}(M \parallel T)$

$T = F_{K_m}(C)$

**return**  $C \parallel T$

---



---

**ALGORITHM 4.6:** The encrypt-then-MAC decryption algorithm.

**Input:**  $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$

**Input:**  $C$ , an input ciphertext message.

$M = \mathcal{D}'_{K_e}(C)$

**if**  $T = F_{K_m}(C)$  **then**

**return**  $M$

**end**

**return**  $\perp$

---

#### 4.2.4 In Practice...

It's important to remember that the above results hold *in general*; that is, they hold for arbitrary secure building blocks. That does not mean it's impossible to craft a *specific* AE scheme that holds under a generally-insecure composition method.

Protocol	Composition Method	In general...	In this case...
SSH <sup>3</sup>	Encrypt-and-MAC	Insecure	Secure
SSL	MAC-then-encrypt	Insecure	Secure
IPSec	Encrypt-then-MAC	Secure	Secure
WinZip	Encrypt-then-MAC	Secure	<b>Insecure</b>

**Table 4.1:** Though EtM is a provably-secure generic composition method, that does not mean the others can't be used to make secure AE schemes. Furthermore, that does not mean it's impossible to do EtM wrong! (*cough* WinZip)



### 4.2.5 Dedicated Authenticated Encryption

Rather than using generic composition of lower-level building blocks, could we craft a [mode of operation](#) or something that has AE guarantees in mind from the get-go?

The answer is yes, and the [offset codebook](#) mode is such a scheme. It's a one-pass, heavily parallelizable scheme.<sup>4</sup>

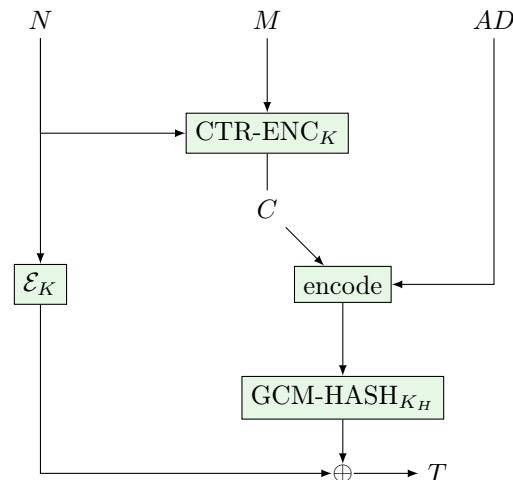
## 4.3 AEAD: Associated Data

The idea of [associated data](#) goes hand-in-hand with authenticated encryption. Its purpose is to provide data that is *not* encrypted (either because it does not need to be secret, or because it *cannot* be encrypted), but **must** be authenticated. Schemes are technically deterministic, but they are still based on [initialization vectors](#) and thus provide similar guarantees in functionality and security.

### 4.3.1 GCM: Galois/Counter Mode

This is probably the most well-known [AEAD](#) scheme. It's widely used and is most famously used in TLS, the backbone of a secure Web.

The scheme is made up of several building blocks. The GCM-HASH is a polynomial-based hash function and the hashing key,  $K_H$ , is derived from the “master” key  $K$  using the [block cipher](#)  $\mathcal{E}$ . It can be used as a MAC and is heavily standardized; its security has been proven under the reasonable assumptions we've seen for the building blocks.



**Figure 4.1:** A visualization of the Galois/Counter mode (GCM) of AEAD encryption.

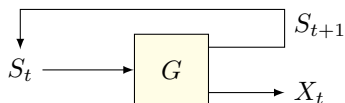
<sup>4</sup> It was designed by Phillip Rogaway, one of the authors for the [lecture notes](#) on cryptography.

# STREAM CIPHERS

This chapter introduces a paradigm shift in the way we’ve been constructing ciphertexts. Rather than encrypting block-by-block using a specific **mode of operation**, we’ll instead be encrypting bit-by-bit with a stream of gibberish. Previously, we needed our input plaintext to be a multiple of the block size; now, we can truly deal with arbitrarily-length inputs without worrying about padding. This will actually be reminiscent of **one-time pads**: a **pseudorandom generator** (or PRG) will essentially be a function that outputs an infinitely-long one-time pad, and a **stream cipher** will use that output to encrypt plaintexts.

## 5.1 Generators

In general, a **stateful generator**  $G$  begins with some initial state  $S_{t=0} \xleftarrow{\$} \{0,1\}^n$  called the **seed**, then uses the output of itself as input to its next run. The sequence of outputs over time,  $X_0X_1X_2\cdots$  should be pseudorandom for a pseudorandom generator: reasonably unpredictable and tough to differentiate from true randomness.



We’ll use the shorthand notation:

$$(X_0X_1\cdots X_m, S_t) = G(S_0, m)$$

to signify running the generator  $m$  times with the starting state  $S_0$ , resulting in an  $m$ -length output and a new state  $S_t$ . This construction is the backbone of all of the instances where we’ve used  $\xleftarrow{\$}$  previously to signify choosing a random value from a set. Pseudorandom generators (PRGs) are used to craft **initialization vectors**, keys, oracles, etc.

### 5.1.1 PRGs for Encryption

Using a PRG for encryption is very easy: just generate bits and use them as a one-time pad for your messages. The hard part is **synchronization**: both you and your recipient need to start with the same seed state to decrypt each others’ messages. This is the basis behind a **stream cipher**.

## 5.2 Evaluating PRGs

Creating a generator with unpredictable, random output is quite difficult. Functions build on **linear congruential generators** (LRGs) and **linear feedback shift registers** (LFSRs) have good distributions (equal numbers of 1s and 0s in the output) but are predictable given enough output.

However, stream ciphers like **RC4** (the 4<sup>th</sup> Rivest cipher) and **SEAL** (software-optimized encryption algorithm) can make these unpredictability guarantees.

As is tradition, we'll need a formal definition of security to analyze PRGs. We'll call this **INDR** security: **ind**istinguishability from **r**andomness. The adversarial experiment is very simple: an oracle picks a secret seed state,  $S_0$  and generates an  $m$ -bit output stream both from the PRG and from truly-random source:

$$(\mathbf{X}^1, S_t) = (X_0^1 X_1^1 \cdots X_m^1, S_t) = G(S_0, m)$$

$$\mathbf{X}^0 = X_0^0 X_1^0 \cdots X_m^0 \xleftarrow{\$} \{0, 1\}^m$$

It then picks a challenge bit  $b$  and gives  $\mathbf{X}^b$  to the attacker. If s/he can output their guess,  $b'$ , such that  $b' = b$  reliably, they win the experiment and  $G$  is not secure under INDR.

### DEFINITION 5.1: INDR Security

A pseudorandom generator  $G$  is considered INDR secure if an efficient adversary's **INDR advantage**—that is their ability to differentiate between the PRG's bitvector  $\mathbf{X}^1$  and the truly-random bitvector  $\mathbf{X}^0$ —is small (near-zero). The advantage is defined as:

$$\text{Adv}_G^{\text{indr}}(\mathcal{A}) = \Pr[b' = 1 \text{ for Exp}_1] - \Pr[b' = 1 \text{ for Exp}_0]$$

## 5.3 Creating Stream Ciphers

Since pseudorandom *functions* (and hence **block ciphers**) output random-looking data and can be keyed with state, they are an easy way to create a reliable, provably-secure pseudorandom generator. All we need to do is continually increment a randomly-initialized value.

**Theorem 5.1** (the ANSI X9.17 standardized PRG). *If  $E$  is a secure pseudorandom function:*

$$E : \{0, 1\}^k \times \{0, 1\}^n \mapsto \{0, 1\}^n$$

*then  $G$  is an INDR-secure pseudorandom generator as defined:*

---

**ALGORITHM 5.1:**  $G(S_t)$ , a PRG based on the CTR mode of operation.

**Input:**  $S_t$ , the current PRG input.

**Result:**  $(X, S_{t+1})$ , the pseudorandom value and the new PRG state.

```

K || V = S_t                                     /* extract the state */
X = E_K(V)
V = E_K(X)
return (X, (K, V))

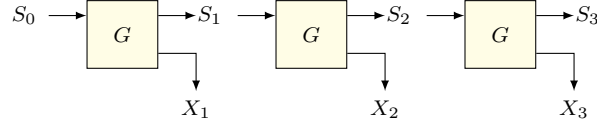
```

---

Interestingly-enough, though this construction is provably-secure under INDR, it's not immune to attacks. The security definition does not capture all vectors.

### 5.3.1 Forward Security

The idea behind **forward secrecy** (also called forward security) is that past information should be kept secret even if future information is exposed.



Suppose an adversary somehow gets access to  $S_2$ . Obviously, they can now derive  $X_3$ ,  $X_4$ , and so on, but can they compute  $X_1$  or  $X_2$ , though? A scheme that preserves forward secrecy should say “no.”

The scheme presented in [algorithm 5.1](#), though secure under INDR does not preserve forward secrecy. Leaking any state  $(K, V_t)$  lets the adversary construct the entire chain of prior states if they have been capturing the entire history of generated  $X_{0..t}$  values.

Consider a simple forward-secure pseudorandom generator: regenerate the key anew on every iteration.

---

**ALGORITHM 5.2:**  $G(K)$ , a forward-secure pseudorandom generator.

**Input:**  $S_t$ , the current PRG input.

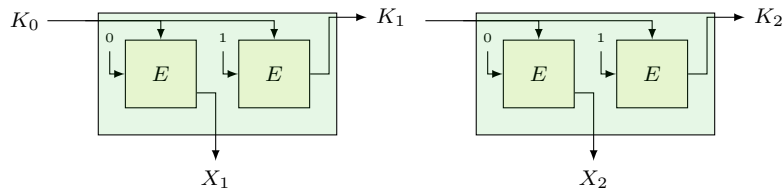
**Result:**  $(X, S_{t+1})$ , the pseudorandom value and the new PRG state.

$X = E_K(0)$

$K = E_K(1)$

**return**  $(X, K)$

---



**Figure 5.1:** A visualization of a PRG with forward security.

#### NOMENCLATURE: Forward Secrecy

In the cryptographic literature and community, **perfect** forward secrecy is when even exposing the *very next* state reveals no information about the past. Often this is expensive (either in terms of computation or in communicating key exchanges), and so forward secrecy generally refers to a regular cycling of keys that isolates the post-exposed vulnerability interval to certain (short) time periods.

### 5.3.2 Considerations

To get an unpredictable PRG, you need an unpredictable key for the underlying PRF. This is the [seed](#), and it causes a bit of a chicken-and-egg problem. We need random values to generate (pseudo)random values.

Entropy pools typically come from “random” events from the real world like keyboard strokes, system events, even CPU temperature. Then, seeds can be pulled from this entropy pool to seed PRGs.

Seeding is not exactly a cryptographic problem, but it’s an important consideration when using PRGs and stream ciphers.

# COMMON IMPLEMENTATION MISTAKES

Now that we’ve covered symmetric cryptography to a reasonable degree of rigor, it’s useful to cover many of the common pitfalls, missed details, and other implementation mistakes that regularly lead to gaping cryptographic security holes in the real world.

**Primitives** There are far more primitives that don’t work compared to those that work. For example, using [block ciphers](#) with small block sizes or small key spaces are vulnerable to [exhaustive key-search](#) attacks, not even to mention their vulnerability to the [birthday paradox](#). Always check NIST and recommendations from other standards committees to ensure you’re using the most well-regarded primitives.

**(Lack of) Security Proofs** Using a [mode of operation](#) with no proof of security—or worse, modes with proofs of *insecurity*—is far too common. Even ECB mode is used way more often than it should be. The fact that [AES](#) is a secure block cipher is often a source of false confidence.

**Security Bounds** Recall that we proved that the CTR mode of operation (see [Figure 1.4](#)) had the following adversarial advantage:

$$\text{Adv}_{\text{CTR}}^{\text{ind-cpa}}(\mathcal{A}) \leq \text{Adv}_{\text{E}}^{\text{prf}}(\mathcal{B}) \cdot \frac{q^2}{2^{L+1}}$$

Yet if we use constants that are far too low, this becomes easily achievable. The WEP security protocol for WiFi networks used  $L = 24$ . With  $q = 4096$  (trivial to do), the advantage becomes  $1/2$ ! In other words, the IVs are far too short<sup>1</sup> to provide any semblance of security from a reasonably-resourced attacker.

**Trifecta** Just because you have achieved [confidentiality](#), you have not necessarily achieved [integrity](#) or [authenticity](#). Not keeping these things in mind leads to situations where false assumptions are made.

**Implementation** Given a provable scheme, you must implement it *exactly* to achieve the security guarantees. This simple rule has been broken many times before: Diebold voting machines using an all-zero IV, Excel didn’t regenerate the random IV for every message (just once), and many protocols use the previous ciphertext block as the IV for the next one. These mistakes quickly break [IND-CPA](#) security.

**Security Proofs** As we’ve seen, we often need to extend our security definitions to encompass more sophisticated attacks (like [IND-CCA](#) over IND-CPA). Thus, even using a provably-secure scheme does not absolve you of an attack surface. For example, the [Lucky 13 attack](#) used a side-channel

---

<sup>1</sup> It’s *so* easy to break WEP-secured WiFi networks; I did it as a kid with a [\\$30 USB adapter](#) and 15 minutes on [Backtrack Linux](#).

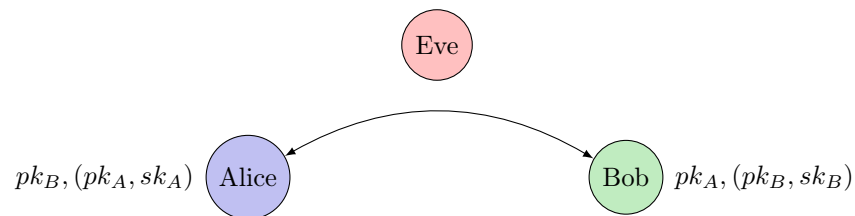
[timing attack](#) to break TLS. The security definitions we've recovered did not consider an attacker being able to the difference between decryption and MAC verification failures, or how fragmented ciphertexts (where the received doesn't know the borders between ciphertexts) are handled.

# PART II

---

## ASYMMETRIC CRYPTOGRAPHY

This class of algorithms is built to solve the **key distribution** problem. Here, secrets are only known to one party; instead, a key  $(pk, sk)$  is broken into two mathematically-linked components. There is a **public key** that is broadcasted to the world, and a **secret key** (also called a **private key**) that must be kept secret.



Asymmetric cryptography is often used to mutually establish a secret key (without revealing it!) for use with faster symmetric key algorithms. It's a little counterintuitive: two strangers can “meet” and “speak” publicly, yet walk away having established a mutual secret.

### Contents

<b>7 Overview</b>	<b>49</b>
<b>8 Number Theory</b>	<b>51</b>
<b>9 Encryption</b>	<b>61</b>
<b>10 Digital Signatures</b>	<b>75</b>
<b>11 Secret Sharing</b>	<b>84</b>
<b>Index of Terms</b>	<b>90</b>



# OVERVIEW

We need to translate some things over from the world of symmetric encryption to proceed with our same level of rigor and analysis as before, this time applying our security definitions to asymmetric encryption schemes.

## 7.1 Notation

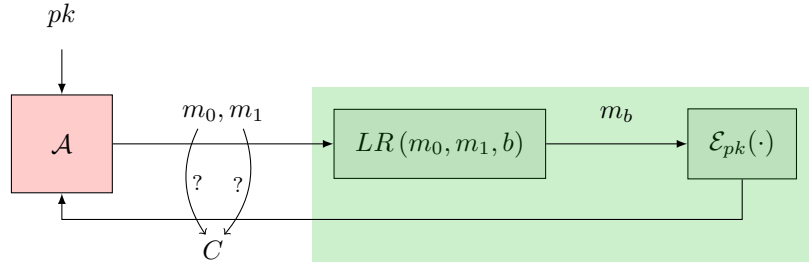
An asymmetric encryption scheme is similarly defined by an encryption and decryption function pair as well as a key generation algorithm. Much like before, we denote these as  $\mathcal{AE} = (\mathcal{E}, \mathcal{D}, \mathcal{K})$ .

The key is now broken into two components: the **public key** (shareable) and the **private key** (secret). These are typically composed as:  $K = (pk, sk)$ .

## 7.2 Security Definitions

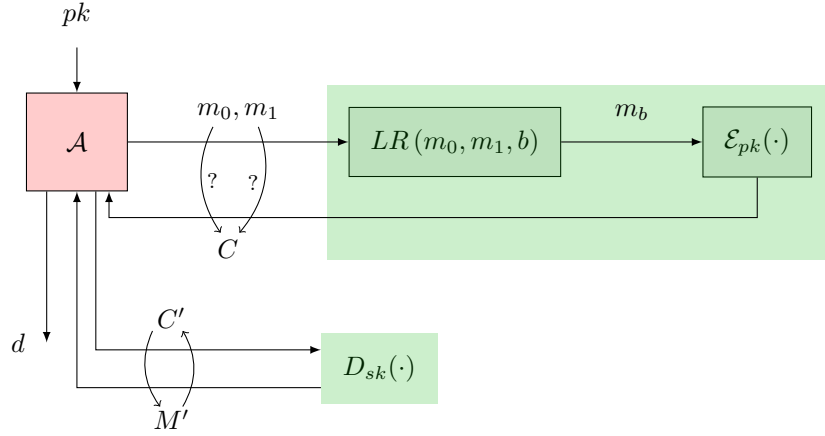
Our definitions of **IND-CPA** and **IND-CCA** security will be very similar to the way they were defined previously; the main difference is the (obvious) introduction of the private- and public-key split, as well as the fact that we can be more precise about what “reasonable” attacker resources are. Just to be perfectly precise, we’ll reiterate the new definitions here.

