Perfect Security

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

0.1 Notation & Syntax

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

A well-described symmetric encryption scheme covers the following:

- a **message space**, denoted as the $\mathcal{M}\text{sgSp}$ 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, $\mathcal{K}\text{eySp}$, 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 $K \overset{\$}{\longleftarrow} \mathcal{K}\text{eySp}$. 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.

0.2 One-Time Pads

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

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

- the key space is all n-bit strings: $\mathcal{K}\text{eySp} = \{0,1\}^n$
- 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 0.1: Shannon-secure

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

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

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:

$$\begin{aligned}&\forall m \in \mathcal{M}\text{sgSp}, \forall C : \\ &\quad \Pr\left[M = m \,|\, C\right] = \Pr\left[M = m\right]\end{aligned}$$

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

0.2.1 The Beauty of XOR

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

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

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

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

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

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

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

0.2.2 Proving Security

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

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

Proof. We start by fixing an arbitrary n-bit ciphertext: $C \in \{0,1\}^n$. We also choose a fixed n-bit message, $m \in \mathcal{M}\text{sgSp}$. 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

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

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 $1/2$:

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

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 0.2. *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 1.1: Block Cipher

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

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

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

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

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

Math Review: Functions

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

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

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

1.1 Modes of Operation

Block ciphers are limited to encrypting an n-bit string, but we want to be able to encrypt arbitrary-length strings. A **mode of operation** is a way to combine block ciphers to achieve this goal. For 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.

1.1.1 ECB—Electronic Code Book

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

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

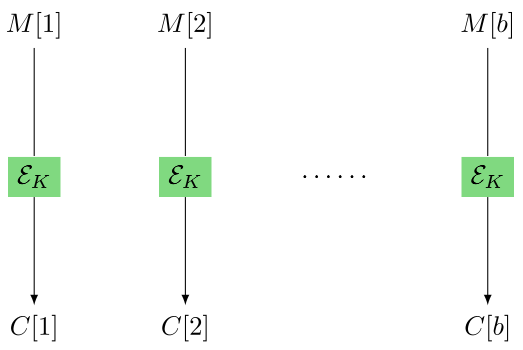


Figure 1.1: The ECB ciphering mode.

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

1.1.2 CBC—Cipher-Block Chaining

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

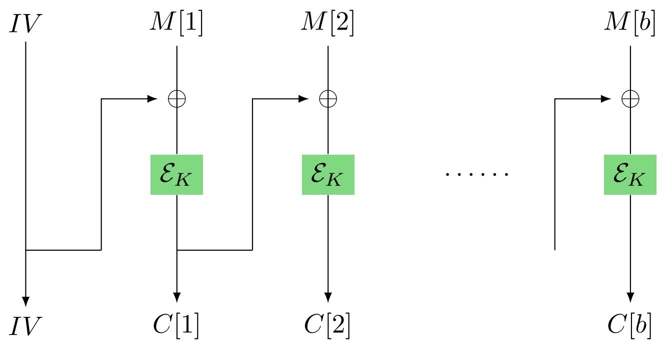


Figure 1.2: The CBC ciphering mode.

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

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

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

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

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

1.1.3 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:

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

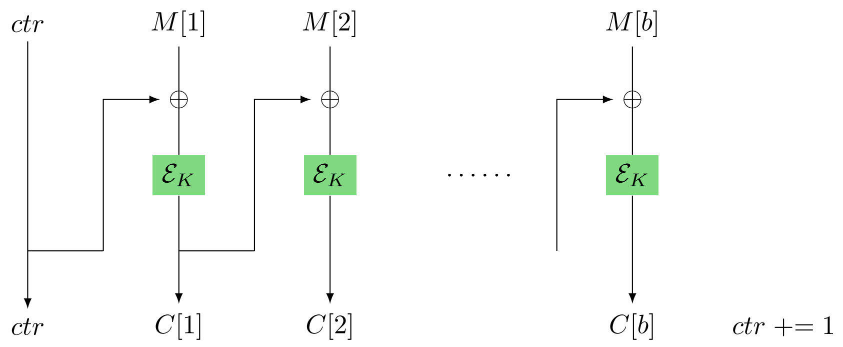


Figure 1.3: The CBCC ciphering mode.

