M⊕dern Cryptogr🔒phy

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

George Kudrayvtsev
george.k@gatech.edu

Last Updated: March 31, 2020
(through Module 12)

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` 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! 😇

Preface

> *I read that Teddy Roosevelt once said, "Do what you can with what you have where you are." Of course, I doubt he was in the tub when he said that.*
>
> — Bill Watterson, *Calvin and Hobbes*

Before we begin to dive into all things cryptography, I'll enumerate a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a "term;" this is some vocabulary that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the Index.
- An item that is **highlighted like this** is a "mathematical property;" such properties are often used in subsequent sections and their understanding is assumed there.
- An item in a maroon box, like...

Boxes: A Rigorous Approach

... this often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it's not immediately rewarding.

- An item in a **blue box**, like...

Quick Maffs: Proving That the Box Exists

... this is a mathematical aside; I only write these if I need to dive deeper into a concept that's mentioned in lecture. This could be

proofs, examples, or just a more thorough explanation of something that might've been "assumed knowledge" in the text.

- An item in a green box, like...

Definition 0.1: Example

... this is an important cryptographic definition. It will often be accompanied by a highlighted **term** and dive into it with some mathematical rigor.

Linky I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further.

Introduction

> *Cryptography is hard.*
>
> — Anonymous

The purpose of a cryptographic scheme falls into three very distinct categories. A common metaphor used to explain these concepts is a legal document.

- **confidentiality** ensures content *secrecy*—that it can't be read without knowledge of some secret. In our example, this would be like writing the document in a language nobody except you and your recipient understand.
- **authenticity** guarantees content *authorship*—that its author can be irrefutably proven. In our example, this is like signing the original document in pen (assuming, of course, your signature was impossible to forge).
- **integrity** guarantees content *immutability*—that it has not been changed. In our example, this could be that you get an emailed copy of the signed document to ensure that its language cannot be changed post-signing.

Note that even though all three of these properties can go hand-in-hand, they are not mutually constitutive. You can have any of them without the others: you can just get a copy of an unsigned document sent to you in plain English to ensure its integrity later down the line.

$$\underbrace{\text{crypto}}_{\text{secret}}\underbrace{\text{graphy}}_{\text{writing}}$$

Analysing any proposed protocol, handshake, or other cryptographic exchange through the lens of each of these principles will be enlightening. Not every scheme is intended to guarantee all three of them, and different methods are often combined to achieve more than one of these properties. In fact, cases in which only one of the three properties are necessary occur all the time. It's important to not make a cryptographic scheme more complicated than it needs to be to achieve a given purpose: complexity

breeds bugs.

TRIVIA: Cryptography's Common Cast of Characters

It's really useful to anthropomorphize our discussion of the mathematical intricacies in cryptography. For that, we use a cast of characters whose names give us immediate insight into what we should expect from them.

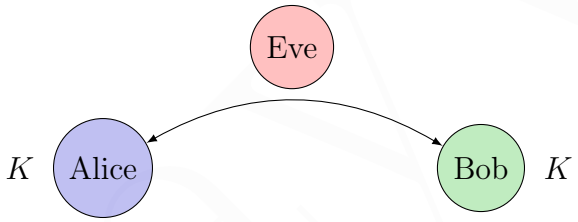
- **Alice** and **Bob** are the most common sender-recipient pairing. They are generally acting in good faith and aren't trying to break the cryptographic scheme in question. If a third member is necessary, **Carol** will enter the fray (for consistency of the allusion to Lewis Carroll's *Alice in Wonderland* ☺).
- **Eve** and **Mallory** are typically the two members trying to break the scheme. Eve is a *passive* attacker (short for eavesdropper) that merely observes messages between Alice and Bob, whereas malicious Mallory is an *active* attacker who can capture, modify, and inject her own messages into exchanges between other members.

You can check out the Wikipedia article on the topic for more historic trivia and the full cast of characters.

PART I

Symmetric Cryptography

The notion of **symmetric key**s 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 asymmetric cryptography.

Contents

Perfect Security

> *How long do you want these messages to remain secret? I want them to remain secret for as long as men are capable of evil.*
>
> — Neal Stephenson, *Cryptonomicon*

As mentioned in the Introduction, 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.

2.1 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 $\mathcal{M}$sgSp 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, $\mathcal{K}$eySp, 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:

$$\forall m \in \mathcal{M}\text{sgSp},\ \forall k \in \mathcal{K}\text{eySp} : \mathcal{D}(k, \mathcal{E}(k, m)) = m$$

An encryption scheme defines the message space and three algorithms: $(\mathcal{M}\text{sgSp}, \mathcal{E}, \mathcal{D}, \mathcal{K})$. 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 $K \xleftarrow{\$} \mathcal{K}\text{eySp}$. 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.

2.2 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: $\mathcal{K}\text{eySp} = \{0,1\}^n$
- the message space is the same: $\mathcal{M}\text{sgSp} = \{0,1\}^n$
- 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 2.1: **Shannon-secure**

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

$$\forall m_1, m_2 \in \mathcal{M}\text{sgSp}, \forall C : \\ \Pr[\mathcal{E}(K, m_1) = C] = \Pr[\mathcal{E}(K, m_2) = C] \tag{2.1}$$

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,[1] where all messages are equally-probable (i.e. independent of being) given a ciphertext:

$$\forall m \in \mathcal{M}\text{sgSp}, \forall C : \\ \Pr[M = m \,|\, C] = \Pr[M = m]$$

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

2.2.1 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?

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

Table 2.1: The truth table for XOR.

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 ($\frac{1}{2} = 50\%$).

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.

2.2.2 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.

[1] See *Definition 2.3* in Katz & Lindell, pp. 29, parts of which are available on Google Books.

Theorem 2.1. *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, $m \in \mathcal{M}\text{sgSp}$. Then, what's the probability that a randomly-generated key $k \in \mathcal{K}\text{eySp}$ 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 $1/2$:

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

Note that this is true $\forall m \in \mathcal{M}\text{sgSp}$, 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,[2] 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:

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

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 corrolary 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):

[2] 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...

Theorem 2.2. *If a scheme is Shannon-secure, then the key space cannot be smaller than the message space. That is,*

$$|\mathcal{K}eySp| \geq |\mathcal{M}sgSp|$$

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, ($K_1 \in \mathcal{K}$eySp and plaintext message, $m_1 \in \mathcal{M}$sgSp):

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

We know for a fact, then, that at least one key exists that can craft C; thus if we pick a key $K \in \mathcal{K}$eySp *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 $m_2 \in \mathcal{M}$sgSp which we can *never* get from decrypting C:

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

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:

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

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 $K_2 \in \mathcal{K}$eySp 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 3.1: 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 $\mathcal{D}_K(\cdot)$.

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

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: $\{0,1\}^n$.

3.1 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 simplicitly, we'll assume that our arbitrarily-long messages are actually a multiple of a block length; if they wereThesen't, we could just pad them, but we'll omit that detail for brevity.

3.1.1 ECB—Electronic Code Book

The simplest mode of operation is ECB mode, visually described in Figure 3.1. 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:

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

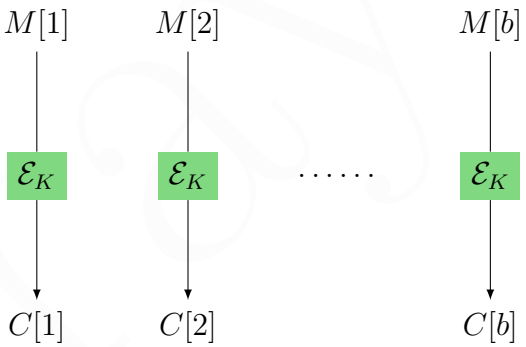


Figure 3.1: The ECB ciphering mode.

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.

3.1.2 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.