**Asymmetric IND-CPA** The following figure highlights IND-CPA security for asymmetric schemes (see Figure 1.6 for the original, symmetric version).



Notice that the encryption is under the public key  $pk$ ; however, since  $\mathcal{E}$  should be non-deterministic, this does not cause problems for the security definition. The scheme is IND-CPA secure if any adversary  $\mathcal{A}$ 's advantage is negligible with resources polynomial in the security parameter (typically the length of the key). This latter differentiation is what makes the definition more specific than for symmetric encryption schemes: our sense of a “reasonable” attacker is limited to polynomial algorithms.

**Asymmetric IND-CCA** For chosen ciphertext attacks, we keep the same restriction as before: the attacker cannot query the decryption oracle with ciphertexts s/he acquired from the encryption oracle.



Much like before, a scheme being IND-CCA implies it's also IND-CPA (recall the inverse direction of [Theorem 4.1](#)).

**Theorem 7.1.** *Let  $\mathcal{AE} = (\mathcal{E}, \mathcal{D}, \mathcal{K})$  be an asymmetric encryption scheme. For an IND-CPA adversary  $\mathcal{A}$  who makes at most  $q$  queries to the left-right oracle, there exists another adversary  $\mathcal{A}'$  with the same running time that only makes one query. Their advantages are related as follows:*

$$Adv_{\mathcal{AE}}^{ind-cpa}(\mathcal{A}) \leq q \cdot Adv_{\mathcal{AE}}^{ind-cpa}(\mathcal{A}')$$

Essentially, this theorem states that a scheme that is secure against a single query is just as secure against multiple queries because the factor of  $q$  does not have a significant overall effect on the advantage.

# NUMBER THEORY

Modular arithmetic and other ideas from number theory are the backbone of asymmetric cryptography. Like the name implies, the foundational security principles rely on the asymmetry of difficulty in mathematical operations. For example, verifying that a number is prime is easy, yet factoring a product of primes is hard.

The RSA and modular arithmetic discussions in this chapter are ripped from my notes for *Graduate Algorithms* which also covers these topics; these sections may not align perfectly with lectures in terms of overall structure.

**Measuring Complexity** We're going to be working with massive numbers. Typically in computer science, we would compute the “time complexity” of something as simple as addition as taking constant time. This presumes, though, that the numbers in question fit within a single CPU register (which might allow up to 64-bit numbers, for example). Since this is no longer the case, we're actually going to need to factor this into our calculations.

Specifically, we'll be measuring complexity in terms of the number of bits in our numbers. For example, adding two  $n$ -bit numbers will have complexity  $\mathcal{O}(n)$ .

## Notation

- $\mathbb{Z}^+$  is the set of positive integers,  $\{0, 1, \dots\}$ .
- $\mathbb{Z}_N$  is the set of positive integers up to  $N$ :  $\{0, 1, \dots, N - 1\}$ .
- $\mathbb{Z}_N^*$  is the set of integers that are coprime with  $N$ , meaning their **greatest common divisor** is 1:

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

- $\varphi(N) = |\mathbb{Z}_N^*|$  is Euler's **totient function**, measuring the size of the set of relatively prime numbers under  $N$ .

## 8.1 Groups

A **group** is just a set of numbers on which certain operations hold true. Let  $G$  be a non-empty set and let  $\cdot$  be some binary operation. Then,  $G$  is a **group** under said operation if:

- **closure**: the result of the operation should stay within the set:

$$\forall a, b \in G : a \cdot b \in G$$

- **associativity**: the order of operations should be swappable:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

- **identity**: there should be some element in the set such that binary operations on that element have no effect:

$$\forall a \in G : a \cdot 1 = 1 \cdot a = a$$

The 1 here is a placeholder for the identity element; it doesn't need to be the actual positive integer 1.

- **invertibility**: for any value in the set, there should be another unique element in the set such that their result is the identity element:

$$\forall a \in G, \exists b \in G : a \cdot b = b \cdot a = 1$$

This latter element  $b$  is called the **inverse** of  $a$ .

For example,  $\mathbb{Z}_N$  is a group under addition modulo  $N$ , and  $\mathbb{Z}_N^*$  is a group under multiplication modulo  $N$ . The **order** of a group is just its size.

**Property 8.1.** For a group  $G$ , if we let  $m = |G|$ , the order of the group, then:

$$\forall a \in G : a^m = 1$$

where 1 is the identity element. Furthermore,

$$\forall a \in G, i \in \mathbb{Z} : a^i = a^{i \bmod m} \quad (8.1)$$

**Example** These properties let us do some funky stuff with calculating seemingly-impossible values. Suppose we're working under  $\mathbb{Z}_{21}^*$ :

$$\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$$

Note that  $|\mathbb{Z}_{21}^*| = 12$ . What's  $5^{86} \bmod 21$ ? Simple:

$$\begin{aligned} 5^{86} \bmod 21 &= (5^{86 \bmod 12}) \bmod 21 \\ &= 5^2 \bmod 21 = 25 \bmod 21 \\ &= \boxed{4} \end{aligned}$$

**Subgroups** A subset  $S \subseteq G$  is called a **subgroup** if it's a group in its own right under the same operation that makes  $G$  a group. To test if  $S$  is a subgroup, we only need to check the invertibility property:

$$\forall x, y \in S : x \cdot y^{-1} \in S$$

Here,  $y^{-1}$  is the inverse of  $y$ . If  $S$  is a subgroup of  $G$ , then the order of  $S$  divides the order of  $G$ :  $|G| \bmod |S| = 0$ .

**Exponentiation** We define exponentiation as repeated application of the group operation. Note that this doesn't necessarily mean multiplication. For example, if we operate under the group  $\mathbb{Z}_N$  with addition, then exponentiation is repeated addition **not** repeated multiplication:  $2^3 = 6$  in such a group. This nuance won't really come up in our discussion since we are often concerned with the group  $\mathbb{Z}_N^*$  under multiplication, but it's worth noting.

## 8.2 Modular Arithmetic

For any two numbers,  $x \in \mathbb{Z}$  and  $N \in \mathbb{Z}^+$ , there is a *unique* quotient  $q$  and remainder  $r$  such that:  $Nq + r = x$ . Modular arithmetic lets us isolate the remainder:  $x \bmod N = r$ . Then, we say  $x \equiv y \pmod{N}$  if  $x/N$  and  $y/N$  have the same remainder.

An **equivalence class** is the set of numbers which are equivalent under a modulus. So mod 3 has 3 equivalence classes:

$$\begin{aligned} \dots, -6, -3, 0, 3, 6, \dots \\ \dots, -5, -2, 1, 4, 7, \dots \\ \dots, -4, -1, 2, 5, 8, \dots \end{aligned}$$

### 8.2.1 Running Time

Under modular arithmetic, there are different time complexities for common operations. The “cheat sheet” in Table 8.1 will be a useful reference for computing the overall running time of various asymmetric cryptography schemes.

Algorithm	Inputs	Running Time
integer division	$N > 0; a$	$\mathcal{O}( a  \cdot  N )$
modulus	$N > 0; a$	$\mathcal{O}( a  \cdot  N )$
extended GCD	$a; b; (a, b) \neq 0$	$\mathcal{O}( a  \cdot  b )$
mod addition	$N; a, b \in \mathbb{Z}_N$	$\mathcal{O}( N )$
mod multiplication	$N; a, b \in \mathbb{Z}_N$	$\mathcal{O}( N ^2)$
mod inverse	$N; a \in \mathbb{Z}_N^*$	$\mathcal{O}( N ^2)$
mod exponentiation	$N; n; a \in \mathbb{Z}_N^*$	$\mathcal{O}( n  \cdot  N ^2)$
exponentiation in $G$	$n; a \in G$	$\mathcal{O}( n )$ $G$ -operations

**Table 8.1:** A list of runtimes for common operations in asymmetric cryptography. Note that the syntax  $|x|$  specifies the number of bits needed to specify  $x$ , so you could say that  $|x| = \log_2 x$ .

We will consider a scheme to be secure if any adversary’s advantage is **negligible** relative to the security parameter (which is typically the number of bits in  $N$ ).

#### DEFINITION 8.1: Negligibility

A function  $g : \mathbb{Z}^+ \mapsto \mathbb{R}$  is **negligible** if it vanishes faster than the reciprocal of any polynomial. Namely, for every  $c \in \mathbb{Z}^+$ , there exists an  $n_c \in \mathbb{Z}$  such that  $g(n) \leq n^{-c}$  for all  $n \geq n_c$ .

This will become clearer with examples, but in essence we’re looking for inverse exponentials, so an advantage of  $2^{-k}$  is negligible.

### 8.2.2 Inverses

The **multiplicative inverse** of a number under a modulus is the value that makes their product 1. That is,  $x$  is the multiplicative inverse of  $z$  if  $zx \equiv 1 \pmod{N}$ . We then say  $x \equiv z^{-1} \pmod{N}$ .

Note that the multiplicative inverse does not always exist (in other words,  $\mathbb{Z}_N$  is not a group under multiplication); if it does, though, it's **unique**. They exist if and *only* if their greatest common divisor is 1, so when  $\gcd(x, N) = 1$ . This is also called being **relatively prime** or **coprime**.

### Greatest Common Divisor

The greatest common divisor of a pair of numbers is the largest number that divides both of them evenly. **Euclid's rule** states that if  $x \geq y > 0$ , then

$$\gcd(x, y) = \gcd(x \bmod y, y)$$

This leads directly to the **Euclidean algorithm**.

### Extended Euclidean Algorithm

**Bézout's identity** states that if  $n$  is the greatest common divisor of both  $x$  and  $y$ , then there are some "weight" integers  $a, b$  such that:

$$ax + by = n$$

These can be found using the **extended Euclidean algorithm** and are crucial in finding the multiplicative inverse. If we find that  $\gcd(x, n) = 1$ , then we want to find  $x^{-1}$ . By the above identity, this means:

$$\begin{aligned} ax + bn &= 1 \\ ax + bn &\equiv 1 \pmod{n} && \text{taking mod } n \text{ of both sides} \\ ax &\equiv 1 \pmod{n} && \text{doesn't change the truth} \end{aligned}$$

$bn \bmod n = 0$

Thus, finding the coefficient  $a$  will find us  $x^{-1}$ .

## 8.2.3 Modular Exponentiation

We already know how exponentiation works; why do we need to talk about it? Well, because of time complexity... we need to be able to exponentiate *fast*. With big numbers repeated multiplication gets out of hand quickly.

Equivalence in modular arithmetic works just like equality in normal arithmetic. So if  $a \equiv b \pmod{N}$  and  $c \equiv d \pmod{N}$  then  $a + c \equiv a + d \equiv b + c \equiv b + d \pmod{N}$ .

This fact makes *fast* modular exponentiation possible. Rather than doing  $x^y \bmod N$  via  $x \cdot x \cdot \dots$  or even  $((x \cdot x) \bmod N) \cdot x \bmod N \dots$ , we leverage repeated squaring:

$$\begin{aligned} x^y \bmod N : \\ x \bmod N &= a_1 \\ x^2 &\equiv a_1^2 \pmod{N} = a_2 \\ x^4 &\equiv a_2^2 \pmod{N} = a_3 \\ &\dots \end{aligned}$$

Then, we can multiply the correct powers of two to get  $x^y$ , so if  $y = 69$ , you would use  $x^{69} \equiv x^{64} \cdot x^4 \cdot x^1 \pmod{N}$ .

## 8.3 Groups for Cryptography

First, we need to define some more generic group properties.

**Group Elements** The order of a finite **group element**, denoted  $o(g)$  for some  $g \in G$ , is the smallest integer  $n \geq 1$  fulfilling  $g^n = 1$  (the identity element).

For any group element  $g \in G$ , we can generate a subgroup of  $G$  easily:

$$\langle g \rangle = \{g^0, g^1, \dots, g^{o(g)-1}\}$$

Naturally, its order is the order  $o(g)$  of  $G$ . Since we established above that the order of a subgroup divides the order of the group, the same is true for group elements. In other words,  $\forall g \in G : |G| \bmod o(g) = 0$ .

**Generator** These are a crucial part of **asymmetric cryptography**. A group element  $g$  is a **generator** of  $G$  if  $\langle g \rangle = G$ . This means that doing the exponentiation described above just shuffles  $G$  around into a different ordering.

For example, 2 is a generator for  $\mathbb{Z}_{11}^*$ :

$i$	0	1	2	3	4	5	6	7	8	9
$2^i \equiv a \pmod{11}$	1	2	4	8	5	10	9	7	3	6

An element is a generator if and **only** if  $o(g) = |G|$ , and a group is called **cyclic** if it contains at least one generator.

### 8.3.1 Discrete Logarithm

If  $G = \langle g \rangle$  is cyclic, then for every  $a \in G$ , there is a **unique** exponent  $i \in \{0, \dots, |G| - 1\}$  such that  $g^i = a$ . We call  $i$  the **discrete logarithm** of  $a$  to base  $g$ , denoting it by:  $\text{dlog}_{G,g}(a)$ . As you'd expect, it inverts exponentiation in the group.

To continue with our example group  $G = \mathbb{Z}_{11}^*$ , we know 2 is a generator (see above), so the  $\text{dlog}_{G,2}(\cdot)$  of any group element is well-defined.

$a$	1	2	3	4	5	6	7	8	9	10
$\text{dlog}_{G,2}(a)$	0	1	8	2	4	9	7	3	6	5

**Algorithm** How do we compute the discrete logarithm? Here's a naïve algorithm: just try all of the exponentiations. This is a simple algorithm but is exponential. There are better algorithms out there, but are still around  $\mathcal{O}(\sqrt{|G|})$  at best. **There are no polynomial time algorithms for computing the discrete logarithm.** This isn't proven, but much like the **AES** conjecture (see (1.2)), it is the foundation of asymmetric security.

#### FUN FACT: The State of the Art

If the group is based on some prime  $p$ , so  $G = \mathbb{Z}_p^*$ , the most efficient known algorithm is the **general number field sieve** which (still) has exponential complexity of the form:

$$\mathcal{O}\left(e^{1.92(\ln p)^{1/3}(\ln \ln p)^{2/3}}\right) \quad (8.2)$$

If we have a prime-order group over an **elliptic curve**, the best-known algorithm is  $\mathcal{O}(\sqrt{p})$ , where  $p = |G|$ .

We need to scale our security based on the state of the art for the groups in question: a 1024-bit

prime  $p$  is just as secure on  $\mathbb{Z}_p^*$  as a 160-bit prime  $q$  on an [elliptic curve](#) group. Obviously, smaller is preferable because it means our exponentiation algorithms will be much faster.

### 8.3.2 Constructing Cyclic Groups

As we already mentioned, cyclic groups are important for cryptography. How do we build these groups and find generators within them efficiently.

#### Finding Generators

Thankfully, there are some simple cases that let us create such groups:

- If  $p$  is a prime number, then  $\mathbb{Z}_p^*$  is a cyclic group.
- If the *order* of any group  $G$  is prime, then  $G$  is cyclic.
- If the order of a group is prime, then *every* non-trivial element is a generator (that is, every  $g \in G \setminus \{1\}$  where 1 is the identity element).

However, if  $G = \mathbb{Z}_p^*$ , then its order is  $p - 1$  which isn't prime. Though it may be hard to find a generator in general, it's easy if the prime factorization of  $p - 1$  is known. A prime  $p$  is called a [safe prime](#) if  $p - 1 = 2q$ , where  $q$  is *also* a prime. Safe primes are useful because it means that equally-hard to factor  $pq$  into either  $p$  or  $q$ .<sup>1</sup> Here, though, they're useful because  $|\mathbb{Z}_p^*|$  factors into  $(2, q)$  for safe primes.

**Property 8.2.** *Given a safe prime  $p$ , the order of  $\mathbb{Z}_p^*$  can be factored into  $(2, q)$ , where  $q$  is a prime. Then, a group element  $g \in \mathbb{Z}_p^*$  is a generator if and **only** if  $g^2 \not\equiv 1$  and  $g^q \not\equiv 1$ .*

Now, there a useful fact that  $\mathbb{Z}_p^*$  will have  $q - 1$  generators, so a simple randomized algorithm that chooses  $g \xleftarrow{\$} G \setminus \{1\}$  until  $g$  is a generator (checked by finding  $g^2$  and  $g^q$ ) will fail with only  $1/2$  probability. This becomes negligible after enough runs and will take two tries on average, letting us find generators quickly.