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

1.1.4 CTR—Randomized Counter Mode

Like ECB mode, this encryption mode encrypts each block independently, meaning it can be parallelized. Unlike all of the modes we've seen so far, though, it does not use a block cipher as its fundamental primitive.[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:

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

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 1.4, 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,

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

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

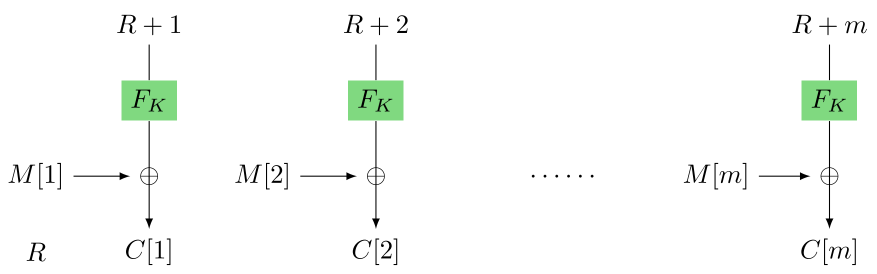


Figure 1.4: The CTR ciphering mode.

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

1.1.5 CTRC—Stateful Counter Mode

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

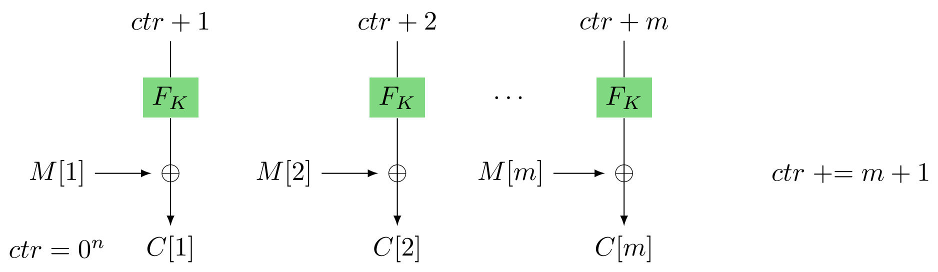


Figure 1.5: The CTRC ciphering mode.

1.2 Security Evaluation

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

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

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

> *Informally, an encryption scheme is secure if no adversary with "reasonable" resources who sees several ciphertexts can compute any*[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.

1.2.1 IND-CPA: Indistinguishability Under Chosen-Plaintext Attacks

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

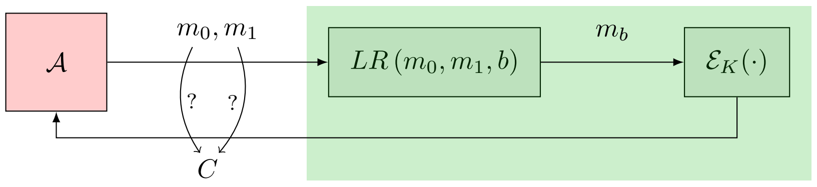
[3] Any information *except* the length of the plaintexts; this knowledge is assumed to be public.

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, $C = \mathcal{E}_K(m_b)$.



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

Definition 1.2: IND-CPA

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

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

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.

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

Theorem 1.1. *Any deterministic, stateless encryption scheme cannot be IND-CPA secure.*

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

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

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.

Algorithm 1.2: A generic algorithm for breaking deterministic encryption schemes.