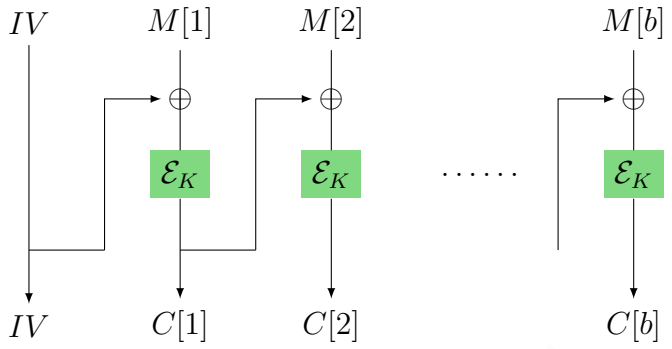


Figure 3.2: The CBC ciphering mode.

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.

3.1.3 CBCC—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:

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

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...

3.1.4 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

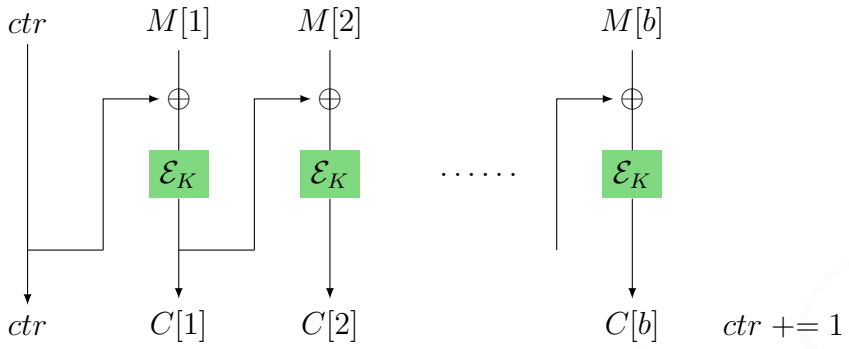


Figure 3.3: The CBCC ciphering mode.

use a block cipher as its fundamental primitive.[1] 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:

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

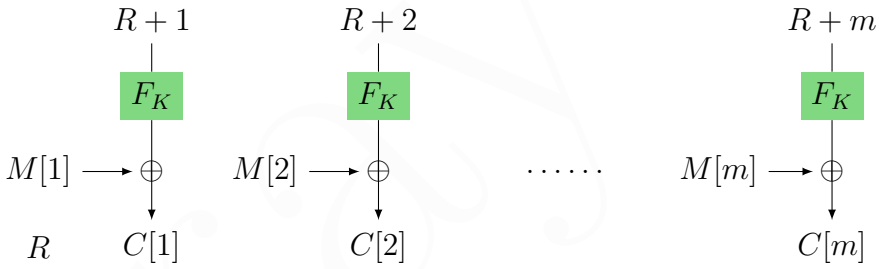


Figure 3.4: The CTR ciphering mode.

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.[2] Notice that to decrypt $C[i]$ in Figure 3.4, one needs to first determine $F_K(R+i)$, then XOR that with the ciphertext to get $M[i]$. 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,

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

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

[1] 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.

[2] 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 $\{0,1\}^n$.

3.1.5 CTRC—Stateful Counter Mode

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

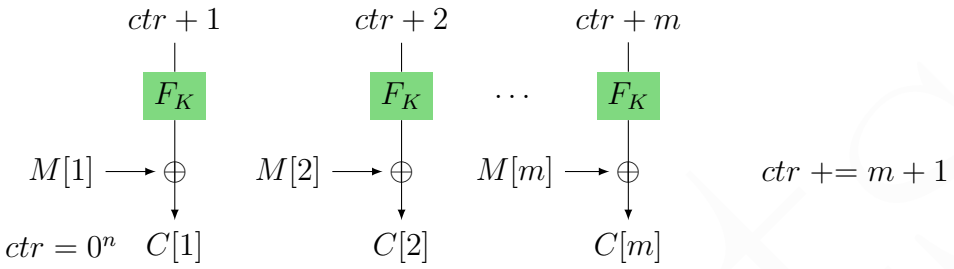


Figure 3.5: The CTRC ciphering mode.

3.2 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*[3] *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.

[3] Any information *except* the length of the plaintexts; this knowledge is assumed to be public.

3.2.1 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, $\mathcal{SE} = (\mathcal{K}\text{eySp}, \mathcal{E}, \mathcal{D})\,; K \xleftarrow{\$} \mathcal{K}\text{eySp}$.

Consider an adversary $\mathcal{A}$ that has access to an oracle. When they provide the oracle with a pair of equal-length messages, $m_0, m_1 \in \mathcal{M}\text{sgSp}$, 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, $C = \mathcal{E}_K(m_b)$.

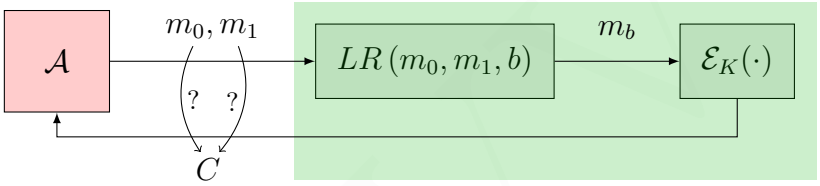


Figure 3.6: A visualization of the IND-CPA adversary scenario.

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 3.2: 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 (≈ 0):

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

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. 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 3.1: 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.

Theorem 3.1. *Any deterministic, stateless encryption scheme cannot be IND-CPA secure.*

Proof. Under deterministic encryption, identical plaintexts result in identical ciphertexts. We can always craft a 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.

The proof holds for an arbitrary $\mathcal{M}$sgSp, we chose $\{0,1\}^n$ in algorithm 3.2 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.

Algorithm 3.2: 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 $m \in \mathcal{M}$sgSp.
$C_3 = \mathcal{E}_K(LR(0^n, 1^n, b))$
if $C_3 = C_1$ **then**
  return *0*
end
return *1*

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 3.3: 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*

3.2.2 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 3.7. 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.

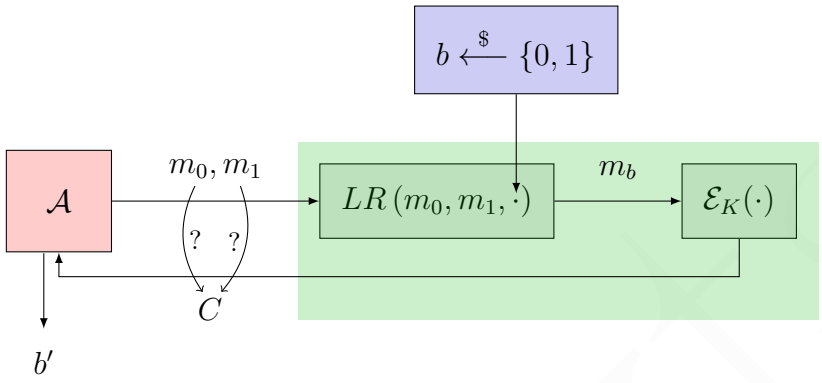


Figure 3.7: The "chosen guess" variant on IND-CPA security, where the attacker must guess a b', and the experiment returns 1 if $b' = b$.

Definition 3.3: 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:

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

The two variants on attacker advantage in Definition 2.3 and the new Definition 2.4 can be proven equal.

Claim 3.1. *$\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{ind\text{-}cpa\text{-}cg}}(\mathcal{A})$ 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 uniformally-random probability.

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

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:

$$\begin{aligned}
\Pr[\text{exp-cg returns 1}] &= \frac{1}{2} + \frac{1}{2}\underbrace{\Pr[b' = 0 \,|\, b = 0] - \Pr[b' = 0 \,|\, b = 1]}_{\text{IND-CPA advantage}} \\
&= \frac{1}{2} + \frac{1}{2}\mathsf{Adv}^{\text{ind-cpa}}(\mathcal{A}) \\
2 \cdot \Pr[\text{exp-cg returns 1}] - 1 &= \mathsf{Adv}^{\text{ind-cpa}}(\mathcal{A}) \\
\mathsf{Adv}^{\text{ind-cpa-cg}}(\mathcal{A}) &= \mathsf{Adv}^{\text{ind-cpa}}(\mathcal{A}) \qquad (3.1)
\end{aligned}$$

■

3.2.3 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 $K \in \mathcal{K}\text{eySp}$ until they find the one that encrypts some known message to a known ciphertext. If we say $k = |\mathcal{K}\text{eySp}|$, 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 3.1, 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.