We just found a way to find a generator  $g$  in the group  $\mathbb{Z}_p^*$ ; the end-goal is to work over  $\langle g \rangle$ . Thus, our difficulty has transferred over to choosing safe primes.

#### Generating Primes

Because primes are dense—for an  $n$ -bit number, we'll find a prime every  $n$  runs on average—we can just generate random bitstrings until one of them is prime. Once we have a prime, making sure it's a [safe prime](#) does not add much complexity because they are also dense.

Given this, how do we check for primality quickly? [Fermat's little theorem](#) gives us a way to check for *positive* primality: if a randomly-chosen number  $r$  is prime, the theorem holds. However, checking all  $r - 1$  values against the theorem is not ideal. Similarly, checking whether or not all values up to  $\sqrt{r}$  divide  $r$  is not ideal.

It will be faster to identify a number as being **composite** (non-prime), instead. Namely, if the theorem *doesn't* hold, we should be able to find any specific  $z$  for which  $z^{r-1} \not\equiv 1 \pmod{r}$ . These are called a [Fermat witnesses](#), and every composite number has at least one.

This “at least one” is the **trivial** Fermat witness: the one where  $\gcd(z, r) > 1$ . Most composite numbers have many **non-trivial** Fermat witnesses: the ones where  $\gcd(z, r) = 1$ .

<sup>1</sup> For example, factoring 4212253 (into  $2903 \cdot 1451$ ) is much harder than factoring 5806 (into  $2903 \cdot 3$ ) because both of the former primes need around 11 bits to represent them.



The composites without non-trivial Fermat witnesses called are called **Carmichael numbers** or “pseudoprimes.” Thankfully, they are relatively rare compared to normal composite numbers so we can ignore them for our primality test.

**Property 8.3.** *If a composite number  $r$  has at least one non-trivial Fermat witness, then at least half of the values in  $\mathbb{Z}_r$  are Fermat witnesses.*

The above property inspires a simple *randomized* algorithm for primality tests that identifies prime numbers to a particular degree of certainty:

1. Choose  $z$  randomly:  $z \xleftarrow{\$} \{1, 2, \dots, r-1\}$ .
2. Compute:  $z^{r-1} \stackrel{?}{\equiv} 1 \pmod{r}$ .
3. If it is, then say that  $r$  is prime. Otherwise,  $r$  is definitely composite.

Note that if  $r$  is prime, this will always confirm that. However, if  $r$  is composite (and not a Carmichael number), this algorithm is correct half of the time by the above property. To boost our chance of success and lower false positives (cases where  $r$  is composite and the algorithm says it’s prime) we choose  $z$  many times. With  $k$  runs, we have a  $1/2^k$  chance of a false positive.

**Property 8.4.** *Given a prime number  $p$ , the number 1 only has the trivial square roots  $\pm 1$  under its modulus. In other words, there is no other value  $z$  such that:  $z^2 \equiv 1 \pmod{p}$ .*

The above property lets us identify Carmichael numbers during the fast exponentiation for  $3/4^{\text{th}}$  of the choices of  $z$ , which we can use in the same way as before to check primality to a particular degree of certainty.

## 8.4 Modular Square Roots

Finding the square roots under a modulus is considered to be just as “hard” as factoring. We say that  $a$  is a **square** (or **quadratic residue**) modulo  $p$  if there’s a  $b$  such that  $b^2 \equiv a \pmod{p}$ . We could also say that  $b$  is the **square root** of  $a$  under modulo  $p$ , though we don’t really use the notation  $\sqrt{a} \equiv b \pmod{p}$ .

Under modular arithmetic, square roots *don’t always exist*. They’re specifically defined when there are two (distinct) roots under a prime. For example, 25 has two square roots under mod 11. Since  $25 \bmod 11 \equiv 3$ , we’re looking for values that are also  $\equiv 3$  under modulo 11:

$$\begin{aligned} 5^2 &= 25 \equiv 3 & (\bmod 11) \\ 6^2 &= 36 \equiv 3 & (\bmod 11) \end{aligned}$$

Thus, the square root of 25 is **both** 5 and 6 under modulo 11 (weird, right?). Weird, right? Well, not so much when you consider that  $-5 \equiv 6 \pmod{11}$ .

Again, not every value has a square root: for example, 28 doesn’t under mod 11 (note that  $28 \bmod 11 = 6$ ). We can verify this by trying all possible values<sup>2</sup> under the modulus:

$$\begin{aligned} 1^2 &= 1 & (\bmod 11) \\ 2^2 &= 4 & (\bmod 11) \end{aligned}$$

<sup>2</sup> Also, notice that there are no other values that are equivalent to  $25 \equiv 3 \pmod{11}$ , confirming that there are only two roots under this modulus.

$$\begin{aligned}
3^2 &= 9 & (\text{mod } 11) \\
4^2 &= 16 \equiv 5 & (\text{mod } 11) \\
5^2 &= 25 \equiv 3 & (\text{mod } 11) \\
6^2 &= 36 \equiv 3 & (\text{mod } 11) \\
7^2 &= 49 \equiv 5 & (\text{mod } 11) \\
8^2 &= 64 \equiv 9 & (\text{mod } 11) \\
9^2 &= 81 \equiv 4 & (\text{mod } 11) \\
10^2 &= 100 \equiv 1 & (\text{mod } 11)
\end{aligned}$$

However, it does under mod 19:  $28 \bmod 19 = 9$ , and  $3^2 \bmod 19 = 9$ .

### 8.4.1 Square Groups

The **Legendre symbol** (also called the **Jacobi symbol**) is a compact way of indicating whether or not a value is a square:

$$J_p(a) = \begin{cases} 1 & \text{if } a \text{ is a square mod } p, \\ 0 & \text{if } a \bmod p = 0, \\ -1 & \text{otherwise} \end{cases} \quad (8.3)$$

With that, we can define sets of squares (or quadratic residues) in a group as:

$$\begin{aligned}
\text{QR}(\mathbb{Z}_p^*) &= \{a \in \mathbb{Z}_p^* : J_p(a) = 1\} \\
&= \{a \in \mathbb{Z}_p^* : \exists b \text{ such that } b^2 \equiv a \pmod{p}\}
\end{aligned} \quad (8.4)$$

For example, for  $\mathbb{Z}_{11}^*$ , we have the following:

$x$	1	2	3	4	5	6	7	8	9	10
$x^2 \pmod{11}$	1	4	9	5	3	3	5	9	4	1
$J_{11}(x)$	1	-1	1	1	1	-1	-1	-1	1	-1

Thus,  $\text{QR}(\mathbb{Z}_{11}^*) = \{1, 3, 4, 5, 9\}$ . There are exactly five squares and five non-squares, and each of the squares has exactly two square roots (this isn't a coincidence). Note that generators are never squares.

Recall that 2 is a generator of  $\mathbb{Z}_{11}^*$ . Let's map the discrete log table with the Jacobi table, now:

$x$	1	2	3	4	5	6	7	8	9	10
$\text{dlog}_{\mathbb{Z}_{11}^*, 2}(x)$	0	1	8	2	4	9	7	3	6	5
$J_{11}(x)$	1	-1	1	1	1	-1	-1	-1	1	-1

Notice that  $x$  is a square if  $\text{dlog}_{\mathbb{Z}_{11}^*, 2}(x)$  is even. This makes sense, since for any generator  $g$ , if it's raised to an even power (i.e. some  $2j$ ), then it's obviously a square:  $g^{2j} = (g^j)^2$ . It generalizes well:

**Property 8.5.** *If  $p \geq 3$  is a prime and  $g$  is a generator of  $\mathbb{Z}_p^*$ , then the set of quadratic residues (squares) is  $g$  raised to all of the even powers of  $p - 2$ :*

$$\text{QR}(\mathbb{Z}_p^*) = \{g^i : 0 \leq i \leq p - 2 \text{ and } i \bmod 2 = 0\}$$

Now previously, we defined the **Legendre symbol** as a simple indicator function (8.3); conveniently,

it can actually be computed for any prime  $p \geq 3$ :

$$J_p(a) \equiv a^{\frac{p-1}{2}} \pmod{p} \quad (8.5)$$

This is a cubic-time algorithm (in  $|p|$ ) to determine whether or not a number is a square. From this, we have another useful property.

**Property 8.6.** *If  $p \geq 3$  is a prime and  $g$  is a generator of  $\mathbb{Z}_p^*$ , then  $J_p(g^{xy} \bmod p) = 1$  if and only if either  $J_p(g^x \bmod p) = 1$  or  $J_p(g^y \bmod p) = 1$  (that is, at least one of them is a square), for all  $x, y \in \mathbb{Z}_{p-1}$ .*

*The corollary from this is that  $|\text{QR}(\mathbb{Z}_p^*)| = \frac{p-1}{2}$ .*

The Legendre symbol has the property **multiplicity**:  $J_p(ab) = J_p(a) \cdot J_p(b)$  for any  $a, b \in \mathbb{Z}$ . It also has an **inversion** property: the Legendre symbol of a value's inverse is the same as the value's. That is,  $L_p(a^{-1}) = L_p(a)$ . Both of these apply only for non-trivial primes:  $p \geq 3$ .

The following are “bonus” sections not directly related to the lecture content.

### 8.4.2 Square Root Extraction

A secondary, key fact of square-root extraction is the following: if square roots exist under a modulus, there are two for *each* prime in the modulus. For example, we found that 5 and 6 are the roots of 25 under modulus 11, but what about under 13? We know  $25 \bmod 13 = 12$ , so let's search:

$$\begin{array}{ll} 1^2 = 1 & (\bmod 13) \\ 2^2 = 4 & (\bmod 13) \\ 3^2 = 9 & (\bmod 13) \\ 4^2 = 16 \equiv 3 & (\bmod 13) \\ 5^2 = 25 \equiv \boxed{12} & (\bmod 13) \\ 6^2 = 36 \equiv 10 & (\bmod 13) \\ 7^2 = 49 \equiv 0 & (\bmod 13) \\ 8^2 = 64 \equiv \boxed{12} & (\bmod 13) \\ 9^2 = 81 \equiv 3 & (\bmod 13) \\ 10^2 = 100 \equiv 9 & (\bmod 13) \\ 11^2 = 121 \equiv 4 & (\bmod 13) \\ 12^2 = 144 \equiv 1 & (\bmod 13) \end{array}$$

Looks like 5 and 8 are the roots of 25 under mod 13. Thus, if we look for the roots under the *product* of  $11 \cdot 13 = 143$ , we will find *exactly* four values:<sup>3</sup>

$$\begin{array}{ll} 5^2 = 25 \equiv 25 & (\bmod 143) \\ 60^2 = 3600 \equiv 25 & (\bmod 143) \\ 83^2 = 6889 \equiv 25 & (\bmod 143) \end{array}$$

<sup>3</sup> These were found with a simple Python generator:

```
filter(lambda i: (i**2) % P == v % P, range(P))
```

$$138^2 = 19004 \equiv 25 \pmod{143}$$

The key comes from the following fact: by knowing the roots under both 11 and 13 separately, it's really easy to find them under  $11 \cdot 13$  *without* iterating over the entire space. To reiterate, our roots are 5, 6 (mod 11) and 5, 8 (mod 13). We can use the [Chinese remainder theorem](#) to find the roots quickly under  $13 \cdot 11$ .

**Finding Roots Efficiently** In our case, we have the four roots under the respective moduli, and we can use the CRT to find the four roots under the product. Namely, we find  $r_i$  for each pair of roots:

$$\begin{array}{ll} r_1 \equiv 5 \pmod{11} & r_2 \equiv 5 \pmod{11} \\ r_1 \equiv 5 \pmod{13} & r_2 \equiv 8 \pmod{13} \\ \\ r_3 \equiv 6 \pmod{11} & r_4 \equiv 6 \pmod{11} \\ r_3 \equiv 5 \pmod{13} & r_4 \equiv 8 \pmod{13} \end{array}$$

Finding each  $r_i$  can be done very quickly using the [extended Euclidean algorithm](#) in  $\mathcal{O}((|n| + |m|)^2)$  time (where  $|x|$  represents the bit count of each prime), which is much faster than the exhaustive search  $\mathcal{O}(2^{|m|+|n|})$  necessary without knowledge of 11 and 13. In this case, the four roots are 5, 60, 83, and 138 (in order of the  $r_i$ s above).

**Square root extraction of a product of primes  $pq$  is considered to be as difficult as factoring it.**

## 8.5 Chinese Remainder Theorem

The [Chinese remainder theorem](#) states that if you have a series of coprime values:  $n_1, n_2, \dots, n_k$ , then there exists a single value  $x \pmod{(n_1 n_2 \cdots n_k)}$  (the product of the individual moduli) that is equivalent to a series of values under each of them:

$$\begin{array}{l} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \dots \\ x \equiv a_k \pmod{n_k} \end{array}$$

# ENCRYPTION

When we studied symmetric encryption schemes, we relied on [block ciphers](#) as a fundamental building block for a secure [mode of operation](#). With asymmetric schemes, we no longer have that luxury since we have no shared [symmetric keys](#). However, we still need computationally-difficult problems (like the [PRF](#) conjecture of [AES](#)) to base our security on.

The problems we'll use are the discrete logarithm problem (whose difficulty we alluded to in [this aside](#)) as well as the [RSA](#) problem, later.

## 9.1 Recall: The Discrete Logarithm

Let  $G$  be a cyclic group,  $m = |G|$  be the order of the group, and  $g$  be a generator of  $G$ . Then discrete logarithm function  $\text{dlog}_{G,g}(a) : G \mapsto \mathbb{Z}_m$  takes a group element  $a \in G$  and returns the integer  $i \in \mathbb{Z}_m$  such that  $g^i = a$ .

There are several computationally-difficult problems associated with this function, each of which we'll examine in turn:

- The straightforward discrete log problem, in which you must find  $x$  given  $g^x$ .
- The [computational Diffie-Hellman](#) problem, in which you're given  $g^x$  and  $g^y$  and must find  $g^{xy}$ .
- The [decisional Diffie-Hellman](#) problem, in which you're given  $g^x$ ,  $g^y$ , and  $g^z$  and must figure out whether or not  $z \stackrel{?}{\equiv} xy \pmod{m}$ .

$$\boxed{\text{can solve DL}} \implies \boxed{\text{can solve CDH}} \implies \boxed{\text{can solve DDH}}$$

Though these problems all appear different, they boil down to the same fact: if you can solve the initial discrete log problem, you can solve all of them:

### 9.1.1 Formalization

In each case, suppose again that we're given a cyclic group  $G$ , the order of the group  $m = |G|$ , and a generator  $g$ . The adversary knows all of this (fixed) information.

**DL Problem** The discrete logarithm problem is described by an adversary  $\mathcal{A}$ 's ability to efficiently determine an original, randomly-chosen exponent:

$$\text{Exp}_{G,g}^{\text{dl}}(\mathcal{A}) : \begin{array}{l} x \xleftarrow{\$} \mathbb{Z}_m \\ x' = \mathcal{A}(g^x) \end{array}$$

if  $g^{x'} = g^x$ ,  $\mathcal{A}$  wins

As usual, we define the discrete problem as being “hard” if any adversary’s **dl-advantage** is **negligible** with polynomial resources.

### DEFINITION 9.1: Discrete-Log Advanatage

We can define the **dl-advantage** of an adversary as the probability of winning the discrete logarithm experiment:

$$\text{Adv}_{G,g}^{\text{dl}}(\mathcal{A}) = \Pr \left[ \text{Exp}_{G,g}^{\text{dl}}(\mathcal{A}) \text{ wins} \right]$$

**CDH Problem** The computational Diffie-Hellman problem is described by an adversary  $\mathcal{A}$ ’s ability to efficiently determine the product of two randomly-chosen exponents:

$$\begin{aligned} \text{Exp}_{G,g}^{\text{cdh}}(\mathcal{A}) : \quad & x, y \xleftarrow{\$} \mathbb{Z}_m \\ & z = \mathcal{A}(g^x, g^y) \\ & \text{if } z = g^{xy}, \mathcal{A} \text{ wins} \end{aligned}$$

The **cdh-advantage** and difficulty of CDH is defined in the same way as DL.

**DDH Problem** The decisional Diffie-Hellman problem is described by an adversary  $\mathcal{A}$ ’s ability to differentiate between two experiments (much like with the oracle of **IND-CPA** security and the others we saw with symmetric security definitions):

$$\begin{array}{ll} \text{Exp}_{G,g}^{\text{ddh-1}}(\mathcal{A}) : & \text{Exp}_{G,g}^{\text{ddh-0}}(\mathcal{A}) : \\ \quad x, y \xleftarrow{\$} \mathbb{Z}_m & \quad x, y \xleftarrow{\$} \mathbb{Z}_m \\ \quad z = xy \bmod m & \quad z \xleftarrow{\$} \mathbb{Z}_m \quad \text{key} \\ \quad d = \mathcal{A}(g^x, g^y, g^z) & \quad d = \mathcal{A}(g^x, g^y, g^z) \quad \text{difference} \\ \text{return } d & \text{return } d \end{array}$$

The difficulty of DDH is defined in the usual way based on the **ddh-advantage** of any adversary with polynomial resources.