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

The proof holds for an arbitrary $\mathcal{M}$sgSp, we chose $\{0, 1\}^n$ in algorithm 1.2 for convenience of representation. □

Analysis of CBCC

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

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

Algorithm 1.3: A simple adversarial algorithm to break CBCC mode.

```
// First, determine the counter (can't count on $ctr = 0$).
$C_0 \parallel C_1 = \mathcal{E}_K(LR(0^n, 1^n, b))$
// Craft a message that'll be all-zeros post-counter.
$M_1 = 0^n \oplus (ctr + 1)$
$C_2 \parallel C_3 = \mathcal{E}_K(LR(M_1, 1^n, b))$
// Craft it again, then compare equality.
$M_3 = 0^n \oplus (ctr + 2)$
$C_4 \parallel C_5 = \mathcal{E}_K(LR(M_3, 1^n, b))$
if $C_3 = C_5$ then
    return 0
end
return 1
```

1.2.2 IND-CPA-cg: A Chosen Guess

It turns out that the other modes of operation are provably IND-CPA secure *if* the underlying block cipher is secure. Before we dive into those proofs, though, let's define an alternative interpretation of the IND-CPA advantage; we will call this formulation **IND-CPA-cg**, for "chosen guess," and its shown visually in Figure 1.6. This formulation will be more convenient to use in some proofs.

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

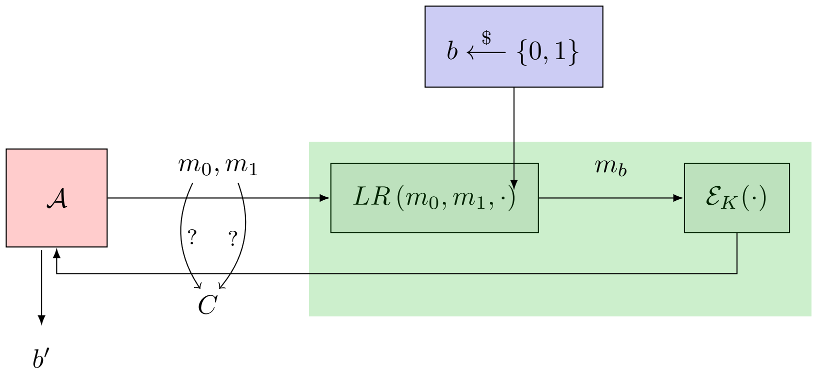


Figure 1.6: The "chosen guess" variant on IND-CPA security, where the attacker must guess a b', and the experiment returns 1 if $b' = b$.

Definition 1.3: IND-CPA-cg

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

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

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

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

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

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

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

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

□

1.2.3 What Makes Block Ciphers Secure?

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

First off, it's important to recall that we expect attackers to be computationally-bounded to a reasonable degree. This is because block ciphers—and all symmetric encryption schemes, for that matter—are susceptible to an **exhaustive key-search** attack, in which an attacker enumerates every possible $K \in \mathcal{K}\text{eySp}$ 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 $O(k)$ time and requires on average 2^{k-1} checks, which is why k must be large enough for this to be infeasible.

Fun Fact: Historical Key Sizes

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

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

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

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

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

1.2.4 Random Functions

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

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

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 1.4).

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

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

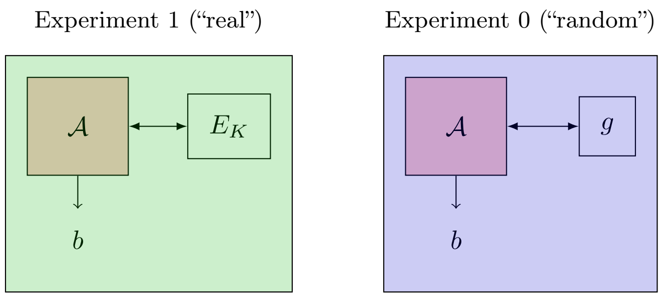
A function family is a **pseudorandom function** family (a PRF) if the input-output behavior of a random instance of the family is computationally indistinguishable from a truly-random function. This input-output behavior is defined by algorithm 1.4 and is hidden from the attacker.

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.

[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.