3.2.4 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:

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

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,[4] we can define g in pseudocode as a deterministic way of picking bitstrings (see algorithm 3.4).

Algorithm 3.4: $g(x)$, a random function.

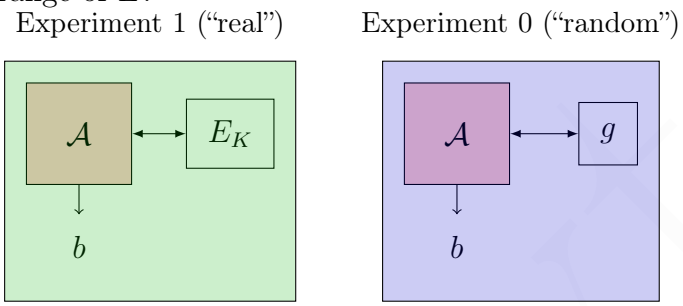
Define a global array T
if $T[x]$ *is not defined* **then**
  $T[x] \xleftarrow{\$} \{0,1\}^L$
end
return $T[x]$

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 3.4 and is hidden from the attacker.

[4] 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 $K \in \mathcal{K}\text{eySp}$; 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 3.4: 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

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

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,

$$\mathsf{Adv}^{\mathsf{prf}}_{\text{AES}}(\mathcal{A}) \leq \underbrace{\frac{ct}{T_{\text{AES}}} \cdot 2^{-128}}_{\text{exhaustive key-search}} + \underbrace{q^2 \cdot 2^{-128}}_{\text{birthday paradox}} \tag{3.2}$$

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 3.4 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, ys 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" (≈ 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

$$\underbrace{\text{CTRC is } \textit{not} \text{ a secure mode of operation}}_{\neg Q}$$
$$\implies \text{only when } \underbrace{\text{the underlying blockcipher is } \textit{not} \text{ secure}}_{\neg P}$$

Theorem 3.2. *CTRC is a secure mode of operation if its underlying block cipher is secure.*

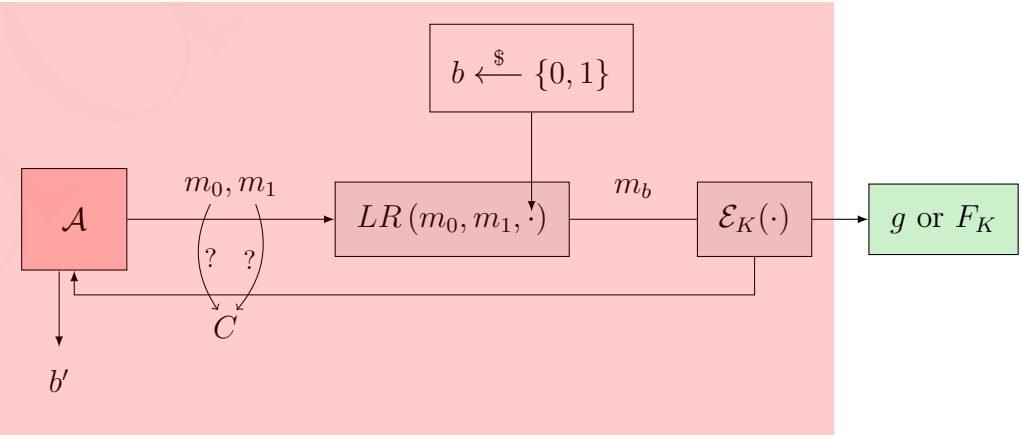
More formally, for any efficient adversary $\mathcal{A}$, $\exists \mathcal{B}$ with similar efficiency such that the IND-CPA advantage of $\mathcal{A}$ under CTRC mode is less than double the PRF advantage of $\mathcal{B}$ under a secure block cipher F:

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

where we know an example of a secure block cipher F =AES that any $\mathcal{B}$'s advantage will be very small (see (3.2)).

Proof. Let $\mathcal{A}$ be an IND-CPA-cg adversary attacking CTRC. 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.



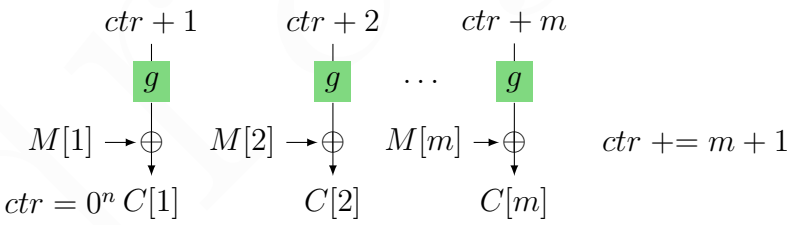
adversary $\mathcal{B}$ ⟶ 1 iff $b' = b$ else 0

- 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 3.5.
- 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 3.5.

Now let's analyze $\mathcal{B}$, expressing its PRF advantage over F in terms of $\mathcal{A}$'s IND-CPA advantage over CTRC.[5] The ability for $\mathcal{B}$ to differentiate between F and some random function $g \in \text{Func}(\ell, L)$ depends *entirely* on $\mathcal{A}$'s ability to differentiate between CTRC with an actual block cipher F and a truly-random function g. Thus,

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{prf}}_{F}(\mathcal{B}) &= \Pr\left[\mathcal{B} \to 1 \text{ in } \mathsf{Exp}^{\mathsf{prf\text{-}0}}_{F}\right] - \Pr\left[\mathcal{B} \to 1 \text{ in } \mathsf{Exp}^{\mathsf{prf\text{-}1}}_{F}\right] && \text{definition} \\
&= \Pr\left[\mathsf{Exp}^{\mathsf{ind\text{-}cpa\text{-}cg}}_{\text{CTRC}[F]} \to 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{ind\text{-}cpa\text{-}cg}}_{\text{CTRC}[g]} \to 1\right] && \mathcal{B} \text{ depends only on } \mathcal{A} \\
&= \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\text{CTRC}[F]}(\mathcal{A}) + \frac{1}{2} - \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\text{CTRC}[g]}(\mathcal{A}) - \frac{1}{2} && \text{IND-CPA is equal to IND-CPA-cg via (3.1)}
\end{aligned}$$

Next, we'll show that $\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\text{CTRC}[g]}(\mathcal{A}) = 0$. 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:

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

[5] The syntax $\mathcal{X} \to n$ means the adversary $\mathcal{X}$ outputs the value n, and the syntax $\mathsf{Exp}^{\mathsf{n}}_{\mathsf{m}}$ refers to the experiment n under some parameter or scheme m, for shorthand.

$$= \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathrm{CTRC}}(\mathcal{A})$$

$$2 \cdot \mathsf{Adv}^{\mathsf{prf}}_{F}(\mathcal{B}) = \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathrm{CTRC}}(\mathcal{A})$$

■

Algorithm 3.5: 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*

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 3.4). It's also provably PRF secure, but we'll state its security level without proof:[6]

Theorem 3.3. *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:*

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

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

[6] 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 Rs and Ms 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).

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 3.2), the last mode of operation whose security we haven't formalized.

Theorem 3.4. *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:*

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

where μ 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 μ. Thus, CTRC is more secure than CBC is more secure than CTR. The constant again comes from the birthday paradox.

3.2.5 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,[7] 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.

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 3.8, this means $C \neq C'$). As before, a scheme is considered IND-CCA secure if an adversary's advantage is small.

[7] 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!