### DEFINITION 9.2: DDH Advantage

The **ddh-advantage** of an adversary  $\mathcal{A}$  is its ability to differentiate between the true and random experiments:

$$\text{Adv}_{G,g}^{\text{ddh}}(\mathcal{A}) = \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-1}}(\mathcal{A}) \rightarrow 1 \right] - \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-0}}(\mathcal{A}) \rightarrow 1 \right]$$

#### 9.1.2 Difficulty

Under the group of a prime  $\mathbb{Z}_p^*$ , DDH is solvable in polynomial time, while the others are considered hard: the best-known algorithm is the **general number field sieve** whose complexity we mentioned in (8.2).

In contrast, under the [elliptic curves](#), all three of the aforementioned problems are harder than their  $\mathbb{Z}_p^*$  counterparts, with the best-known algorithms taking  $\sqrt{p}$  time, where  $p$  is the prime order of the group.

### DL Difficulty

Note that there is a linear time algorithm for breaking the DL problem, but it relies on knowing something that is hard to acquire. The algorithm relies on knowing the [prime factorization](#) of the *order* of the group. Namely, if we know the breakdown such that

$$p - 1 = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \dots \cdot p_n^{\alpha_n}$$

(where each  $p_i$  is a prime), then the discrete log problem can be solved in

$$\sum_{i=1}^n \alpha_i \cdot (\sqrt{p_i} + |p|)$$

time. Thus, if we want the DL problem to stay difficult, then at least one prime factor needs to be large (e.g. a [safe prime](#)) so the factorization is difficult.

### Breaking DDH is Easy

Let's take a look at the algorithm for breaking the decisional DH problem under the prime group  $\mathbb{Z}_p^*$ . Remember, the goal of breaking DDH essential comes down to differentiating between  $g^{xy}$  and a random  $g^{z \neq xy}$ .

The key lies in a fact we covered when discussing [Groups](#): we can easily differentiate squares and non-squares in  $\mathbb{Z}_p^*$  (see Property 8.5). There's an efficient adversary who can have a [ddh-advantage](#) of  $1/2$ : the idea is to compute the [Legendre symbols](#) of the inputs. Recall [Equation 8.5](#) or more specifically Property 8.6: the [Legendre symbol](#) of an exponent product must match the individual exponents.

---

**ALGORITHM 9.1:** An adversarial algorithm for DDH in polynomial time.

**Input:**  $(X, Y, Z)$ , the alleged Diffie-Hellman tuple where  $X = g^x$ ,  $Y = g^y$ , and  $Z$  is either  $g^{xy}$  or a random  $g^z$ .

**Result:** 1 if  $Z = g^{xy}$  and 0 otherwise.

**if**  $J_p(X) = 1$  *or*  $J_p(Y) = 1$  **then**  
     |  $s = 1$

**else**

    |  $s = -1$

**end**

**return** 1 *if*  $J_p(Z) = s$  *else* 0

---

Since  $g^x$  or  $g^y$  will be squares half of the time (by Property 8.5—even powers of  $g$  are squares), and  $g^{xy}$  can *only* be a square if this is the case, this check succeeds with  $1/2$  probability, since:

$$\begin{aligned} \text{Exp}_{G,g}^{\text{ddh-0}} \mathcal{A} &= 1 \\ \text{Exp}_{G,g}^{\text{ddh-1}} \mathcal{A} &= 1/2 \\ \therefore \text{Adv}_{G,g}^{\text{ddh}}(\mathcal{A}) &= 1 - 1/2 = 1/2 \end{aligned}$$

The algorithm only needs two modular exponentiations, meaning it takes  $\mathcal{O}(|p|^3)$  time (refer to Table 8.1) at worst and is efficient. ■

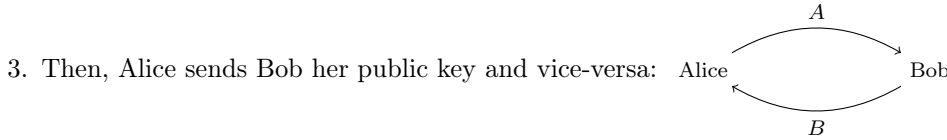
**Making DDH Safe** Since the best-known efficient algorithm relies on squares, we can modify the group in question to avoid the algorithm. Specifically, DDH is believed to be difficult (i.e. a minimal **ddh-advantage** for any polynomial adversary) in  $\text{QR}(\mathbb{Z}_p^*)$  where  $p = 2q + 1$ , a **safe prime**.

## 9.2 Diffie-Hellman Key Exchange

This algorithm, designed for two parties to derive a mutual secret without exposing it, relies on the difficulty of the **computational Diffie-Hellman** problem for its security.

The **Diffie-Hellman key exchange** is defined as follows: we first let  $G = \langle g \rangle$  be a cyclic group of order  $m$ , where both  $g$  and  $m$  are public parameters known to everyone. Alice and Bob derive a mutual secret key by communicating as follows:

1. Alice and Bob choose their **private keys** randomly which are simply random exponents:  $a \xleftarrow{\$} \mathbb{Z}_m$  and  $b \xleftarrow{\$} \mathbb{Z}_m$ , respectively.
2. From these values, they can derive their respective **public keys** by raising the generator to the relevant power:  $A = g^a$  and  $B = g^b$ .



4. Alice derives the mutual secret from Bob's public key:

$$s = B^a = (g^b)^a = g^{ab}$$

5. Bob similarly derives the same key:  $s = A^b = (g^a)^b = g^{ab}$ .

Because it's computationally difficult for an attacker to find either  $a$  or  $b$  (since that would involve finding the discrete log:  $a = \text{dlog}_{G,g}(A)$ ), this shared secret is derived without revealing any insecure information. The attacker is tasked with finding  $g^{ab}$  while only knowing  $g^a$  and  $g^b$ —exactly the **computational Diffie-Hellman** problem.

## 9.3 ElGamal Encryption

With **ElGamal** encryption, which extends Diffie-Hellman to do encryption, we compute a ciphertext with a temporary key exchange. The parameters stay the same, with  $G = \langle g \rangle$  having order  $m$ . This time, Alice is a recipient of a message from Bob, and the message space is the group itself (so  $M \in G$ ):

1. Alice chooses a random exponent as a secret just like before:  $a \xleftarrow{\$} \mathbb{Z}_m$ .
2. Alice again broadcasts her public key the same way:  $A = g^a$ .
3. Bob wants to send Alice a message,  $M$ , encrypting it as follows:
  - (a) He chooses a private exponent as before:  $b \xleftarrow{\$} \mathbb{Z}_m$ .



- (b) He encrypts  $M$  under the to-be-shared secret  $A^b = g^{ab}$  via the group operation:<sup>1</sup>  $c = M \cdot g^{ab}$  and sends  $(g^b, c)$  to Alice.
4. Alice can obviously derive the same secret  $s = g^{ab}$  and can find  $m$  by finding the inverse of the secret:  $M = c \cdot s^{-1} = (M \cdot s) \cdot s^{-1} = M$ .

This scheme relies on the property that every shared secret  $s = g^{ab}$  has an inverse  $s^{-1}$  which is guaranteed by the invertibility property of groups.

### 9.3.1 Security: IND-CPA

Since this is an encryption scheme, knowing that the secret cannot be found efficiently because of CDH is insufficient. We want to know if ElGamal is **IND-CPA** secure under specific groups.

Unfortunately, it's **not secure** under  $\mathbb{Z}_p^*$  for the same reasons as before: ElGamal boils down to the *decisional* Diffie-Hellman problem which we broke earlier with [algorithm 9.1](#). By the very definition of **IND-CPA**, an attacker should be unable to infer anything about the plaintext from the ciphertext, but under  $\mathbb{Z}_p^*$ , they can easily determine whether or not it's a square via the [Legendre symbol](#). Specifically, when given a ciphertext  $(B, c)$  for a message  $M$ , an adversary can compute  $J_p(M)$ ; the full attack is formalized in [algorithm 9.2](#).

---

**ALGORITHM 9.2:** An efficient adversary breaking ElGamal encryption under  $\mathbb{Z}_p^*$ .

**Input:** A public key,  $pk$ .

**Input:** The ElGamal parameters:  $G = \mathbb{Z}_p^* = \langle g \rangle, m$ .

**Result:** 1 if the left message is encrypted, 0 otherwise.

Let  $M_0$  be chosen such that  $J_p(M_0) = 1$  (for ex.  $M_0 = 1$ ).

Let  $M_1$  be chosen such that  $J_p(M_1) = -1$  (for ex.  $M_1 = g$ ).

$(X, c) = \mathcal{E}_P(LR(M_0, M_1))$

**if**  $J_p(X) = 1$  **or**  $J_p(pk) = 1$  **then**

$s = J_p(c)$

**else**

$s = -J_p(c)$

**end**

**return** 1 *if*  $s = 1$  *else* 0

---

There is good news, though: ElGamal encryption is **IND-CPA** secure for a group if the DDH problem on the same group is hard. As a reminder, such groups include prime-order subgroups of  $\mathbb{Z}_p^*$  or elliptic curve prime order groups.

**Claim 9.1.** *If the **decisional** Diffie-Hellman problem is computationally hard for a group  $G$ , ElGamal encryption is **IND-CPA** under the same group.*

*Proof.* We proceed by showing that if ElGamal is not **IND-CPA**, then the DDH problem is not hard (this is the contrapositive technique we used when proving the **IND-CCA** security of the CBC mode of operation; see [Theorem 1.2](#)).

Assume  $\mathcal{A}$  is an **IND-CPA** attacker for ElGamal. We will use  $\mathcal{A}$  to construct an DDH adversary  $\mathcal{B}$ .

---

<sup>1</sup> Recall that a group is defined by a set *and* a specific operation (see [section 8.1](#) for a review) which we have been denoting as  $\cdot$  (as opposed to  $\cdot$  for multiplication).

Simply put,  $\mathcal{B}$  will pass along the Diffie-Hellman tuple it receives as part of the DDH challenge and let  $\mathcal{A}$  differentiate between “real” and “random” tuples.

Recall that the [DDH Problem](#) is to differentiate a value  $g^z$  where  $z$  is chosen randomly from a  $g^z$  where  $z = xy \bmod m$  when given  $g^x$  and  $g^y$ .

Our adversary  $\mathcal{B}$  is given  $\left(g, X = g^x, Y = g^y, Z = g^{z \stackrel{?}{=} xy}\right)$  and will use  $\mathcal{A}$  as follows:

1. Flip a coin as an oracle, choosing  $b \xleftarrow{\$} \{0, 1\}$ .
2. Run  $\mathcal{A}$  with the ElGamal parameters  $g$  and [public key](#)  $X$ .
3. When  $\mathcal{A}$  makes a query (with, say,  $(m_0, m_1)$ ), return  $(Y, m_b \cdot Z)$ , exactly as specified by ElGamal encryption— $Y$  here is the one-time public key of the “sender” and  $m_b \cdot Z$  is the ciphertext.
4. Let  $d$  be  $\mathcal{A}$ ’s result—its guess of  $b$ .
5. If  $d = b$ , return 1 (this is a *real* DDH tuple,  $Z = g^{xy}$ ); otherwise, return 0 (this is a *random* DDH tuple).

The rationale is that because  $Z$  is an invalid ciphertext construction in the case of a random DDH tuple,  $\mathcal{A}$  *should* fail at breaking the scheme since its invalid. Let’s justify the [ddh-advantage](#). By definition,

$$\text{Adv}_{G,g}^{\text{ddh}}(\mathcal{B}) = \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-1}}(\mathcal{B}) \rightarrow 1 \right] - \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-0}}(\mathcal{B}) \rightarrow 1 \right]$$

Let’s break this down into its component parts. Notice that the first probability (differentiating real tuples correctly) is simply dependent on  $\mathcal{A}$ ’s ability to break ElGamal encryption under the [IND-CPA-cg](#) variant (in other words,  $\mathcal{B}$  acts exactly like an [IND-CPA-cg](#) oracle by choosing  $b$  randomly and comparing  $d \stackrel{?}{=} b$ ):

$$\begin{aligned} \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-1}}(\mathcal{B}) \rightarrow 1 \right] &= \Pr \left[ \text{Exp}_{\text{EG}}^{\text{ind-cpa-cg}}(\mathcal{A}) \rightarrow 1 \right] \\ &= \frac{1}{2} + \frac{1}{2} \text{Adv}_{\text{EG}}^{\text{ind-cpa}}(\mathcal{A}) \end{aligned} \quad \begin{array}{l} \text{from the proof of} \\ \text{Definition 3.3} \end{array}$$

On the other side, we have  $\mathcal{B}$ ’s chance of failing to differentiate random tuples. By construction,  $\mathcal{B}$  outputs 1 when  $\mathcal{A}$  is correct; because  $\mathcal{A}$  receives a random group element in this case (some random  $g^z$ , by definition of a generator), there’s no way  $\mathcal{A}$  can make any sense of it. Thus, it cannot do *better* than a random guess at  $b$ :

$$\Pr \left[ \text{Exp}_{G,g}^{\text{ddh-0}}(\mathcal{B}) \rightarrow 1 \right] \leq \frac{1}{2}$$

Combining these, we see:

$$\begin{aligned} \text{Adv}_{G,g}^{\text{ddh}}(\mathcal{B}) &= \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-1}}(\mathcal{B}) \rightarrow 1 \right] - \Pr \left[ \text{Exp}_{G,g}^{\text{ddh-0}}(\mathcal{B}) \rightarrow 1 \right] \\ &\geq \frac{1}{2} + \frac{1}{2} \text{Adv}_{\text{EG}}^{\text{ind-cpa}}(\mathcal{A}) - \frac{1}{2} \\ &\geq \frac{1}{2} \cdot \text{Adv}_{\text{EG}}^{\text{ind-cpa}}(\mathcal{A}) \end{aligned}$$

As desired, we see that if  $\mathcal{A}$  is successful, then  $\mathcal{B}$  is successful. Thus, by the contrapositive, if a group’s DDH problem is hard, the corresponding ElGamal scheme is [IND-CPA](#) secure.  $\blacksquare$

### 9.3.2 Security: IND-CCA

Regardless of the security of the underlying DDH problem, ElGamal is not **IND-CCA**. It's trivially breakable by passing a message inverse. For example, the ciphertext is  $(B, c) = \mathcal{E}_A(LR(m_0, m_1))$ . Then, what is  $\mathcal{D}_B(c \cdot m_1^{-1})$ ?

$$\begin{aligned}\mathcal{D}_B(c \cdot m_1^{-1}) &= c \cdot m_1^{-1} \cdot (g^{ab})^{-1} \\ &= m_b \cdot m_1^{-1} \cdot g^{ab} \cdot (g^{ab})^{-1} \\ &= m_b \cdot m_1^{-1}\end{aligned}$$

If  $b = 1$  (the right message was chosen), then this simply evaluates to the identity element 1! In the other case, it does not, so this is a sufficient check for always correctly differentiating which plaintext was encrypted.

## 9.4 Cramer-Shoup Encryption

With **ElGamal** failing to provide ideal security properties, can we even construct an asymmetric scheme that is **IND-CCA** secure? Yes: the **Cramer-Shoup** scheme invented in '98 provides this level of security.

It's far more involved, and defined formally as follows: let  $G$  be a cyclic group of order  $q$ , and  $g_1, g_2$  are random, distinct generators of  $G$ .

First, key generation is defined as follows:

$$\begin{aligned}\textbf{Key generation } \mathcal{K} : \quad & x_1, x_2, y_1, y_2, z \xleftarrow{\$} \mathbb{Z}_q \\ & c = g_1^{x_1} \cdot g_2^{x_2} \\ & d = g_1^{y_1} \cdot g_2^{y_2} \\ & h = g_1^z\end{aligned}$$

Notice that there are **five** secret exponents.

The encryption algorithm requires many more (expensive) exponentiations:

**Encryption**  $\mathcal{E}_{(G,q,g_1,g_2,c,d,h)}(M) :$

$$\begin{aligned} k &\xleftarrow{\$} \mathbb{Z}_q \\ u_1 &= g_1^k \\ u_2 &= g_2^k \\ e &= h^k \cdot M \\ \alpha &= H(u_1, u_2, e) \\ v &= c^k \cdot d^{k\alpha} \\ \text{return } &(u_1, u_2, e, v) \end{aligned}$$

Here,  $H$  is a [cryptographically-secure hash function](#). Finally, decryption uses the secret exponent as follows:

**Decryption**  $\mathcal{D}_{(x_1,x_2,y_1,y_2,z)}(u_1, u_2, e, v) :$

$$\begin{aligned} \alpha &= H(u_1, u_2, e, v) \\ \text{if } &(u_1^{x_1} \cdot u_2^{x_2} \cdot (u_1^{y_1} \cdot u_2^{y_2})^\alpha = v) \\ &\text{return } \frac{e}{u_1^z} \\ \text{return } &\perp \end{aligned}$$

**Property 9.1.** *If the [decisional Diffie-Hellman](#) problem for the group  $G$  is hard and  $H$  is a [cryptographically-secure hash function](#), then Cramer-Shoup is [IND-CCA](#) secure.*

Despite its strong security, this scheme is not used in practice because far more efficient [IND-CCA](#) secure asymmetric algorithms exist, though they do not rely on the difficulty of the discrete logarithm problem.