Definition 1.4: Block Cipher Security

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

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

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,

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

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

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

Fun Fact: The Birthday Paradox

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

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

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

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

If a scheme $\mathcal{T}$ is y-secure,

then a scheme $\mathcal{S}$ is x-secure.

(for some fill-in-the-blank x, 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:

$$\text{if } p \implies q$$
$$\text{then } \neg q \implies \neg p$$

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

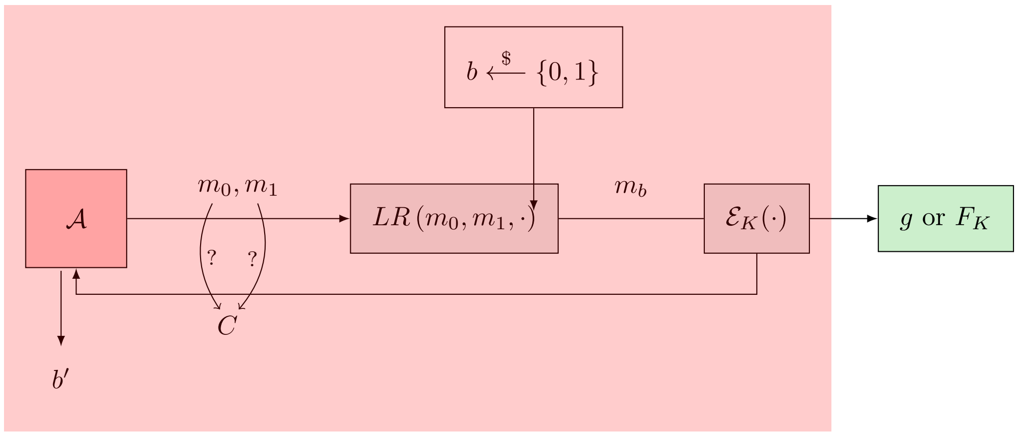
More formally, for any efficient adversary $\mathcal{A}$, $\exists \mathcal{B}$ with similar efficiency such that the IND-CPA advantage of $\mathcal{A}$ under CTRC mode is less than double the PRF advantage of $\mathcal{B}$ under a secure block cipher F:

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

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

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.



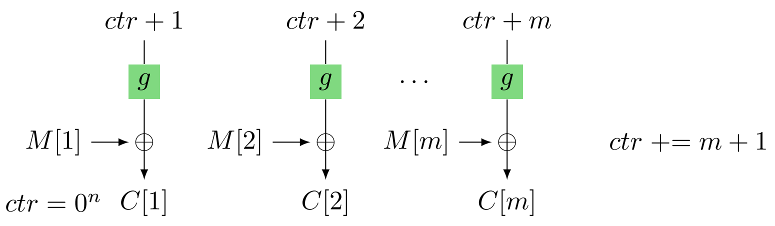
adversary $\mathcal{B} \longrightarrow$ 1 iff $b' = b$ else 0

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

Now let's analyze $\mathcal{B}$, expressing its PRF advantage over F in terms of $\mathcal{A}$'s IND-CPA advantage over CTRC.[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,

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

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



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

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

The theorem claim can then be trivially massaged out:

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

□

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

Input: An adversary $\mathcal{A}$ that executes oracle queries.
Result: 1 if $\mathcal{A}$ succeeds in breaking the emulated scheme, 0 otherwise.

```
Let $g \xleftarrow{\$} F(\ell, L)$ where $F$ is a PRF, which $\mathcal{B}$ will use as a block cipher.
Let $\mathcal{E}_f(\cdot)$ be an encryption function that works like $\mathcal{A}$ expects (for example, a CTRC scheme).
Choose a random bit: $b \xleftarrow{\$} \{0,1\}^1$
repeat
    Get a query from $\mathcal{A}$, some $(M_1, M_2)$
    $C \xleftarrow{\$} \mathcal{E}_f(M_b)$
    return $C$ to $\mathcal{A}$
until $\mathcal{A}$ outputs its guess, $b'$
return 1 iff $b = b'$, 0 otherwise
```

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

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

$$Adv_{CTR}^{ind\text{-}cpa}(\mathcal{A}) \le 2\cdot Adv_F^{prf}(\mathcal{B}) + \frac{\mu_A^2}{\ell 2^\ell}$$

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

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

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

[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).