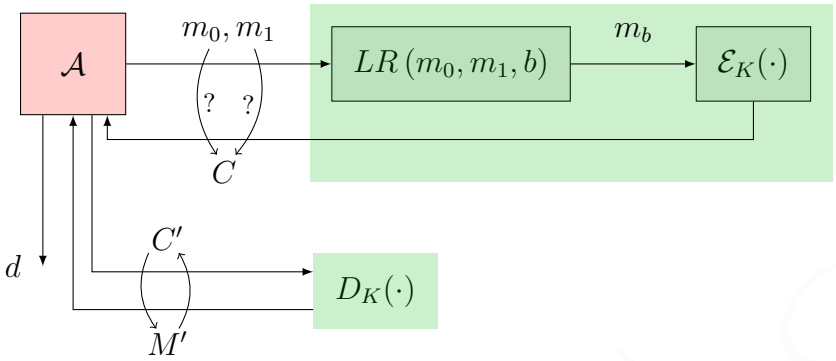


Figure 3.8: A visualization of the IND-CCA security definition. The adversary $\mathcal{A}$ submits two messages, (m_0, m_1), to an encryption oracle that (consistently) chooses one of them based on a bit b and returns m_b's ciphertext. The adversary can also submit any C' that hasn't been submitted to LR to a decryption oracle and see the resulting plaintext.

Definition 3.5: **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 (≈ 0):

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

Note that since IND-CCA is stronger than IND-CCA, 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 3.2 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:

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

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 3.2), 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,

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

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?

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

Well, according to Figure 3.2, 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. ■

3.3 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 CBCC) were not secure under Definition 3.2, IND-CPA security. Then, we dug deeper to study the underlying block ciphers and what it meant for *those* to

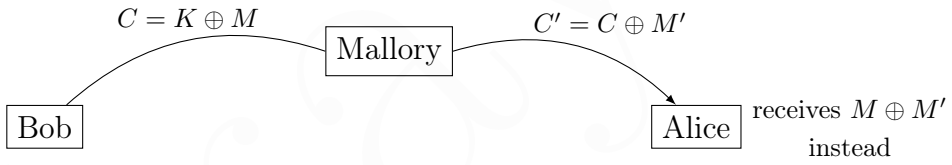
be PRF secure: it must be hard to differentiate them from a random function (see Definition 3.3). Finally, we gave the attacker more power under the last, strictest metric of security: IND-CCA, and showed that our remaining modes of operation (that is, CBC, CTR, and CTRC) broke under this adversarial 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

> *When in doubt, use brute force.*
>
> — Ken Thompson

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).

4.1 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 $\mathcal{M}$sgSp 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, $\mathcal{K}$eySp, describes the set of possible keys for the scheme and how they are created.
- the MAC algorithm itself, $\mathcal{M}_{\text{AC}}$ (also called a **tagging** or **signing algorithm**) defines the way some $m \in \mathcal{M}$sgSp is authenticated and returns a tag.
- the MAC's corresponding **verification algorithm**, $\mathcal{V}_{\text{F}}$, 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 $\Pi = (\mathcal{K}, \mathcal{M}_{\text{AC}}, \mathcal{V}_{\text{F}})$, and by definition

$$\forall k \in \mathcal{K}\text{eySp}, \forall m \in \mathcal{M}\text{sgSp}: \qquad \mathcal{V}_{\text{F}}(k, m, \mathcal{M}_{\text{AC}}(k, m)) = 1$$

If a MAC algorithm is deterministic, then $\mathcal{V}_{\text{F}}$ 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?

4.2 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 `(message, 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.

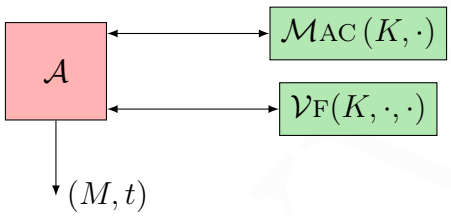
For example, imagine Bob sends an encrypted, authenticated message "You owe my friend Mallory $5." to Alice that everyone can see. Alice knows this message came from Bob, so she pays her dues. Then, Mallory decides to just... send Alice that message again! It's again deemed valid, and Alice

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.

4.2.1 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, $\mathcal{V}_{\mathrm{F}}(K, M, t) = 1$) and M was never an input to the tagging oracle.[1]



Definition 4.1: UF-CMA

A message authentication code scheme Π 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 by the probability of the oracle mistakenly verifying a message:

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

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 4.1).

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

$$\begin{aligned} T_1 = \mathcal{M}_{\mathrm{AC}}(X_1 \parallel Y_1) &\longrightarrow \mathcal{E}_K(1 \parallel X_1) \oplus \mathcal{E}_K(2 \parallel Y_1) \\ T_2 = \mathcal{M}_{\mathrm{AC}}(X_1 \parallel Y_2) &\longrightarrow \mathcal{E}_K(1 \parallel X_1) \oplus \mathcal{E}_K(2 \parallel Y_2) \end{aligned}$$

[1] This lone restriction on the adversary is exactly like the one for IND-CCA, where its trivial to get a perfect advantage if you're allowed to decrypt messages you've encrypted.

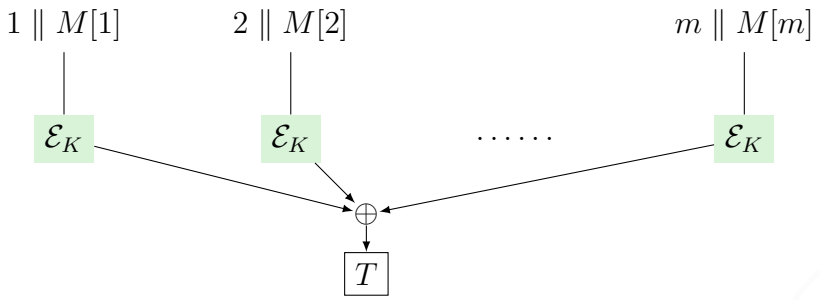


Figure 4.1: A simple MAC algorithm.

$$T_3 = \mathcal{MAC}\,(X_2 \parallel Y_1) \qquad \longrightarrow \qquad \mathcal{E}_K(1 \parallel X_2) \oplus \mathcal{E}_K(2 \parallel Y_1)$$

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

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

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 $\mathcal{MAC}$).

4.3 Mode of Operation: CBC-MAC

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

Theorem 4.1. *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 3.1.2) 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 $\mathcal{M}\text{sgSp} = \{0,1\}^{mn}$, **fixed** m-block messages (obviously $m \geq 1$).

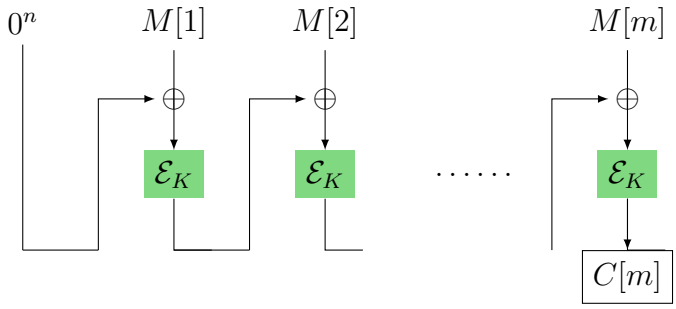


Figure 4.2: The CBC-MAC authentication mode.

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 4.2. *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:

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

(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

> *Realize you won't master data structures until you are working on a real-world problem and discover that a hash is the solution to your performance woes.*
>
> — Robert Love

An essential part of modern cryptography, **hash function**s is transforms 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 5.1. They should be pretty familiar: SHA-1 is used by `git` and SHA-3 is used by various cryptocurrencies like Ethereum.[1] 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 5.1: A list of some modern hash functions and their output digest length.

[1] 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.

5.1 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 uniformally across the output space to minimize *hash collisions*. In fact, most of the functions in Table 5.1 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:

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 5.1: 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 (≈ 0).

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

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 simplicitly, 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:

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

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

5.2 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 5.1. It's used by many modern hash function families, including the MD and SHA families. Visually,

Figure 5.1: A visualization of the Merkle-Damgård transform.

Algorithm 5.1: 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 $i = 1, \ldots, m+1$ **do**
  $V[i] := h\left(M[i] \parallel V[i-1]\right)$
end
return $V[m+1]$

---

it looks like Figure 5.1: each "block" of the input message is concatenated with the hashed version of its previous block, then hashed again.