## 9.5 RSA Encryption

The [RSA](#) cryptosystem does not rely on the difficulty of the discrete logarithm. Instead, it relies on a different, but similar problem: **factorization**. Given a product of two prime numbers,  $N = pq$ , it's considered computationally difficult to find the original  $p$  and  $q$ .

With our basic understanding of group theory, the math behind RSA is very simple. Recall that  $x^a = x^{a \bmod m}$ , if  $m$  is the order of the group. Well when given  $N = pq$  and working in  $\mathbb{Z}_N^*$  we have order  $\varphi(N) = (p-1)(q-1)$  (this is Euler's [totient function](#) and it's proved below). Given a choice of  $e$  such that we can find  $de \equiv 1 \pmod{\varphi(N)}$  (that is  $d$  is the multiplicative inverse of  $e$ ), we have the following property:

$$\begin{aligned} m^{de} &\equiv m^{de \bmod \varphi(N)} \pmod{N} \\ &\equiv m^1 \equiv m \pmod{N} \end{aligned} \quad \text{recall (8.1)}$$

This means that raising a message  $m$  to the  $de$  power simply returns the original message. The RSA protocol thus works as follows. A user reveals their [public key](#) to the world: the exponent  $e$  and the modulus  $N$ . To send them a message,  $m$ , you send  $c = m^e \bmod N$ . They can find your message by raising it to their [private key](#) exponent,  $d$ :

$$= c^d \bmod N$$

$$\begin{aligned}
&= (m^e \bmod N)^d = m^{ed} \bmod N \\
&= m \bmod N
\end{aligned}$$

This is secure because you cannot determine  $(p-1)(q-1)$  from the revealed  $N$  and  $e$  without exhaustively enumerating all possibilities<sup>2</sup> (i.e. “factoring is hard”); thus, if  $p$  and  $q$  are large enough, it’s computationally infeasible for an adversary to factor  $N$ .

### QUICK MAFFS: Proof of Euler’s Totient Function

We claimed above that given  $N = pq$ , then  $\varphi(N) = (p-1)(q-1)$ .

The proof of this is straightforward. Which values can divide  $pq$ ? Obviously any multiple of  $p$ :  $1p, 2p$ , and so on up to  $(q-1)p$ , and likewise for  $q$ . We can calculate the total number of these directly:

$$\begin{aligned}
\varphi(N) = \varphi(pq) &= \left| \underbrace{\{1, 2, \dots, N-1\}}_{\text{the entire set, } \mathbb{Z}_N} \right| - \left| \underbrace{\{ip : 1 \leq i \leq q-1\}}_{\text{multiples of } p} \right| - \left| \underbrace{\{iq : 1 \leq i \leq p-1\}}_{\text{multiples of } q} \right| \\
&= (N-1) - (q-1) - (p-1) = N-1-q+1-p+1 \\
&= pq - q - p + 1 \\
&= (p-1)(q-1) \quad \blacksquare
\end{aligned}$$

#### 9.5.1 Protocol

With the math out of the way, here’s the full protocol. Note that  $pk = (e, N)$  is the **public key** information, and  $sk = (d, N)$  is the private key information (where  $d$  is the only real “secret,” but both values are needed).

$$\begin{aligned}
\mathcal{K} : ed &\equiv 1 \pmod{\varphi(N)} && (e, d \in \mathbb{Z}_{\varphi(N)}^*) \\
\mathcal{E}_{pk}(m) &= x^e \bmod N && (\mathbb{Z}_N^* \mapsto \mathbb{Z}_N^*) \\
\mathcal{D}_{sk}(c) &= c^d \bmod N
\end{aligned}$$

The RSA encryption function  $f$  (defined with  $\mathcal{E}$  above) is a **one-way** permutation with a **trap door**: without the special “trap door” value  $d$ , it’s hard to find  $f^{-1}$ .

**Receiver Setup** To be ready to receive a message:

1. Pick two  $n$ -bit random prime numbers,  $p$  and  $q$ .
2. Then, choose an  $e$  that is relatively prime to  $(p-1)(q-1)$  (that is, by ensuring that  $\gcd(e, (p-1)(q-1)) = 1$ ). This can be done quickly by enumerating the low primes and finding their GCD; an exponent of  $e = 3$  is fairly common for computational efficiency.
3. Let  $N = pq$  and publish the public key  $(N, e)$ .
4. Your private key is  $d \equiv e^{-1} \pmod{(p-1)(q-1)}$  which we know exists and can be found with the **extended Euclidean algorithm**.

<sup>2</sup> Notice that if an adversary knew both  $N$  and  $\varphi(N)$ , they could use this information to form a simple quadratic equation which can obviously be solved in polynomial time. If they knew  $\varphi(N)$  and  $e$ , then  $d$  is just the modular inverse of  $e$  under mod  $\varphi(N)$ .

**Sending** Given an intended recipient’s public key,  $(N, e)$ , and a message  $m \leq N$ , simply compute and send  $c = m^e \bmod N$ . This can be calculated quickly using fast exponentiation (refer to [subsection 8.2.3](#)).

**Receiving** Given a received ciphertext,  $c$ , to find the original message simply calculate  $c^d \bmod N = m$  (again, use fast exponentiation).

### 9.5.2 Limitations

For this to work, the message must be small:  $m \in \mathbb{Z}_N^*$ . This is why asymmetric cryptography is typically only used to exchange a secret **symmetric key** which is then used for all other future messages. There are also a number of attacks on plain RSA that must be kept in mind:

- We need to take care to choose  $ms$  such that  $\gcd(m, N) = 1$ . If this isn’t the case, the key identity  $m^{ed} \equiv m \pmod{N}$  still holds—albeit this time by the [Chinese remainder theorem](#) rather than [Euler’s theorem](#)—but now there’s a fatal flaw. If  $\gcd(m, N) \neq 1$ , then it’s either  $p$  or  $q$ . If it’s  $p$ , then  $\gcd(m^e, N) = p$  and now  $N$  can easily be factored (and likewise if it’s  $q$ ).
- Similarly,  $m$  can’t be too small, because then it’s possible to have  $m^e < N$ —the modulus has no effect and directly taking the  $e^{\text{th}}$  root will reveal the plaintext!
- Even though small  $e$  are acceptable, the private exponent  $d$  cannot be too small. Specifically, if  $d < \frac{1}{3} \cdot N^{1/4}$ , then given the public key  $(N, e)$  one can efficiently compute  $d$ .
- Another problem comes from sending the same *message* multiple times via different *public keys*. The [Chinese remainder theorem](#) can be used to recover the plaintext from the ciphertexts; this is known as **Hastad’s broadcast attack**. Letting  $e = 3$ , for example, the three ciphertexts are:

$$\begin{aligned} c_1 &\equiv m^3 \pmod{N_1} \\ c_2 &\equiv m^3 \pmod{N_2} \\ c_3 &\equiv m^3 \pmod{N_3} \end{aligned}$$

The CRT states that  $c_1 \equiv c_2 \equiv c_3 \pmod{N_1 N_2 N_3}$ , but these would just be  $m^3$  “unrolled” without the modulus, since  $m^3 < N_1 N_2 N_3$ ! Thus,  $m = \sqrt[3]{m^3}$ , and finding  $m^3$  can be done quickly with the [extended Euclidean algorithm](#).

#### RSA: Alternative Derivation

In much of the literature surrounding RSA, the fundamentals of group theory are not discussed prior, making the derivations of the math a little more confusing. Because the theorems used are still very important to know and recognize, I cover them here.

Rather than relying on our well-known property that  $x^a = x^{a \bmod m}$  for a group of order  $m$ , we will instead use [Fermat’s little theorem](#) and its older brother, [Euler’s theorem](#) to understand the math in RSA. Both of these theorems are simply special cases of the generic exponentiation fact we showed when covering [Groups](#) (again, see [Equation 8.1](#)).

**Theorem 9.1** (Fermat’s little theorem). *If  $p$  is any prime, then*

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

*for any number  $1 \leq a \leq p - 1$ .*

### Intuition via Fermat

Suppose we take two values  $d$  and  $e$  such that  $de \equiv 1 \pmod{p-1}$ . By the definition of modular arithmetic, this means that:  $de = 1 + k(p-1)$  (i.e. some multiple  $k$  of the modulus plus a remainder of 1 adds up to  $de$ ). Notice, then, that for some  $m$ :

$$\begin{aligned}
 m^{de} &= m \cdot m^{de-1} \\
 &= m \cdot m^{k(p-1)} \\
 &\equiv m \cdot (m^{p-1})^k \pmod{p} \\
 &\equiv m \cdot 1^k \pmod{p} && \text{by Fermat's little theorem} \\
 \therefore m^{de} &\equiv m \pmod{p}
 \end{aligned}$$

We're almost there; notice what we've derived: Take a message,  $m$ , and raise it to the power of  $e$  to "encrypt" it. Then, you can "decrypt" it and get back  $m$  by raising it to the power of  $d$ .

### Intuition via Euler

Unfortunately, for the above to work, we need to reveal  $p$ , which obviously reveals  $p-1$  and lets someone derive the "private" key  $d$ . We'll hide this by using  $N = pq$  and [Euler's theorem](#) which relates the [totient function](#) to multiplicative inverses.

**Theorem 9.2** (Euler's theorem). *For any  $N, a$  that are [relatively prime](#) (that is, where  $\gcd(a, N) = 1$ ), then:*

$$a^{\varphi(N)} \equiv 1 \pmod{N}$$

where  $\varphi(N)$  is Euler's [totient function](#).

If  $N = pq$ , then this theorem tells us that:

$$a^{(p-1)(q-1)} \equiv 1 \pmod{N}$$

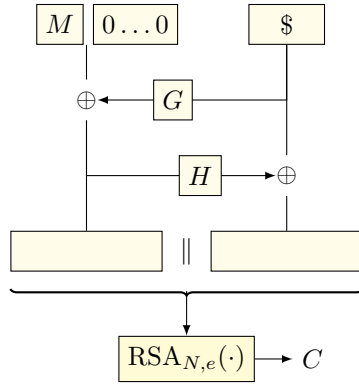
The rationale for "encryption" is the same as before: take  $d, e$  such that  $de \equiv 1 \pmod{\varphi(N)}$ . Then,

$$\begin{aligned}
 m^{de} &= m \cdot m^{de-1} \\
 &= m \cdot m^{(p-1)(q-1)k} && \text{def. of mod} \\
 &\equiv m \cdot (m^{(p-1)(q-1)})^k \pmod{N} \\
 &\equiv m \cdot 1^k \pmod{N} && \text{by Euler's theorem} \\
 \therefore m^{de} &\equiv m \pmod{N}
 \end{aligned}$$

## 9.5.3 Securing RSA

By its very nature, RSA is a deterministic protocol, so it cannot be [IND-CPA](#) secure (see [Theorem 1.1](#)). There is a protocol called [RSA-OAEP](#) that is *conjectured* to be [IND-CCA](#) secure.

It relies on two [hash functions](#),  $H$  and  $G$ , and random inputs to pad the message in a non-deterministic way to derive the final ciphertext. This scheme has been *proven* to be [IND-CCA](#) secure if RSA is secure (obviously) and if  $G$  and  $H$  are modeled as [random oracles](#).



**Algorithm:**

```

 $m = M \parallel \{0 \dots 0\}$ 
 $r \xleftarrow{\$} \{0, 1\}^k$ 
 $\text{left} = G(r) \oplus m$ 
 $\text{right} = H(\text{left}) \oplus r$ 
return  $\text{RSA}_{N,e}(\text{left} \parallel \text{right})$ 

```

**Figure 9.1:** A visualization of the RSA-OAEP encryption algorithm.

The **random oracle** model assumes that all parties involved must access an oracle that acts as a truly-random function. This does not match reality, since hashes can be computed locally without consulting an oracle; furthermore, the hash functions often imitate *pseudorandom functions*. However, it's still a useful construction and key to many security proofs.

## 9.6 Hybrid Encryption

As already mentioned, asymmetric encryption schemes are often used to exchange keys for symmetric schemes, because the latter class of algorithms is much more performant. Fortunately, the security of these combined schemes is guaranteed:

**Property 9.2.** *If the components of a hybrid encryption scheme are **IND-CPA**, then the associated hybrid encryption is overall also **IND-CPA**. This property also holds also for **IND-CCA**.*

## 9.7 Multi-User Encryption

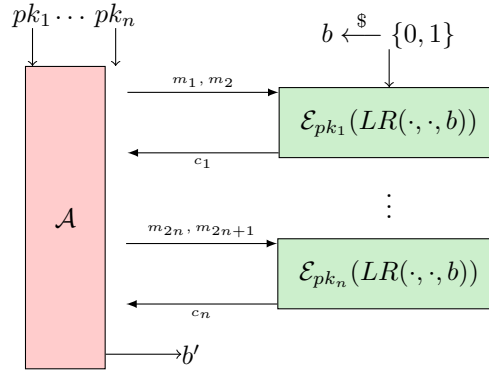
The setting of one user sending messages to another user is getting rarer by the day; more often than not, we have one-to-many relationships with our messages, from dozen-member group chats to thousand-member channels. To evaluate our schemes under these multi-user scenarios, we need to reconsider our security definitions.

### 9.7.1 Security Definitions

In both **IND-CPA** and **IND-CCA**, adversaries were trying to break communications between a single user and oracle. Recall **Hastad's broadcast attack** on RSA: sending the same message to many users is completely broken, despite the fact that we can't recover plaintexts under single-user RSA. Similarly, we know that **RSA-OAEP** is **IND-CCA** secure (with the **random oracle** assumption) with just one pair of users, but again, what about the multi-user setting? Thus, we need a new setting in which a user communicates with many distinct oracles.

The adversarial scenario for the ***n*-IND-CPA** experiment is visualized in **Figure 9.2** and should be relatively intuitive: instead of a single oracle and public key, there are  $n$  oracles and  $n$  public keys. All of them use the same  $b$  (that is, if they choose  $b = 0$ , they will always encrypt the left message), and the adversary's goal is to differentiate between left and right messages to output  $b'$ , their guess.





**Figure 9.2:** A visualization of the IND-CPA adversary in the multi-user setting, where  $\mathcal{A}$  knows  $n$  public keys and must output their guess,  $b'$ , corresponding to the left-right oracles' collective choice of  $b$ .

### DEFINITION 9.3: $n$ -IND-CPA Advantage

An asymmetric encryption scheme  $\mathcal{AE}$  is considered secure under  $n$ -IND-CPA if its  **$n$ -IND-CPA advantage** is negligible, where the advantage is defined similarly to the IND-CPA case:

$$\text{Adv}_{\mathcal{AE}}^{\text{n-ind-cpa}}(\mathcal{A}) = \Pr \left[ \text{Exp}_{\mathcal{AE}}^{\text{n-ind-cpa-0}}(\mathcal{A}) \rightarrow 0 \right] - \Pr \left[ \text{Exp}_{\mathcal{AE}}^{\text{n-ind-cpa-1}}(\mathcal{A}) \rightarrow 0 \right]$$

We can define multi-party,  **$n$ -IND-CCA** security in a similar way: we now have  $n$  decryption oracles, and the attacker is (only) restricted from decrypting ciphertexts from the *corresponding* encryption oracle.

## 9.7.2 Security Evaluation

The good news is that strong security in the single-user setting implies strong security in the multi-user setting: there is an upper bound on the advantage gained in the multi-user setting.

**Theorem 9.3.** *For an asymmetric encryption scheme  $\mathcal{AE}$ , for any adversary  $\mathcal{A}$  there exists a similarly-efficient adversary  $\mathcal{B}$  that only uses one query to the left-right oracle such that:*

$$\text{Adv}_{\mathcal{AE}}^{\text{n-ind-cpa}}(\mathcal{A}) \leq n \cdot q_e \cdot \text{Adv}_{\mathcal{AE}}^{\text{ind-cpa}}(\mathcal{B})$$

*where  $n$  is the number of users and  $q_e$  is the number of queries made to the encryption oracles.*

An identical definition exists relating  $n$ -IND-CCA to IND-CCA security.

The fact that we have asymptotically similar security is good, but these factors ( $n$  and  $q_e$ ) are significant: **security degrades with more users and more messages**. For example, allowing 200 million users to intercommunicate and encrypt  $2^{30}$  messages under each key (these are large numbers, but far from unreasonable to a dedicated attacker), the  $n$ -IND-CPA advantage is only 0.2 (pretty high!) if the original IND-CPA advantage was  $2^{-60}$  (very low!).

Thankfully, this definition is a *universal* upper bound: we can do better with *specific* schemes that have multiple users in mind. For example, **ElGamal** encryption makes better guarantees:

**Theorem 9.4** (Multi-User ElGamal). *For any adversary  $\mathcal{A}$  under multi-user ElGamal encryption, there exists a similarly-efficient adversary  $\mathcal{B}$  that only makes one query such that:*

$$\text{Adv}_{EG}^{n\text{-ind-cpa}}(\mathcal{A}) \leq \text{Adv}_{EG}^{\text{ind-cpa}}(\mathcal{B})$$

This is obviously much stronger than the generic guarantees of the previous theorem.

Unfortunately, no such improvements exist for RSA. For [Cramer-Shoup](#) encryption, a better bound exists but only drops one of the linear terms.

## 9.8 Scheme Variants

There are some alternative approaches to asymmetric cryptography that don't rely on our so-called "hard math problems" to work:

**identity-based encryption** Recall the linchpin of public key encryption: the sender needs to be able to find the receiver's public key somewhere, typically looking it up from some central, trusted authority. With IBE, senders don't need public keys to encrypt; instead, they can use an arbitrary string owned by the receiver (such as an email address) and provide it to a central authority to get a secret key.

Nothing is free, of course, and this last point is its fundamental flaw: the central authority must be *extremely* trustworthy and safe in order to store secret keys (rather than public keys, which are much more innocuous).

**attribute-based encryption** In this variant, the secret key and ciphertext of a message are associated with attributes about the recipient, such as their age, title, etc. Decryption is only possible if the attributes of the recipient's key match that of the ciphertext.

Of course, this implies a bit of a chicken-and-egg problem much like in [key distribution](#): how does the sender learn the supposedly-hidden attributes of the recipient without anyone else knowing them, and doesn't that mean they can decrypt any ciphertexts now intended for that recipient since they know that secret information?

**homomorphic encryption** This is an active area of research, especially among folks aiming to do machine learning on encrypted data. A homomorphic encryption scheme allows mathematical operations to be done on encrypted data securely while also guaranteeing that a corresponding operation is done on the *underlying* plaintext. More specifically, the encryption of an input  $x$  can be turned into the encryption of an input  $f(x)$ .

Interestingly-enough, RSA is homomorphic under multiplication. Consider two messages encrypted under the same public key:

$$c_1 = m_1^e \bmod N \quad c_2 = m_2^e \bmod N$$

Calculating their product also calculates the product of their plaintexts:

$$c_1 \cdot c_2 \equiv m_1^e \cdot m_2^e \equiv (m_1 \cdot m_2)^e \pmod{N}$$

Recently, there has been a fully-homomorphic encryption scheme designed to work with arbitrary  $f$ s, but it is still impractical for general, applied usage.

**searchable encryption** In this variant, clients can outsource their encrypted data to remote, untrusted servers to perform search queries efficiently without revealing the underlying data.

# DIGITAL SIGNATURES

We need to discuss providing **authenticity** and **integrity** in the world of asymmetric schemes just like we did in the the **symmetric key** setting. As before, we'll ignore **confidentiality** and focus on these goals instead. In the symmetric world, we used **message authentication codes** and **hash functions**; now, we'll use **signatures**.

**Notation** Much like before, digital signatures schemes are described by a tuple describing the way to generate keys, sign messages, and verify them:

$$\mathcal{DS} = (\mathcal{K}, \text{SIGN}, \text{VF})$$

The **sender** runs the signing algorithm using the their private key, and the **receiver** runs the verifying algorithm using the sender's public key.

*Correctness* — For every message in the message space, and every key-pair that can be generated, a signature can be correctly verified if (and **only** if) it was output by the signing function. Formally,

$$\begin{aligned} \forall m \in \text{MsgSp} \text{ and } \forall (pk, sk) \xleftarrow{\$} \mathcal{K} : \\ s = \text{SIGN}(m, sk) \iff \text{VF}(m, s, pk) = 1 \end{aligned}$$

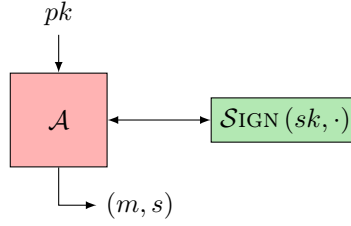
The signing algorithm can be randomized or stateful, but doesn't have to be for security. The message space is typically all bit-strings:  $\text{MsgSp} = \{0, 1\}^*$ .

**Comparison** Interestingly, signatures offer a security advantage over MACs in one specific case: non-repudiation. In the MAC case, the sender and receiver need to share a secret key, meaning the receiver can now impersonate the sender. In the signature case, however, there is no secret to share, so there is no impersonation risk.

## 10.1 Security Definitions

The idea behind security for digital signatures is similar to that of **message authentication codes**, as we've already noted. There, we had the **UF-CMA** definition: an adversary shouldn't be able to create a MAC that verifies a message unless it was received from the oracle.

For signatures, we want to ensure the same principle: an adversary should not be able to craft a valid signature for a message (that is, one verifiable by the oracle's public key) without the oracle signing it with its secret key.



The only difference is that we no longer need a verification oracle, since anyone with the signature, message, and knowledge of the public key should be able to validate the sender. Above,  $(m, s)$  is a message-signature pair such that  $m$  was never queried to the signature oracle. The **UF-CMA advantage** is defined in the same way.

## 10.2 RSA

Fascinatingly, the math behind **RSA** allows it to be used nearly as-is for creating digital signatures.

### 10.2.1 Plain RSA

Initially, all we do is apply the definition of encryption and decryption to correspond to verifying and signing, respectively. In other words,

**Algorithm**  $\text{SIGN}(M, (N, d)) :$

```

if  $M \notin \mathbb{Z}_N^*$ 
  return  $\perp$ 
return  $M^d \bmod N$ 

```

**Algorithm**  $\mathcal{V}_F(M, s, (N, e)) :$

```

if  $M, s \notin \mathbb{Z}_N^* \vee M \neq s^e \bmod N$ ,
  return 0
return 1

```

Though this simple scheme appears effective, it is **not secure** under **UF-CMA** by simple properties of modular arithmetic:

- Consider  $M = 1$ : the signature is  $1^d \bmod N = 1$ ; thus, we can always return  $(1, 1)$  as an adversary without querying the signing oracle whatsoever.
- Consider an arbitrary signature  $s$ . By the RSA property that  $m^{de} = m$ , we can recover the original message since we know that  $s^e \bmod N = m$ . Thus, we can choose any signature  $s \in \mathbb{Z}_N^*$ , calculate its source message, and output the pair as a forgery—no oracle needed.
- Finally, consider the multiplicative property of RSA. It's useful for **homomorphic encryption**, but is the kryptonite of signatures. Given two queried signatures  $s_1, s_2$  for two distinct messages  $m_1, m_2$ , we can derive a novel message-signature pair  $(m_1 m_2 \bmod N, s_1 s_2 \bmod N)$  since:

$$s_1 \cdot s_2 \equiv m_1^d \cdot m_2^d \equiv (m_1 \cdot m_2)^d \pmod{N}$$

All of these could be formalized into proper UF-CMA adversaries, but they are omitted here for brevity; they should be very straightforward.

### 10.2.2 Full-Domain Hash RSA

What are the main pain points of plain RSA? Well, the flaws above were based in modular arithmetic itself. To fix them, improving the previous simple scheme and ensuring **UF-CMA** security, we will *hash* the messages before they're signed to eliminate these relationships.

Let  $H : \{0, 1\}^* \mapsto \mathbb{Z}_N^*$  be a **hash function** mapping arbitrary bit strings to our group  $\mathbb{Z}_N^*$ . Then, **FDH-RSA** is defined as follows:

<p><b>Algorithm</b> <math>\text{SIGN}(M, (N, d)) :</math></p> <p style="padding-left: 20px;"><math>y = H(M)</math></p> <p style="padding-left: 20px;">return <math>y^d \bmod N</math></p>	<p><b>Algorithm</b> <math>\mathcal{V}_F(M, s, (N, e)) :</math></p> <p style="padding-left: 20px;"><math>y = H(M)</math></p> <p style="padding-left: 20px;">if <math>y^e \bmod N \neq s</math>,</p> <p style="padding-left: 40px;">return 0</p> <p style="padding-left: 20px;">return 1</p>
---	--

**Theorem 10.1.** *The FDH-RSA scheme is **UF-CMA** secure under the **random oracle** model.*

*Specifically, let  $\mathcal{K}_{\text{rsa}}$  be a key generating algorithm for RSA and  $\mathcal{DS}$  be the FDH-RSA signature scheme. Then, let  $\mathcal{F}$  be a forging adversary making at most  $q_H$  queries to its hash oracle and at most  $q_S$  queries to its signing oracle. Then, there exists an adversary  $\mathcal{I}$  with composable resources such that*

$$\text{Adv}_{\mathcal{DS}}^{\text{uf-cma}}(\mathcal{F}) \leq (q_S + q_H + 1) \cdot \text{Adv}_{\mathcal{K}_{\text{rsa}}}^{\text{owf}}(\mathcal{I})$$

*where **owf** refers to the strength of RSA as a one-way function—the ability to find  $m$  given  $m^e$  without knowing the **trap door**  $d$ .*

The intuition behind this result is worth understanding since it uses the random oracle model which we’ve largely only alluded to previously. The proof proceeds by leveraging the contrapositive as we’ve seen before: if an adversary  $\mathcal{F}$  exists that can break UF-CMA on with FDH-RSA, then another adversary  $\mathcal{I}$  can use  $\mathcal{F}$  to break RSA’s **one-wayness**. Let’s walk through how this happens.

## Security Proof

In  $\mathcal{I}$ ’s case, it receives three values—the public key tuple  $(N, e)$  and the ciphertext  $y = m^e \bmod N$ —and its goal is to find  $x$ . It models itself as the both the signing and hash oracle for  $\mathcal{F}$ . Given that  $\mathcal{F}$  can crack UF-CMA security, it will return some  $(M, H(M)^d)$ , where  $M$  was never queried to the signing oracle. This means that we know for a fact that  $\mathcal{F}$  needs to make at *least* one query to the hash oracle (to hash its forged  $M$ ); when this occurs,  $\mathcal{I}$  will instead provide  $y$  as the hash value rather than some “true”  $H(M)$ .<sup>1</sup> The returned signature is then  $y^d \bmod N$ , which is the original message that was given as a challenge to  $\mathcal{I}$ !

This devious plan has a little bit of resistance, though: it’s quite likely that  $\mathcal{F}$  needs to make legitimate queries to the signing and hashing oracles in order to form its forgery  $M$ , yet  $\mathcal{I}$  cannot emulate valid signatures! Or can it...?

Recall the second trick we used to break **UF-CMA** for plain RSA: given an arbitrary signature, we can derive its original message. Now,  $\mathcal{I}$  will simply output random signatures for queries to the oracle and track which messages resulted in which random signatures. This way, it can respond with the same signature when given the same message. Furthermore, this means it can respond with the correct *legitimate* hash for a message: for a message  $m$ , its signature should be  $H(M)^d \bmod N$ , so  $\mathcal{I}$  uses  $s^e = H(M)$ , since  $s^{ed} \equiv s$  (where  $s$  is the randomly-chosen signature for  $m$ ).

## Implication

Because of these additive factors, one needs many more bits to ensure the same level of security. For example, to achieve the same security as a 1024-bit RSA key, one needs 3700 bits for FDH-RSA

<sup>1</sup> This is the key of the **random oracle** model: rather than computing the hash “locally” (i.e. using a known hashing algorithm), the adversary  $\mathcal{F}$  *must* instead consult the oracle which is controlled by  $\mathcal{I}$ , which doesn’t necessarily need to provide it with values consistent with the properties expected by a scheme.

(assuming the [GNFS](#) is the best algorithm). In practice, this advice is rarely followed because despite the theoretical bound, no practical attacks exist.

### 10.2.3 Probabilistic Signature Scheme

To do better than the additive factors impacting FDH-RSA's security, we can introduce some randomness to the hash function; this is [PSS](#). Specifically, we concatenate the message with  $S$  bits of randomness before signing:

**Algorithm**  $\text{SIGN}(M, (N, d)) :$

```

 $r \xleftarrow{\$} \{0, 1\}^S$ 
 $y = H(r \parallel M)$ 
return  $y^d \bmod N$ 

```

**Algorithm**  $\text{VF}(M, s, (N, e)) :$

```

Parse  $s$  as  $(r, x)$  where  $|r| = S$ 
 $y = H(r \parallel M)$ 
if  $y^e \bmod N \neq s$ ,
    return 0
return 1

```

This variant makes much stronger guarantees about security, adding only a small, relatively-insignificant factor given a large-enough choice of  $S$ :

**Theorem 10.2.** *The PSS scheme is [UF-CMA](#) secure under the [random oracle](#) model with the following guarantees (where the parameters are the same as in [Theorem 10.1](#), and  $S$  is the number of bits of randomness):*

$$\text{Adv}_{\text{pss}}^{\text{uf-cma}}(\mathcal{F}) \leq \text{Adv}_{\mathcal{K}_{\text{rsa}}}^{\text{owf}}(\mathcal{I}) + \frac{(q_H - 1) \cdot q_S}{2^S}$$

## 10.3 ElGamal

We can build a signature scheme from discrete log-based encryption schemes like [ElGamal](#) just like we can with factoring schemes like [RSA](#).

Recall that in the [ElGamal](#) scheme, we start with a group  $G = \mathbb{Z}_p^* = \langle g \rangle$  where  $p$  is a prime, and the secret key is just any group element  $x \xleftarrow{\$} \mathbb{Z}_{p-1}^*$ . The public key is then its corresponding exponentiation:  $pk = g^x \bmod p$ . Then, we also need a bitstring to group hash function as before,  $H : \{0, 1\}^* \mapsto \mathbb{Z}_{p-1}^*$ . With that, the signing and verification algorithms are as follows:

**Algorithm**  $\text{SIGN}(M, x) :$

```

 $m = H(M)$ 
 $k \xleftarrow{\$} \mathbb{Z}_{p-1}^*$ 
 $r = g^k \bmod p$ 
 $s = k^{-1}(m - xr) \pmod{p-1}$ 
return  $(r, s)$ 

```

**Algorithm**  $\text{VF}(M, (r, s), pk) :$

```

 $m = H(M)$ 
if  $r \notin G \vee s \notin \mathbb{Z}_{p-1}^*$ 
    return 0
if  $pk^r \cdot r^s \equiv g^m \pmod{p}$ 
    return 1
return 0

```

**Correctness** This likely isn't immediately apparent, but when we recall that  $p - 1$  is the order of  $\mathbb{Z}_p^*$ , it emerges cleanly:

$$\begin{aligned}
 pk^r \cdot r^s &\equiv g^{xr} \cdot g^{ks} \pmod{p} \\
 &\equiv g^{xr} \cdot g^{ks \bmod p-1} \pmod{p} && \text{see (8.1)} \\
 &\equiv g^{xr} \cdot g^{k(k^{-1}(m-xr)) \bmod p-1} \pmod{p} && \text{now we can} \\
 &&& \text{substitute for } s \\
 &\equiv g^{xr} \cdot g^{m-xr \bmod p-1} \pmod{p} && \text{cancel inverse} \\
 &\equiv g^m \pmod{p} && \text{combine exponents}
 \end{aligned}$$

**Security** The security of ElGamal signatures under **UF-CMA** has not been proven, even when applying the additional **random oracle** assumption that the other schemes made. There are proofs for variants of the scheme that are not used in practice, but (apparently?) they are “close enough” to grant ElGamal legitimacy.

## 10.4 Digital Signature Algorithm

The **DSA** scheme appears similar to the **ElGamal** scheme on the surface in the way it signs messages, but it increases security by using a second prime  $q$  rather than leveraging  $p - 1$  and Property 8.1 as we did above.