The good news of this transform is the following:

> **Theorem 5.1.** *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."[2] 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; tthis is 100,000 faster than the birthday attack.

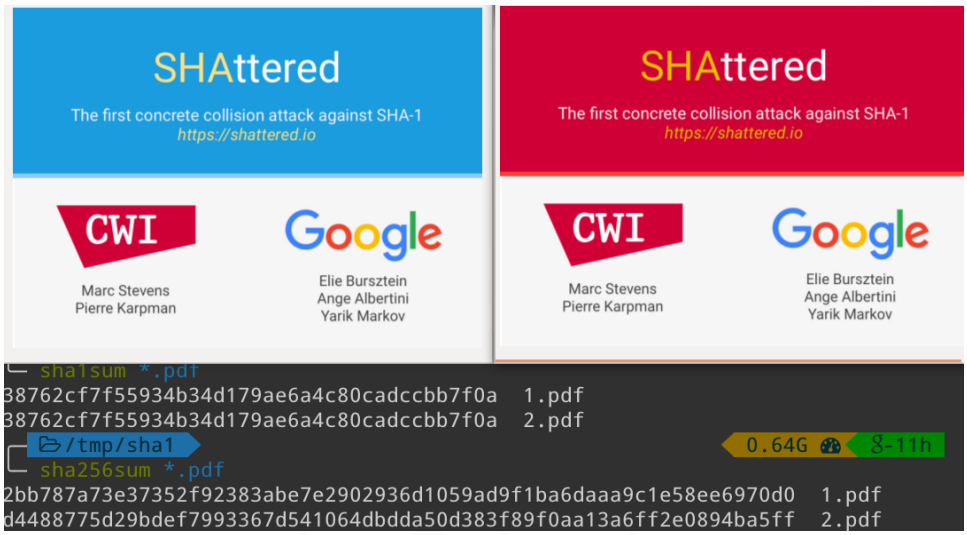


Figure 5.2: The two PDFs in the SHAttered attack and their resulting, identical digests. More details are available on the attack's site (because no security attack is complete without a trendy title and domain name).

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,

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

[2] 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.

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 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.

5.3 One-Way Functions

Hash functions that are viable for cryptographic use must being **one-way function**s: 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).

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

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 5.2: **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 (≈ 0).

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

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 $\implies$ 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

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

5.4 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 5.3.

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[3] The HMAC is computed using a nested structure, mixing the key

[3] Typically, $B = 64$ for modern hash functions like MD5, SHA-1, SHA-256, and SHA-512.

with some constants. Namely, we define

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

where opad and ipad are hex-encoded constants:

$$\text{opad} = \texttt{0x}\underbrace{\texttt{5C5C5C}\ldots}_{\text{repeated } B \text{ times}} \qquad \text{ipad} = \texttt{0x}\underbrace{\texttt{363636}\ldots}_{\text{repeated } B \text{ times}}$$

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

$$\mathsf{Hmac}_K(K) = H\left(K_o \parallel H(K_i \parallel M)\right)$$

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

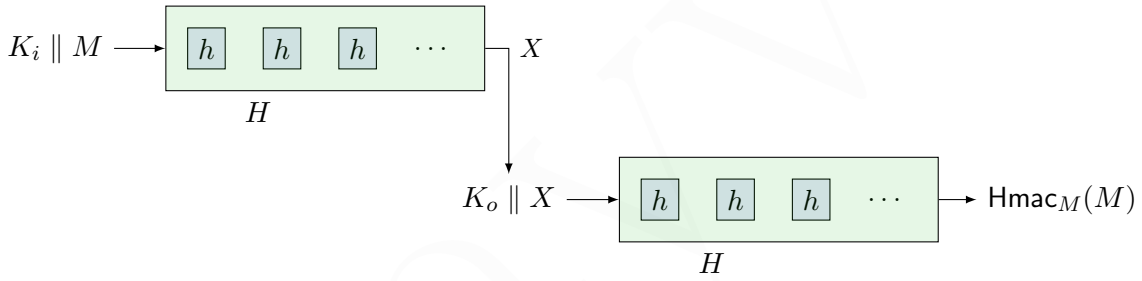


Figure 5.3: A visualization of the two-tiered structure of HMAC, the standard keyed message authentication code scheme.

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 5.2. *HMAC is a PRF assuming that the underlying compression function H is a PRF.*

Authenticated Encryption

> *If privacy is outlawed, only outlaws will have privacy.*
>
> — Philip Zimmermann

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 4.1 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.

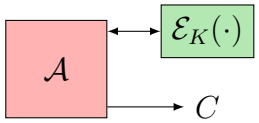
6.1 INT-CTXT: Integrity of Ciphertexts

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 6.1: INT-CTXT Security

A scheme $\mathcal{SE} = (\mathcal{K}\text{eySp}, \mathcal{E}, \mathcal{D})$ is considered secure under INT-CTXT if an adversary's **INT-CTXT advantage**—the probability of producing a valid, forged ciphertext—is small (≈ 0):

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



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 6.1. *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

6.2 Generic Composite Schemes

Given a symmetric encryption scheme and a message authentication code, we can combine them in a number of ways: Wikipedia

- **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 5.1), 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 4.1 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.[1]

[1] 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.

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

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

6.2.1 Encrypt-and-MAC

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

Algorithm 6.1: 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 6.2: 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 $\perp$

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 3.1).[2]

6.2.2 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.

[2] 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.

Algorithm 6.3: 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 6.4: 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:

$$\begin{aligned} \mathcal{SE}' &= \{\mathcal{K}', \mathcal{E}', \mathcal{D}'\} \\ \mathcal{E}'_K(M) &= \mathcal{E}''_K(M) \parallel 0 \\ \mathcal{D}'_K(M \parallel b) &= \mathcal{D}''_K(M) \end{aligned}$$

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*.

6.2.3 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.

> **Caveat: Timing Attacks**
>
> Notice the key, extremely important nuance of the decryption routine in algorithm 6.6: 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 6.1, also under IND-CCA.

Theorem 6.2. *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 3.4) and HMAC-SHA-3 (which is conjectured to be a cryptographically-secure hash function).

Algorithm 6.5: 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 6.6: 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 $\perp$

6.2.4 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.

6.2.5 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,

Protocol	Composition Method	In general...	In this case...
SSH[3]	Encrypt-and-MAC	Insecure	Secure
SSL	MAC-then-encrypt	Insecure	Secure
IPSec	Encrypt-then-MAC	Secure	Secure
WinZip	Encrypt-then-MAC	Secure	**Insecure**

Table 6.1: 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)

heavily parallelizable scheme.[4]

6.3 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.

6.3.1 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.

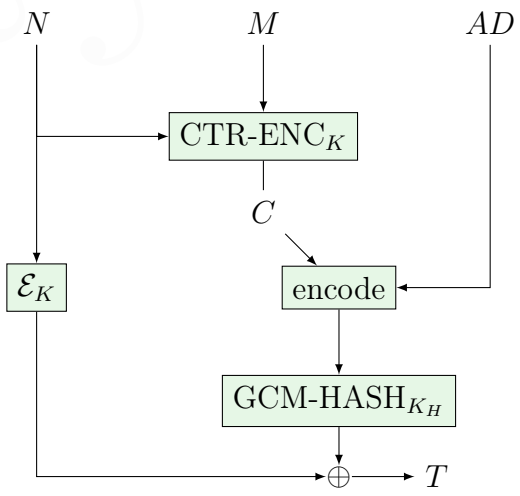


Figure 6.1: A visualization of the Galois/-Counter mode (GCM) of AEAD encryption.

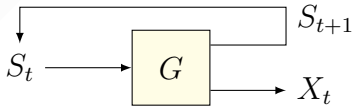
[4] It was designed by Phillip Rogaway, one of the authors for the lecture notes on cryptography.

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 infintely-long one-time pad, and a stream cipher will use that output to encrypt plaintexts.