First, it requires two primes  $p$  and  $q$  configured such that  $q$  divides  $p - 1$  (we don't restrict  $q$  to be exactly half of  $p - 1$ , but safe primes are a popular choice). Then, we have the group  $G = \mathbb{Z}_p^* = \langle h \rangle$  and let  $g = h^{\frac{p-1}{q}}$  so that  $g \in G$  has order  $q$ .<sup>2</sup>

As before, the private key is a random exponent,  $x \xleftarrow{\$} \mathbb{Z}_p^*$ , and the public key comes from the generator,  $pk = g^x \bmod p$ . With that, the scheme is defined as follows:

**Algorithm**  $\text{SIGN}(M, x)$  :

```

 $m = H(M)$ 
 $k \xleftarrow{\$} \mathbb{Z}_q^*$ 
 $r = (g^k \bmod p) \bmod q$ 
 $s = k^{-1}(m + xr) \bmod q$ 
return  $(r, s)$ 

```

**Algorithm**  $\text{VF}(M, (r, s), pk)$  :

```

 $m = H(M)$ 
 $w = s^{-1} \bmod q$ 
 $u_1 = mw \bmod q$ 
 $u_2 = rw \bmod q$ 
 $v = (g^{u_1} \cdot pk^{u_2} \bmod p) \bmod q$ 
if  $v = r$ 
    return 1
return 0

```

**Correctness** As before, we can work through the math:

$$\begin{aligned}
 r &\stackrel{?}{=} v \\
 g^k \bmod p &\stackrel{?}{=} (g^{u_1} \cdot pk^{u_2} \bmod p) \pmod{q} \\
 &\stackrel{?}{=} (g^{mw \bmod q} \cdot pk^{rw \bmod q} \bmod p) \pmod{q} \\
 &\stackrel{?}{=} (g^{ms^{-1} \bmod q} \cdot pk^{rs^{-1} \bmod q} \bmod p) \pmod{q}
 \end{aligned}$$

<sup>2</sup> Recall that the order of a group element  $a$  is the smallest integer  $n$  fulfilling  $a^n = 1$  (see paragraph 8.3). In this case, it means  $g^q \equiv 1 \pmod{p}$ .

$$\begin{aligned}
& \stackrel{?}{\equiv} (g^{ms^{-1} \bmod q} \cdot g^{xrs^{-1} \bmod q} \bmod p) \pmod{q} && pk = g^x \bmod p \\
& \stackrel{?}{\equiv} (g^{ms^{-1} + xrs^{-1} \bmod q} \bmod p) \pmod{q} && \text{combine exponents} \\
& \stackrel{?}{\equiv} (g^{s^{-1}(m+xr) \bmod q} \bmod p) \pmod{q} && \text{factor out } s^{-1} \\
& \stackrel{?}{\equiv} (g^{(k^{-1}(m+xr))^{-1}(m+xr) \bmod q} \bmod p) \pmod{q} && \text{substitute } s \\
& \stackrel{?}{\equiv} (g^{(k^{-1})^{-1} \bmod q} \bmod p) \pmod{q} && \begin{array}{l} a(ab)^{-1} = b^{-1}, \\ \text{here, } a = mx + r \\ \text{and } b = k^{-1} \end{array} \\
& \stackrel{?}{\equiv} g^{k \bmod q} \bmod p \pmod{q} \\
& \equiv g^k \bmod p \pmod{q} \quad \checkmark && \begin{array}{l} \text{since } k \in \mathbb{Z}_q^*, \bmod q \\ \text{has no effect} \end{array}
\end{aligned}$$

This version of DSA works only with groups modulo a prime, but there is a version called **ECDSA** designed for elliptic curves.

**Security** The security of DSA under **UF-CMA** was not proven until 2016 under the hardness of discrete log and **random oracle** assumptions. The proof also confirmed DSA's superiority in terms of efficiency: a 320-bit signature has security on-par with a 1024-bit signature in **ElGamal**.

## 10.5 Schnorr Signatures

There is yet another class of digital signature schemes called **Schnorr** signatures. They are very similar to **PSS**, using both randomness and hashing for security.

We once again start with  $G = \langle g \rangle$ , a cyclic group of prime order  $p$ . We have a bit-string to integer hash function:  $H : \{0, 1\}^* \mapsto \mathbb{Z}_p$ , and our secret and public keys are  $x \xleftarrow{\$} \mathbb{Z}_p^*$  and  $g^x \in G$  (note the lack of mod here).

**Algorithm**  $\text{SIGN}(M, x)$  :

```

 $r \xleftarrow{\$} \mathbb{Z}_p$ 
 $R = g^r$ 
 $c = H(R \parallel M)$ 
 $s = (xc + r) \bmod p$ 
return  $(R, s)$ 

```

**Algorithm**  $\mathcal{V}_F(M, (R, s), pk)$  :

```

 $m = H(M)$ 
if  $R \notin G$ 
    return 0
 $c = H(R \parallel M)$ 
if  $g^s = R \cdot pk^c$ 
    return 1
return 0

```

**Correctness** Notice the odd difference in this scheme: many values (like  $R$ ) are not calculated under a modulus. This makes correctness trivial to verify:

$$R \cdot pk^c = g^r \cdot (g^x)^c = g^{xc+r} = g^s \quad \checkmark$$

**Security** The Schnorr signature scheme works on arbitrary groups as long as they have a prime order. It has been proven to be secure under **UF-CMA** with the **random oracle** and discrete log assumptions for modulo groups,<sup>3</sup> and is as efficient as **ECDSA** with a 160-bit elliptic curve group.

<sup>3</sup> It's worth noting that this security proof is pretty "loose."



## 10.6 Scheme Variants

As with asymmetric encryption schemes (see [section 9.8](#)), there are a number of variants on digital signatures schemes that are worth highlighting.

**multi-signatures** In this variant, several signers create a signature for a single message in a way that is much more efficient than repeatedly using a generic signature scheme that isn't meant for multiple signatures.

These schemes are used frequently in cryptocurrencies like [Bitcoin](#) to have multi-party consensus on complex transactions.

**aggregate signatures** These are similar to multi-signatures, but the parties are all signing different messages. The purpose of these schemes is still to do this in a way that is more efficient than with standard one-user-one-message schemes.

**threshold signatures** This variant is designed to provide a method for group consensus. A group of  $n$  users shares a public key, and each of them only has a piece of the secret key. In order to sign a message, at least  $t \leq n$  users need to cooperate. We'll allude to similar ideas later when discussing [key distribution](#).

**group signatures** A group of users holds a single public key. Each user can *anonymously* sign messages on behalf of the group; their identity is hidden except from the manager of the group who controls the joining and revocation of group "members." A similar variant called **ring signatures** drops the manager and allows members to always be anonymous.

**blind signatures** This variant allows users to obtain signatures from a signer without the signer knowing what it was that they signed. This may seem odd, but is actually very useful in many applications like password strengthening or anonymizing digital currency (through a centralized bank, not cryptocurrency).

### 10.6.1 Simple Multi-Signature Scheme

Recall that the purpose of a multi-signature scheme is for many users to be able to sign the same message efficiently. For this scheme, we will operate under the assumption that we're working in a group  $G = \langle g \rangle$  for which the DDH problem is *easy*, and there's an efficient algorithm for it. Namely,  $\mathcal{V}_{\text{ddh}}[g, g^x, g^y, g^z]$  outputs 1 if  $z \equiv xy \pmod{|G|}$ .

Our secret key in this context will be  $x \in \mathbb{Z}_{|G|}$  and the public key will be  $pk = g^x$ . To sign a message, we hash it and raise it to the  $x$  power (much like in [FDH-RSA](#)):  $\text{SIGN}(M, x) = H(M)^x$ . To verify a message, we'll run the DDH verification algorithm as follows:  $\mathcal{V}_F(M, s, pk) = \mathcal{V}_{\text{ddh}}(g, pk, H(M), s)$ .

Notice that  $pk = g^x$ ,  $H(M)$  results in some  $g^y$  since  $g$  is a generator, and  $s$  is  $g^{xy}$  for a valid signature based on the definition of  $\text{SIGN}$ .

Now we'll apply this to the multi-signature context. For demonstration, suppose we have three users, each with their own key pair:  $(x_1, g^{x_1}), (x_2, g^{x_2}), (x_3, g^{x_3})$ . Given a message  $m$ , each user signs the message as described above: we get  $s_1 = H(m)^{x_1}$ ,  $s_2 = H(m)^{x_2}$ , and  $s_3 = H(m)^{x_3}$ .

The multi-signature is then the product of the individual signatures:

$$s^* = \prod_i s_i = H(m)^{\sum_i x_i}$$

Concretely for our example,  $s^* = H(m)^{x_1+x_2+x_3}$ . Our verification algorithm for multi-signatures is:

$$\mathcal{V}_F(M, s_1, s_2, s_3, pk_1, pk_2, pk_3) = \mathcal{V}_{\text{ddh}}(g, pk_1 pk_2 pk_3, H(M), s^*)$$

Notice that the DDH scenario is fulfilled for a valid signature:

$$\begin{aligned} pk_1 pk_2 pk_3 &= g^{x_1+x_2+x_3} \\ H(M) &= g^y \\ s^* &= g^{y(x_1+x_2+x_3)} \end{aligned}$$

The efficiency of the multi-signature signing and verifying algorithms is apparent: for  $n$  users, rather than  $n$  verifies, we only need to do a single verify (note that we still need  $n + 1$  signs). The cumulative verification only requires some additional multiplications which are much faster than exponentiations.

### 10.6.2 Simple Blind Signature Scheme

Recall that the purpose of a blind signature scheme is for users to be able to get signatures from an authority without the authority knowing what they signed.

Our authoritative signer will be  $S$ , with the [RSA](#) public key  $pk = (N, e)$  and the secret key  $d$ . We'll also have a hash function  $H$  as we've been using in this chapter: it maps from arbitrary bit-strings to a value in the group  $\mathbb{Z}_N$ .

Our user  $U$  wants to get a message  $m$  signed. They choose a random value  $r \xleftarrow{\$} \mathbb{Z}_N^*$  and submit  $(H(M) \cdot r^e) \bmod N$  to the signer  $S$ . The signer will follow normal RSA signing algorithm and return the signature  $s = (H(M) \cdot r^e)^d \bmod N$ . Notice, though, that  $r^{ed} \equiv r \pmod{N}$ , so the user can obtain a signature on the original message by applying the inverse  $r^{-1}$ :

$$\begin{aligned} r^{-1} \cdot s &= r^{-1} \cdot (H(M) \cdot r^e)^d \bmod N \\ &= r^{-1} \cdot (H(M)^d \cdot r \bmod N) \\ &= H(M)^d \bmod N \end{aligned}$$

Neither  $M$  nor  $H(M)$  was revealed to the signer at any point:  $r$  is a random number, and a meaningful value multiplied by a random number still looks like a random number. Further, this scheme can be proven to be unforgeable.

## 10.7 Signcryption

Our final asymmetric construction—a primitive called [signcryption](#)—will achieve all three of our security goals: message [confidentiality](#) (contents are private), [integrity](#) (contents are unchanged), and [authenticity](#) (contents are genuine).

To fulfill these requirements, a simple asymmetric scenario is not enough: now, [both](#) the sender and the recipient must have public key pairs. As such, signcryption must be considered in the multi-user setting. Our notions of integrity from symmetric cryptography (see [INT-CTXT](#)) and privacy ([IND-CCA](#) and friends) transfer over in a similar fashion.

Though additional attacks need to be considered (one called “identity fraud” that wasn't present in the separate encryption or signature worlds), the security result is reminiscent of the one for [Hybrid Encryption](#):

**Property 10.1.** *If an encryption scheme is [IND-CPA](#) secure and a signature scheme is [SUF-CMA](#) secure, then the [encrypt-then-sign](#) signcryption scheme is [IND-CCA](#) and [INT-CTXT](#) secure in the two-user model.*

This new SUF-CMA security notion (**S** = strongly) is stronger than **UF-CMA**. It's satisfied by all practical schemes, and is equivalent to UF-CMA for deterministic schemes. The encrypt-then-sign construction is exactly what it sounds like: first, we do encryption (see schemes in the [previous chapter](#)), then sign the result.

To achieve security in the multi-user model, users need to take extra precautions: add the public key of the *sender* to the message being **encrypted**, and add the public key of the *receiver* to the message being **signed**.

# SECRET SHARING

In this final chapter, we'll discuss the various nuances in how crucial information that we've been relying upon in the previous chapters is distributed. This includes things of authenticating public keys, securely sharing secrets among groups, and using [session keys](#). We'll also briefly discuss some potpourri topics like passwords and [PGP](#).

## 11.1 Public Key Infrastructure

The big assumption in the world of [asymmetric encryption](#) is that there's a trusted [public key infrastructure](#) (or PKI) that somehow securely provides everyone's *authentic* public keys. Without this assumption, we're back at the [key distribution](#) problem: how do we obtain Bob's key in the first place, and furthermore, how do we know that what we obtained truly is *Bob's* public key? In this section, we'll discuss how PKI is designed and ultimately see the unfortunate fragility of cryptography in the real world.

Public key infrastructure relies on a trusted third party as a “starting point” for verifying key authenticity. This is called the [certificate authority](#) or CA for short. The first “gotcha” of this design is immediately apparent: we *must* start by assuming that the public key of the CA is known and trusted by everyone. Though this can generally be guaranteed<sup>1</sup> simply by hard-coding a list of public keys in the software itself.

### 11.1.1 Registering Users

Suppose a user, who knows the CA's public key  $pk_{CA}$ , wants to register their own public key  $pk_U$  with the CA so that others can send them message. They first need some sort of identity—which we'll call  $id_U$  to associate with their public key. They also need a secure communication channel with the CA, which can be achieved by any asymmetric scheme since  $pk_{CA}$  is known. Registration is then done as follows:

- The user sends  $(id_U, pk_U)$  to the CA.
- The CA needs to verify that the user truly knows the associated secret key, so it generates and send a random challenge  $R$  for the user to sign.
- The user signs and sends the challenge,  $s = \text{SIGN}(sk_U, R)$ .
- Finally, the CA checks the validity of the signature:  $\mathcal{V}_F(pk_U, s) \stackrel{?}{=} 1$ .

---

<sup>1</sup> ... by a small leap of faith that your software download is genuine, which can usually be made easily, since you start with a browser provided by your operating system, and since you (hopefully) installed it from a disk OR somehow got it from a source that installed it from a disk... The chain of trust can be very, *very* deep, but regardless, at some point we are relying on the physical distribution of genuine data to fully guarantee authenticity.

After the CA determines that this is a genuine user, it issues a **certificate**: a collection of information about the user that is signed by the CA itself:

$$\text{cert}_U = \text{SIGN}(sk_{CA}, (id_U, pk_U, \text{expiration}, \dots))$$

The user can obviously verify the validity of the certificate as a sanity check. Now, they can present the certificate to anyone who requests their public key to show that it is genuine and authentic: the recipient simply independently verifies the certificate against  $pk_{CA}$ .

### 11.1.2 Revocation

The certificate authority does more than act as an authentic repository of public keys. It also handles key **revocation**: if a user's key is compromised, rotated, or otherwise no longer trusted before its expiration date, they can notify the CA who will update its certificate revocation list (CRL) accordingly. This list is public, and anyone verifying a certificate should ideally cross-reference it against this list.

Practically, though, users will instead download a copy of the CRL periodically. Naturally, this human-dependent element has a flaw: between the time of key compromise, revocation, and updated CRL, the attacker wreak havoc impersonating the user, signing and encrypting malicious messages.

Shockingly, 8-20% of all issued certificates are revoked. This means CRLs can grow very large and unwieldy; revocation is one of the biggest pain points of widespread PKI adoption.

**OCSP** The on-line certification status protocol, or **OCSP**, is a method to enable realtime checks of whether or not a certificate has been revoked. To verify  $\text{cert}_{\text{alice}}$ , Bob passes it along to the CA immediately. Of course, this kind of defeats the purpose of certificates in the first place, since Bob simply could've acquired Alice's public key from the CA directly.

### 11.1.3 Scaling

*(or, as I like to call it, the CA house of cards)*

In order to avoid overwhelming a single centralized authority with certificate requests and CRL queries, in practice we have a hierarchy of certificate authorities. At the top is the "root" CA, and it delegates responsibility (and trust) to "local" CAs to handle issuing and revoking certificates. The verification process then involves following and validating the chain of trust up until the final root CA signature. The **X.509** protocol is the industry standard for this process.

Naturally, if *any* of these "trusted" CAs (of which there are **hundreds**, see the study by [Fadai et al.](#)) is breached or its algorithms compromised, it has massive implications for: anyone who had their certificates signed by the CA, any downline CAs that were "ordained" by the CA, and any *new* certificates that are maliciously signed by the CA after the compromise.

### 11.1.4 An Alternative: Pretty Good Privacy

The **PGP** protocol is an attempt at building PKI without relying on centralized certificate authorities. In this model, there is a **web of trust**: when you get Alice's key from Carol, its authenticity is directly correlated with your trust in Carol.

This model obviously requires user involvement, and is just as fallible to compromise as the CA-based approach. The difference is in the implicit trust of hundreds of CAs that your browser and OS "approve" versus the explicit trust of members of your web.

## 11.2 Secret Sharding

We briefly discussed [signatures](#) in the multi-user setting (refer to most of the methods in [section 10.6](#)). Consider now the idea of splitting and distributing a *single* secret among a group of users with the following goals:

- only a consensus of some minimal collection of users results in the derivation of the original secret, and
- no partial information about the secret is gained from collusion without full participation.

In other words, given a secret  $k$  shared among  $n$  users, any consensus of less than  $t$  users results in no information about  $k$ , while consensus of at *least*  $t$  users enables derivation of  $k$ . This prevents the secret from being compromised even if  $t - 1$  devices / users / etc. are compromised and conspire maliciously.

We'll denote this as a  $(t, n)$  sharing scheme.

### 11.2.1 Shamir's Secret Sharing

Recall from geometry that a line is uniquely defined by two points. When given just one point, there's an infinite number of lines that pass through it; the single point gives you no information. A parabola can be uniquely defined by three points: we get a system of three equations for each of the three unknowns in the parabola  $ax^2 + bx + c = y$ . Any number of parabolas could be designed to pass through just two of the points. For a third-degree polynomial, we'd need four points for uniqueness, and so on. . .

This is the intuition behind [Shamir's secret sharing](#) scheme:<sup>2</sup> we can achieve a  $(t, n)$  sharing scheme by forming an  $(t - 1)^{\text{th}}$ -degree polynomial and distributing  $n$  evaluations of it to the participants. Any  $t$  points will uniquely describe the polynomial.