7.1 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.

7.1.1 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**.

7.2 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$^{\text{th}}$ 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 7.1: **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:
>
> $$\mathsf{Adv}_G^{\mathsf{indr}}(\mathcal{A}) = \Pr\left[b' = 1 \text{ for } \mathsf{Exp}_1\right] - \Pr\left[b' = 1 \text{ for } \mathsf{Exp}_0\right]$$

7.3 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 7.1 (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 7.1: $G(S_t)$, a PRG based on the CTR mode of operation.

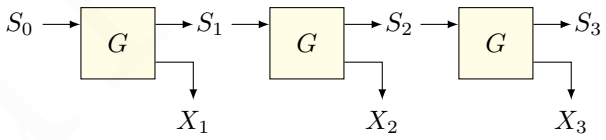
Input: *S_t, the current PRG input.*
Result: *(X, S_{t+1}), the pseudorandom value and the new PRG state.*

$K \parallel V = S_t$ `/* 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.

7.3.1 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 7.1, 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 7.2: $G(K)$, 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)

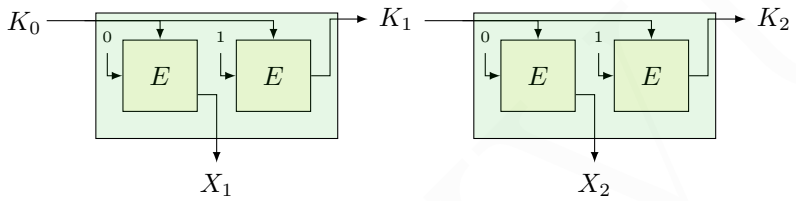


Figure 7.1: A visualization of a PRG with forward secrecy.

> 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-exposured vulnerability interval to certain (short) time periods.

7.3.2 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

> *Nowadays most people die of a sort of creeping common sense, and discover when it is too late that the only things one never regrets are one's mistakes.*
>
> — Oscar Wilde, *The Picture of Dorian Gray*

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 *in*security—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 3.4) had the following adversarial advantage:

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

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[1] 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.

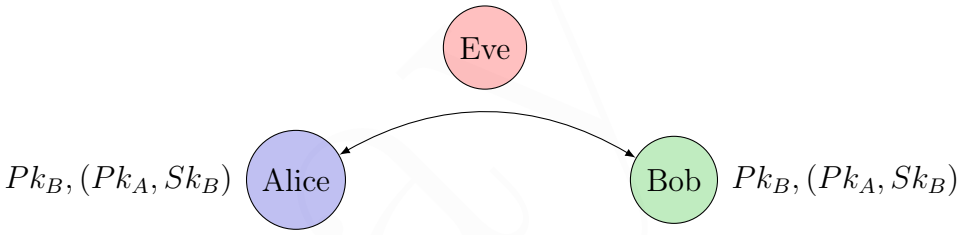
Security Proofs As we've seen, we often need to extend our security definitions to encompass more sophisticated attacks (like IND-CCA over IND-CPA). Thus, even using a provably-secure scheme does not absolve you of an attack surface. For example, the Lucky 13 attack used a side-channel timing attack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

[1] 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.

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 **private key** that must is 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.

The big assumption 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.

Contents

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.

9.1 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 $\mathcal{AE} = (\mathcal{E}, \mathcal{D}, \mathcal{K})$.

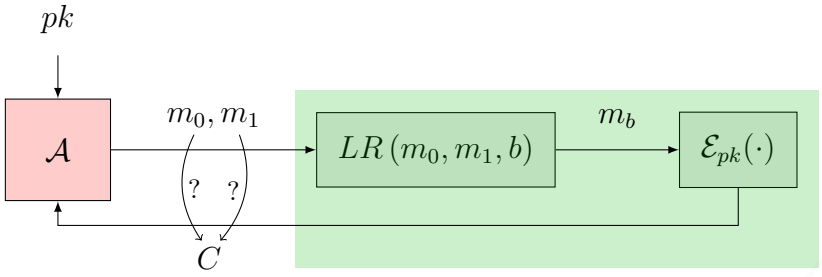
The key is now broken into two components: the **public key** (shareable) and the **private key** (secret). These are typically composed as: $K = (pk, sk)$.

9.2 Security Definitions

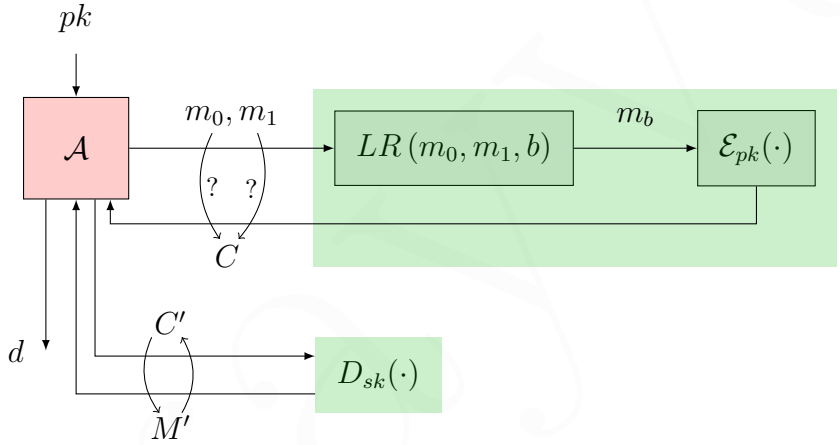
Our definitions of IND-CPA and IND-CCA security will be very similar to the way they were defined previously; the main difference is the (obvious) introduction of the private- and public-key split, as well as the fact that we can be more precise about what "reasonable" attacker resources are. Just to 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 3.6 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 6.1).

Theorem 9.1. *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:*

$$\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{AE}}(\mathcal{A}) \leq q \cdot \mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{AE}}(\mathcal{A}')$$

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.

> These notes are partially ripped from my notes for *Graduate Algorithms*, which covers number theory briefly while building up to the RSA algorithm, so not everything aligns 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, $\{0, 1, \ldots\}$.
- $\mathbb{Z}_N$ is the set of positive integers up to N: $\{0, 1, \ldots, N-1\}$.
- $\mathbb{Z}_N^*$ 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.

10.1 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 number $1 \in \mathbb{Z}^+$.
- **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 10.1. *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,
$$\forall a \in G, i \in \mathbb{Z} : a^i = a^{i \bmod m}$$

Example These facts 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:
$$\begin{aligned} 5^{86} \bmod 21 &= \left(5^{86 \bmod 12}\right) \bmod 21 \\ &= 5^2 \bmod 21 = 25 \bmod 21 \\ &= \boxed{4} \end{aligned}$$

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: $|G| \bmod |S| = 0$.

10.2 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:

$$\ldots, -6, -3, 0, 3, 6, \ldots$$
$$\ldots, -5, -2, 1, 4, 7, \ldots$$
$$\ldots, -4, -1, 2, 5, 8, \ldots$$

10.2.1 Running Time

Under modular arithmetic, there are different time complexities for common operations. The "cheat sheet" in Table 10.1 will be a useful reference for computing the overall running time of various asymmetric cryptography schemes.

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

Table 10.1: 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 10.1: Negligibility

A function $g : \mathbb{Z}^+ \mapsto \mathbb{R}$ is **negligible** if it vanishes faster than the reciprocal of any polynomial. Namely, for every $c \in Z^+$, 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.

10.2.2 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, 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 x and y are the greatest common divisors of n, 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:

$$\begin{aligned} ax + bn &= 1 \\ ax + bn &\equiv 1 \pmod{n} && \text{taking mod}n \text{ of both sides doesn't change the truth} \\ ax &\equiv 1 \pmod{n} && bn \bmod n = 0 \end{aligned}$$

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

10.2.3 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 $x \cdot x \cdot \ldots$ or even $((x \cdot x) \bmod N) \cdot x) \bmod N) \ldots$, we leverage repeated squaring:

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

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

10.3 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:

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

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.