#### Protocol

Let's describe the  $(t, n)$  scheme formally. We first choose  $p$  to be a large prime, and our secret is any  $z \in \mathbb{Z}_p$ . Then,

- Choose  $t - 1$  random elements:  $a_1, a_2, \dots, a_{t-1} \in \mathbb{Z}_p$ .
- The secret will be denoted  $a_0 = z$ .
- Now, view these elements as coefficients of a polynomial:

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_{t-1}x^{t-1}$$

- Create  $n$  shares by evaluating  $f(x)$ :

$$y_i = f(i)$$

For simplicity, we can evaluate them on  $i = 0, 1, \dots, n$ , but we can also choose random locations, instead.

To recover the secret, we use [Lagrange interpolation](#) from the set  $S$  of any  $t$  points:

$$z = a_0 = f(0) = \sum_{i \in S} \left( y_i \cdot \prod_{j \in S, j \neq i} \frac{-i}{i - j} \right)$$

<sup>2</sup> This is the same Shamir who is the "S" in [RSA](#)!

This scheme is *unconditionally* secure: given any  $t - 1$  shares, we gain no information about the secret, yet with  $t$  shares we recover  $z$  in its entirety.

### Weakness

Unfortunately, there's a fundamental flaw in this scheme: if **any** parties cheat during reconstruction, the true secret cannot be recovered *and* there's no way to detect cheating; **verifiable secret sharing** schemes are designed to get around this problem.

## 11.2.2 Threshold Schemes

It may be desirable to allow the group to perform operations involving the secret key without holding it in its entirety. We alluded to this when discussing **threshold signatures**, but a similar principle can be applied to encryption. The party can combine partial shares, encrypted with their respective partial keys, to form a full, properly-encrypted ciphertext using the full secret key without ever actually reconstructing it.

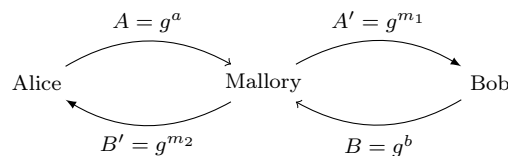
## 11.3 Man-in-the-Middle Attacks

Consider as a case study the **Diffie-Hellman key exchange** protocol we discussed earlier. Alice and Bob want to communicate securely: Alice sends Bob a **public key**  $A = g^a$ , while Bob sends her  $B = g^b$ . Together, they can form the mutual secret  $k = g^{ab}$  without revealing the private exponents  $a$  and  $b$  (neither to the world nor to each other).

If we presume the **computational Diffie-Hellman** and **decisional Diffie-Hellman** problems to be hard, this is secure against *passive*, eavesdropping attackers.<sup>3</sup>

All of the asymmetric schemes we've discussed thus far are unfortunately only secure against *passive* attackers: attackers who can eavesdrop on, but not modify, messages between parties.

However, consider what happens when the messages are routed through an *active* attacker, Mallory, who can modify messages as she sees fit. She could take the form of a compromised server, a hacked Internet service provider, a government wire-tap, etc. When she sees Alice transmitting  $g^a$ , she instead transmits to Bob a malicious  $g^{m_1}$ . Similarly, she transmits  $g^{m_2}$  to Alice instead of Bob's  $g^b$ .



Now, Bob forms the secret  $g^{bm_1}$ , and Alice forms the secret  $g^{am_2}$ . It seems like they can't communicate effectively: attempting to decrypt Alice's messages with Bob's secret would result in gibberish or failure! However, if Mallory continues to facilitate communication, she can ensure that Alice and Bob receive valid ciphertexts while also reading the plaintexts in-transit.

When Alice sends Bob a message encrypted under  $g^{am_2}$ , Mallory decrypts it, then **re-encrypts** it using  $g^{bm_1}$  (and vice-versa for Bob's messages to Alice). This is called a **man-in-the-middle attack**, and it leads to an unfortunate conclusion:

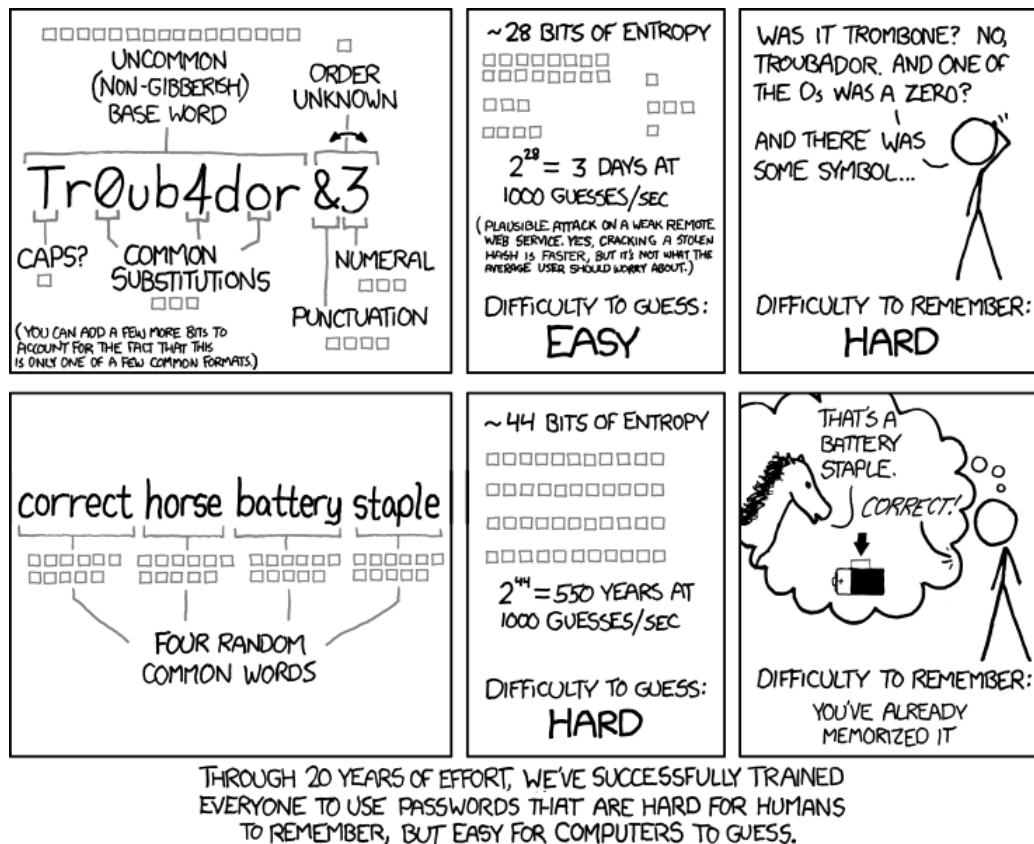
**Property 11.1.** *Asymmetric key exchange schemes cannot be made secure against active attackers when starting from scratch.*

<sup>3</sup> Note that even though the DDH problem is not hard under  $\mathbb{Z}_p^*$ , it's still often used in practice. This is because of a small modification that enables provable security of the secret: the true mutual secret is instead the *hashed*  $g^{xy}$ .

An *a priori* information advantage against Mallory is *required*, and it often comes in the form of long-term keys: the hope is that Mallory is not ever-present on the communication channel.

## 11.4 Passwords

Despite all of the fantastic, provably-secure cryptographic methods and schemes we’ve studied in the last 88 pages, human-memorable passwords remain the weakest link in many security architectures.



**Figure 11.1:** The classic [XKCD comic](#) demonstrating the futility of memorizing “complex” passwords which have low entropy relative to the ease of *long* passwords that are far harder to guess.

Dictionary attacks—brute force methods that simply compare dictionary-like passwords against the hashes of a compromised server—are still very effective.<sup>4</sup> Studies show that despite of efforts to complicate requirements (which are in and of themselves largely ineffective, see [Figure 11.1](#)), many passwords are still just words in the dictionary.

The key to a secure future is simply to move away from passwords: the Fast ID Online alliance is moving towards an ambitious idea of secure authentication based on secure *devices* (“something you *have*”) and biometrics (“something you *are*”) rather than on passwords (“something you *know*”).

<sup>4</sup> Note that “salting” a password hash—appending and storing a random value so that identical passwords have different hashes,  $H(p \parallel r_1) \neq H(p \parallel r_2)$ —only helps against *mass* cracking of compromised passwords. They have no bearing on the time it takes to crack a *specific* user’s password.



# EPILOGUE

Though we’ve covered many foundational concepts in applied cryptography at length—ways to ensure [confidentiality](#), [integrity](#), and [authentication](#) in both the symmetric and asymmetric setting; security definitions; proofs of correctness and security; and more—there are still many advanced topics we’ve left out:

- secure [multi-party computation](#): this primitive allows two or more parties to mutually compute some information while keeping their independent inputs private.

The classic example of MPC is [Yao’s millionaire problem](#), in which two rich people want to determine which of them is richer without revealing their net worth to the other. There is also the simpler, more-intuitive [socialist millionaire problem](#), in which two rich people simply want to see if they have the *same* net worth.

[Wikipedia](#)

- [zero-knowledge proofs](#): this related primitive allows someone (the “Prover”) to prove that they have a solution to a particular problem to someone else (the “Verifier”) without revealing anything about the solution itself.
- [Merkle trees](#) and the [blockchain](#): these are the founding cryptographic primitives powering many cryptocurrencies.
- [post-quantum cryptography](#): if quantum computers continue to increase in their computational capability, they will completely break asymmetric schemes that rely on the hardness of the discrete log problem by turning it into a polynomial-time operation.

Many innovations have risen to combat this threat: novel, quantum-resistant cryptographic schemes are often based on [lattices](#), which are a mathematical construct that I don’t (yet) understand.

I cover some of these topics in [??](#), but the content comes from independent research rather than professionally-curated content.

At the end of the day, the content we’ve learned in this course should let you analyze the efficacy of security protocols you encounter in the wild, though it’s still universally-advised that rolling your own low-level cryptographic primitives is not a good idea.

# INDEX OF TERMS

*n*-IND-CCA, [73](#)

*n*-IND-CPA, [72](#)

AEAD, [41](#)

AES, [10](#), [18](#), [19](#), [29](#), [46](#), [55](#), [61](#)

aggregate signatures, [81](#)

associated data, [41](#)

asymmetric cryptography, [5](#), [55](#)

asymmetric encryption, [84](#)

attribute-based encryption, [74](#)

authenticated encryption, [37](#)

authentication, [89](#)

authenticity, [25](#), [26](#), [37](#), [46](#), [75](#), [82](#)

Bézout's identity, [54](#)

birthday paradox, [20](#), [22](#), [23](#), [29](#), [32](#), [46](#)

Bitcoin, [81](#)

blind signatures, [81](#)

block cipher, [10](#), [17](#), [19](#), [20](#), [23](#), [25](#), [27](#), [41](#), [43](#),  
[46](#), [61](#)

blockchain, [30](#), [89](#)

both, [82](#)

cdh-advantage, [62](#)

certificate, [85](#)

certificate authority, [84](#)

Chinese remainder theorem, [60](#), [60](#), [70](#)

collision resistant, [30](#), [31](#), [32](#), [34](#), [35](#)

confidentiality, [26](#), [27](#), [37](#), [38](#), [46](#), [75](#), [82](#), [89](#)

cr-advantage, [31](#)

Cramer-Shoup, [67](#), [74](#)

cryptographically-secure hash function, [35](#), [40](#),  
[68](#)

ddh-advantage, [62](#), [62–64](#), [66](#)

DES, [10](#), [18](#)

Diffie-Hellman key exchange, [64](#), [87](#)

Diffie-Hellman, computational, [61](#), [64](#), [65](#), [87](#)

Diffie-Hellman, decision, [81](#)

Diffie-Hellman, decisional, [61](#), [65](#), [68](#), [87](#)

dl-advantage, [62](#), [62](#)

DSA, [79](#)

ECDSA, [80](#), [80](#)

ElGamal, [64](#), [67](#), [73](#), [78–80](#)

elliptic curve, [55](#), [56](#), [63](#)

encrypt-and-MAC, [38](#)

encrypt-then-MAC, [38](#)

encrypt-then-sign, [82](#)

Euclidean algorithm, [54](#)

Euclidean algorithm, extended, [54](#), [60](#), [69](#), [70](#)

Euler's theorem, [70](#), [70](#), [71](#)

exclusive-or, [6](#)

exhaustive key-search, [17](#), [46](#)

FDH-RSA, [77](#), [81](#)

Fermat witness, [56](#)

Fermat's little theorem, [56](#), [70](#), [71](#)

forward secrecy, [44](#)

general number field sieve, [55](#), [62](#), [78](#)

generator, [42](#)

greatest common divisor, [51](#)

group signatures, [81](#)

hash function, [30](#), [71](#), [75](#), [77](#)

Hstad's broadcast attack, [70](#), [72](#)

HMAC, [35](#)

homomorphic encryption, [74](#), [76](#)

identity-based encryption, [74](#)

impossibility result, [9](#)

IND-CCA, [23](#), [25](#), [27](#), [37](#), [40](#), [46](#), [49](#), [65](#), [67](#), [68](#),  
[71](#), [72](#), [82](#)

IND-CCA advantage, [24](#)

IND-CPA, [14](#), [15](#), [18](#), [19](#), [23](#), [25](#), [37](#), [40](#), [46](#), [49](#),  
[62](#), [65](#), [66](#), [71–73](#), [82](#)

IND-CPA advantage, [14](#), [16](#), [21](#)

IND-CPA-cg, [16](#), [21](#), [66](#)

IND-CPA-cg advantage, [17](#)

INDR, [43](#)

INDR advantage, [43](#)

initialization vector, [11](#), [16](#), [24](#), [41–43](#)

INT-CTXT, [37](#), [40](#), [82](#)

INT-CTXT advantage, [37](#)

integrity, [25](#), [26](#), [37](#), [38](#), [46](#), [75](#), [82](#), [89](#)

Kerckhoff's principle, [9](#), [26](#)

key distribution, [5](#), [48](#), [74](#), [81](#), [84](#)

key separation principle, [38](#)

key-derivation function, [35](#)

- Legendre symbol, [58](#), [58](#), [59](#), [63](#), [65](#)
- MAC-then-encrypt, [38](#)
- man-in-the-middle attack, [87](#)
- Merkle tree, [89](#)
- Merkle-Damgård transform, [32](#), [32](#), [38](#)
- message authentication code, [26](#), [35](#), [38](#), [75](#)
- mode of operation, [10](#), [16](#), [17](#), [27](#), [31](#), [41](#), [42](#), [46](#), [61](#), [65](#)
- multi-party computation, [89](#)
- multi-signatures, [81](#)
- negligible, [53](#), [53](#), [62](#)
- OCSP, [85](#)
- offset codebook, [41](#)
- one-time pad, [6](#), [13](#), [22](#), [26](#), [42](#)
- one-way, [34](#), [35](#), [69](#), [77](#)
- ow-advantage, [34](#)
- perfect security, [7](#), [26](#)
- PGP, [84](#), [85](#)
- post-quantum cryptography, [89](#)
- PRF, [61](#)
- PRF advantage, [19](#), [21](#)
- PRF secure, [19](#), [22](#), [23](#), [25](#), [33](#), [38](#), [40](#)
- prime factorization, [63](#)
- private key, [48](#), [49](#), [64](#), [68](#)
- pseudorandom function, [18](#), [23](#), [38](#), [43](#)
- pseudorandom functions, [72](#)
- pseudorandom generator, [38](#), [42](#)
- PSS, [78](#), [80](#)
- public key, [48](#), [49](#), [64](#), [66](#), [68](#), [69](#), [84](#), [87](#)
- public key infrastructure, [84](#)
- random function, [25](#), [72](#)
- random oracle, [71](#), [72](#), [72](#), [77–80](#)
- RC4, [43](#)
- relatively prime, [71](#)
- replay attack, [27](#), [28](#)
- revocation, [85](#)
- ring signatures, [81](#)
- RSA, [61](#), [68](#), [76](#), [78](#), [82](#), [86](#)
- RSA-OAEP, [71](#), [72](#)
- safe prime, [56](#), [56](#), [63](#), [64](#)
- Schnorr, [80](#)
- searchable encryption, [74](#)
- secret key, [48](#)
- seed, [42](#), [45](#)
- session key, [84](#)
- Shamir's secret sharing, [86](#)
- Shannon-secure, [7](#), [9](#), [13](#), [22](#), [26](#)
- signature, [75](#), [81](#), [86](#)
- signcryption, [82](#), [82](#)
- socialist millionaire problem, [89](#)
- stream cipher, [42](#), [42](#)
- SUF-CMA, [82](#)
- symmetric key, [5](#), [61](#), [70](#), [75](#)
- threshold signatures, [81](#), [87](#)
- timing attack, [40](#), [47](#)
- totient function, [51](#), [68](#), [71](#)
- trap door, [69](#), [77](#)
- UF-CMA, [27](#), [29](#), [37](#), [75–80](#), [83](#)
- UF-CMA advantage, [27](#), [76](#)
- verifiable secret sharing, [87](#)
- web of trust, [85](#)
- X.509, [85](#)
- Yao's millionaire problem, [89](#)
- zero-knowledge proof, [89](#)