10.3.1 Discrete Logarithm

If $G = \langle g \rangle$ is cyclic, then for every $a \in G$, there is a **unique** exponent $i \in \{0, \ldots, |G|-1\}$ such that $g^i = a$. We call i the **discrete log**arithm 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 $\mathcal{O}\left(\sqrt{|G|}\right)$ at best. **There are no polynomial time algorithms for computing the discrete logarithm.** This isn't proven, but much like the AES conjecture (see (3.2)), 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:
>
> $$\mathcal{O}\left(e^{1.92(\ln p)^{1/3}(\ln \ln p)^{2/3}}\right) \tag{10.1}$$
>
> If we have a prime-order group over an elliptic curve, the best-known algorithm is $\mathcal{O}\left(\sqrt{p}\right)$, 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.

10.3.2 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 two 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.

Furthermore, 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.[1] Here, though, they're useful because $\left|\mathbb{Z}_p^*\right|$ factors into $(2, q)$ for safe primes.

Property 10.2. *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 $g^q \neq 1$.*

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 witness**es, 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

[1] 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.

$\gcd(z, r) = 1$.

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 10.3. *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: $z \xleftarrow{\$} \{1, 2, \ldots, r-1\}$.
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 10.4. *Given a prime number p, the number 1 only has the trivial square roots ± 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 $3/4^{\text{ths}}$ of the choices of z, which we can use in the same way as before to check primality to a particular degree of certainty.

10.4 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:

$$5^2 = 25 \equiv 3 \pmod{11}$$

$$6^2 = 36 \equiv 3 \pmod{11}$$

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[2] under the modulus:

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

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

10.4.1 Square Groups

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

$$J_p(a) = \begin{cases} 1 & \text{if } a \text{ is a square mod } p,\\ 0 & \text{if } a \bmod p = 0,\\ -1 & \text{otherwise} \end{cases} \tag{10.2}$$

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

$$\begin{aligned}
\mathrm{QR}\left(\mathbb{Z}_p^*\right) &= \{a \in \mathbb{Z}_p^* : J_p(a) = 1\}\\
&= \{a \in \mathbb{Z}_p^* : \exists b \text{ such that } b^2 \equiv a \pmod{p}\}
\end{aligned} \tag{10.3}$$

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

[2] 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.

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 10.5. *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 (10.2); conveniently, it can actually be computed for any prime $p \geq 3$:

$$J_p(a) \equiv a^{\frac{p-1}{2}} \pmod{p} \tag{10.4}$$

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 10.6. *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 $\left|\text{QR}(\mathbb{Z}_p^*)\right| = \frac{p-1}{2}$.

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 (yet?) to the lecture content.

10.4.2 Square Root Extraction

A secondary, key fact of square-root extraction is the following: if square roots exists 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:

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

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:[3]

$$\begin{aligned}
5^2 &= 25 \equiv 25 && \pmod{143} \\
60^2 &= 3600 \equiv 25 && \pmod{143} \\
83^2 &= 6889 \equiv 25 && \pmod{143} \\
138^2 &= 19004 \equiv 25 && \pmod{143}
\end{aligned}$$

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 $5, 6 \pmod{11}$ and $5, 8 \pmod{13}$. 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,

[3] These were found with a simple Python generator:

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

we find r_i for each pair of roots:

$$\begin{aligned} r_1 &\equiv 5 \pmod{11} \\ r_1 &\equiv 5 \pmod{13} \end{aligned} \qquad \begin{aligned} r_2 &\equiv 5 \pmod{11} \\ r_2 &\equiv 8 \pmod{13} \end{aligned}$$

$$\begin{aligned} r_3 &\equiv 6 \pmod{11} \\ r_3 &\equiv 5 \pmod{13} \end{aligned} \qquad \begin{aligned} r_4 &\equiv 6 \pmod{11} \\ r_4 &\equiv 8 \pmod{13} \end{aligned}$$

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

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

10.5 Chinese Remainder Theorem

The **Chinese remainder theorem** can be used to break RSA when the exponents are too small. The theorem states that if you have a series of coprime values: $n_1, n_2, \ldots, n_k$, then there exists a single value $x \bmod (n_1 n_2 \cdots n_k)$ that is equivalent to a series of values under each of those moduli:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\cdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Asymmetric Schemes

> *How long do you want these messages to remain secret? I want them to remain secret for as long as men are capable of evil.*
>
> — Neal Stephenson, *Cryptonomicon*

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 out 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.

11.1 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 $z \stackrel{?}{\equiv} xy \pmod{|G|}$.

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:

11.1.1 Formalization

In each case, suppose 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:

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

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

> DEFINITION 11.1: **Discrete-Log Advanatage**
>
> We can define the **dl-advantage** of an adversary as the probability of winning the discrete logarithm experiment:
>
> $$\mathsf{Adv}^{\mathsf{dl}}_{G,\,g}(\mathcal{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{dl}}_{G,\,g}(\mathcal{A}) \text{ wins}\right]$$

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:

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

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):

$$\mathsf{Exp}^{\text{ddh-1}}_{G,g}(\mathcal{A}): \quad x, y \xleftarrow{\$} \mathbb{Z}_m \quad z = xy \bmod m \quad d = \mathcal{A}(g^x, g^y, g^z) \quad \text{return } d$$

$$\mathsf{Exp}^{\text{ddh-0}}_{G,g}(\mathcal{A}): \quad x, y \xleftarrow{\$} \mathbb{Z}_m \quad z \xleftarrow{\$} \mathbb{Z}_m \quad \text{key diff.} \quad d = \mathcal{A}(g^x, g^y, g^z) \quad \text{return } d$$

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

Definition 11.2: DDH Advantage

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

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

11.1.2 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 (10.1).

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

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

(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.

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 10.5). 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 10.4 or more specifically Property 10.6: the Legendre symbol of an exponent product must match the individual exponents.

Algorithm 11.1: 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 10.5—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:

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

The algorithm only needs two modular exponentiations, meaning it takes $\mathcal{O}(|p|^3)$ time (refer to Table 10.1) 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 believe to be difficult in $\mathrm{QR}(\mathbb{Z}_p^*)$ where $p = 2q + 1$, a safe prime.

11.2 Diffie-Hellman Key Exchange

11.3 RSA Algorithm

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.

11.3.1 Fermat's Little Theorem

This is the basis of the RSA algorithm we're about to see and it's just a specific case of the generic exponentiation facts we showed when covering Groups.

Theorem 11.1 (Fermat's little theorem)**.** *If p is any prime, then*

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

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

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:

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

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.

Unfortunately, you need to reveal p to do this, which reveals $p-1$ and lets someone derive de. We'll hide this by using $N = pq$ and **Euler's theorem**, which generalizes Fermat's little theorem and relates the totient function to multiplicative inverses.

Theorem 11.2 (Euler's theorem)**.** *For any N, a where $\gcd(a, N) = 1$ (that is, they are relatively prime), then*

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

Let $N = pq$ be the product of two prime numbers. Then, $\varphi(N) = (p-1)(q-1)$, so Euler's theorem tells us that:

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

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

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

That's RSA.

A user reveals some information to the world: their public key, e and their 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, d:

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

This is secure because you cannot determine $(p-1)(q-1)$ from the revealed N and e without exhaustively enumerating all possibilities (i.e. "factoring is hard"); thus, if p and q are large enough, it's computationally infeasible to factor N.

11.3.2 Protocol

With the theory out of the way, here's the full protocol.

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 by enumerating the low primes and finding their GCD.
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.

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 10.2.3).

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

11.3.3 Limitations

For this to work, the message must be small: $m < N$. This is why asymmetric cryptography is typically only used to exchange a secret **symmetric** key which is then used for all other future messages.

We also 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 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!

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. Let $e = 3$, then 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 this is just m^3 "unrolled" without the modulus! Thus, $m = \sqrt[3]{m^3}$, and finding m^3 can be done quickly with the extended Euclidean algorithm.

PART III

Advanced Topics

This part of the guide is dedicated to advanced topics in cryptography that I've been researching myself recently out of interest. It has no relation to the official course content (unless we cover the topic later, that is). It's worth noting that $\log n$ in this section refers to the discrete logarithm under base 2, so $\log 8 = \log_2 8 = 3$.

Contents

Zero-Knowledge Proofs

The beauty of a **zero-knowledge proof** (or ZKP) is that one party can prove something to another without revealing any concrete information about it.

> This is a complicated concept from a mathematical standpoint, so this section will have a *lot* of examples gathered from all over the Internet to entice the development of an intuition about the topic.

12.1 Knowledge Proof

First, it's worth covering the basic concept of a "knowledge proof." In a proof of knowledge, Alice is a **prover** aiming to demonstrate a solution to a problem, while Bob is a **verifier** that can validate whether or not this is a valid solution.

The fundamental underpinning of this method is to use a problem that is difficult to solve "on-the-fly." Typically, we use something from the NP (non-polynomial) class of problems; by definition, these problems are computationally-expensive to solve. The prover claims to have solved a particular NP problem; then, the verifier chooses inputs to the problem at random. The key is that the prover can't think of a valid solution on the spot without having truly actually solved the problem already.

In other words: **the "knowledge" must be hard to compute but easy to verify.** This is what prevents "cheating" on Alice's part.

12.1.1 Example: Factorization

Finding the factorization of a number is considered "hard" (this is the very basis of asymmetric cryptography). Thus, an example of a knowledge proof could be Alice claiming to know the factorization (p, q) of some number $n = pq$. How can Alice convince Bob that she really does know the factorization?

Correlated with factorization is an equally-hard problem: Modular Square Roots. The key is that Bob provides numbers to find roots of and Alice returns their roots. Alice wouldn't be able to find the roots of arbitrary values (which is NP) efficiently

without knowing the factorization (which makes it P).

The example from Modular Square Roots used small primes that are easy to brute force, but it'll suit our purpose for demonstration since all we need to do to make it computationally-infeasible is use bigger primes.

Alice, the prover, claims to Bob to know a factorization for $n = 143$. She knows that $p = 13, q = 11$. Bob, as the verifier, submits to Alice the following test:

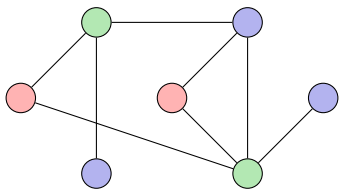
> *If you know some $pq = n$, what are the roots of 25, 100, 1000, and 2000?*

Alice must compute these quicker than Bob can brute-force them. Alice calculates the partial roots of each prime using brute force *on the factors* (not their product), then of n via the Chinese remainder theorem. This can be done in polynomial time, whereas brute force takes exponential time in the number of digits in n. Bob can validate the roots trivially.

Bob can then test Alice again, over and over until he's satisfied of her knowledge. More formally, though there may some small chance that Alice can cheat and fake her knowledge, with enough trials this chance gets closer and closer to zero. At no point in time does Alice actually reveal p or q, yet Bob is satisfied that Alice knows them.

12.1.2 Example: Graph Coloring

The k-color problem is well known in computer science. Given a graph, it's considered difficult to find a way to label each node with one of k colors such that no neighboring nodes share a color. For example, the following is a valid 3-coloring of the graph:



Finding the coloring is hard, but verifying it is easy. However, it seems like Alice would need to reveal the entire graph coloring to prove to Bob that it's valid, no? Fortunately not. Because we are making probabilistic guarantees, we can use a scheme that reveals nothing about the true coloring of the graph yet also guarantees its validity.

Bob presents a simple challenge: "Show me the colorings of edge e." Alice then reveals the colors she chose for that edge. Note that she doesn't *respond* with some pair of colors (since she could just reply with two distinct colors every time), but instead

reveals the colors of the edge.[1] What's the probability of Alice lying about having a valid coloring? Well, in the worst case, Alice's graph coloring was perfectly colored except for a single edge. The chance that Bob did not choose this invalid edge is pretty high: $\frac{E-1}{E}$, where E is the number of edges in the graph. For the above 8-edge graph, there's an 87.5% chance that Alice is full of it.

How does Bob get more certainty? Run it again! Now, critically, Alice *shuffles the colors she used* before presenting the colored graph; notice that the validity of the above coloring does not change if we swap reds and greens, for example. We do the challenge and reveal again, and this time there's a $\left(\frac{E-1}{E}\right)^2$ chance of cheating (a 76.6% chance for our example). That's progress... In general, the probability of cheating after n runs is $\left(\frac{E-1}{E}\right)^n$, so if we wanted, for example, 99% *certainty* that Alice isn't lying, we can solve for n:

$$\begin{aligned}
\left(\frac{E-1}{E}\right)^n &< 0.01 \\
\left(\frac{7}{8}\right)^n &< 0.01 \\
\left(\frac{7}{8}\right)^n &< 0.01 \\
\log_{7/8} 0.01 &\geq n \\
34.5 &\gtrapprox n
\end{aligned}$$

So after 35 rounds (shuffle, challenge, reveal), Bob can be pretty dang sure that Alice isn't lying about her graph coloring, and since Alice always shuffled and did not reveal a full coloring, the full solution is still a secret.

12.1.3 Formalization

Knowledge proofs are pretty easy to define formally. Let's refer to a prover, Alice, that actually has knowledge A^* and prover-Alice that is faking the knowledge as $\overline{A}$. In general, the prover is A and can be either Alice. Our verifier, Bob, will just be B.

The "problem" in question is the predicate $P(Q, S)$, where Q is a challenge query proposed by B and S is the solution output by A.

Formally, then, a verifier should act as follows:

1. If $A = A^*$, B should accept its proofs with *overwhelming* probability for all q for which $P(q, S)$ is satisfiable.

[1] Cryptographically-speaking, this sort of "secret reveal" might be done by telling Bob the keyed hash of each node's color, then revealing the keys for the specified edge. This is a nuance of the protocol itself and bears its own discussion, but the main point is that the graph colors are fixed-yet-unknown when Bob presents the challenge.

2. If $A = \overline{A}$, B should accept its proofs with *negligible* probability for all q.

12.2 Zero-Knowledge

A knowledge proof might reveal details about what Alice knows. In both of the above examples, though, we took care not to reveal any secret information (the (p, q) factorization in subsection 12.1.1 and the full graph coloring in subsection 12.1.2), so these were actually **zero**-knowledge proofs! In general, Bob should learn nothing *at all* from Alice about her "truth" claim in a ZKP.

12.2.1 Formalization

Formally, a zero-knowledge proof must satisfy the following key properties:

- **Completeness**: For all *valid* queries or challenges to the problem that the verifier Bob can present, Alice must respond with a valid solution. Colloquially, if Alice is telling the truth about having a solution, they will convince Bob with a high probability.
- **Soundness**: For all *invalid* queries, the verifier must reject Alice's solution with a $\geq \frac{1}{2}$ probability. Colloquially, lying should be near-impossible: Alice can only convince Bob if she really does have a solution.
- **Perfection** (also called *zero-knowledge-ness* [no joke]): For all verifiers V^*, there exists a simulator S^* that is a randomized polynomial time algorithm such that for all $x \in L$.

  $$\{\text{transcript}((P, V^*)(x))\} = \{S^*(x)\}$$

  (the existence of a simulator implies that if $x \in L$, then V^* cannot learn more than the fact that $x \in L$)

  Colloquially, Bob shouldn't learn *any*thing about Alice's solution.

12.3 References

1. Feige, U., Fiat, A., and Shamir, A. Zero-knowledge proofs of identity. *Journal of Cryptology* (1988), 77–94
2. Boneh, D. Zero-knowledge proofs. In *Notes on Cryptography*. Stanford University, 2002
3. Cossack Labs. Explain like I'm 5: Zero knowledge proof, October 2017
4. Ray, S. What are zero knowledge proofs? In *Towards Data Science*. Medium, April 2019
5. Green, M. Zero knowledge proofs: An illustrated primer. In *A Cryptographic Engineering*. November 2014

Index of Terms