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

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

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.

1.2.5 IND-CCA: Indistinguishability Under Chosen-Ciphertext Attacks

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

This isn't a far-fetched possibility,[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.

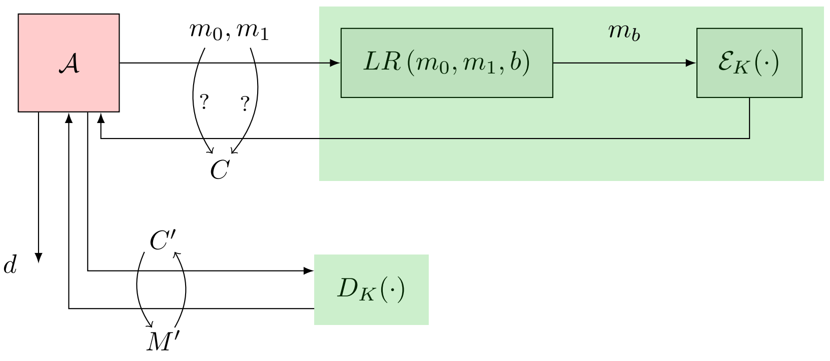


Figure 1.7: A visualization of the IND-CCA security definition. The adversary $\mathcal{A}$ submits two messages, (m_0, m_1), to an encryption oracle that (consistently) chooses one of them based on a bit b and returns m_b's ciphertext. The adversary can also submit any C' that hasn't been submitted to LR to a decryption oracle and see the resulting plaintext.

The only restriction on the attacker is that they cannot query the decryption oracle on ciphertexts returned from the encryption oracle (obviously, that would make determining b trivial) (in Figure 1.7, 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!

Definition 1.5: IND-CCA

A scheme $\mathcal{SE}$ is considered secure under IND-CCA if an adversary's **IND-CCA advantage**—the difference between their probability of guessing correctly and guessing incorrectly—is small (≈ 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 1.2 the way that message construction works under CBC with random initialization vectors.

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

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

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

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

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

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

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

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

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

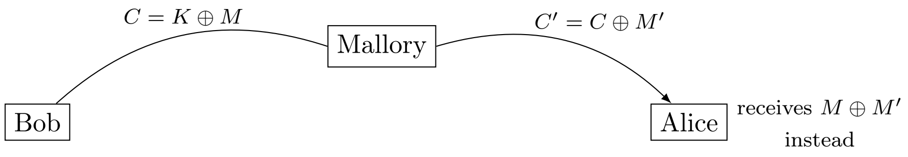
1.3 What Now?

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

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

Message Authentication Codes

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



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

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

2.1 Notation & Syntax

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

- a **message space**, denoted as the $\mathcal{M}$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, $\mathcal{M}$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**, $\mathcal{V}$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 $\mathcal{V}$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?

2.2 Security Evaluation

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

To start, we can imagine that an adversary can see some number of `(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

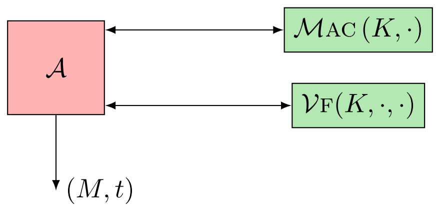
A **replay attack** is one where an adversary uses valid messages from the past that they captured to duplicate some action.

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.

2.2.1 UF-CMA: Unforgeability Under Chosen-Message Attacks

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


$\mathcal{A}$
$\mathcal{M}\text{AC}(K, \cdot)$
$\mathcal{V}\text{F}(K, \cdot, \cdot)$
(M, t)

Definition 2.1: 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:

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

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

[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.

A Toy Example

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

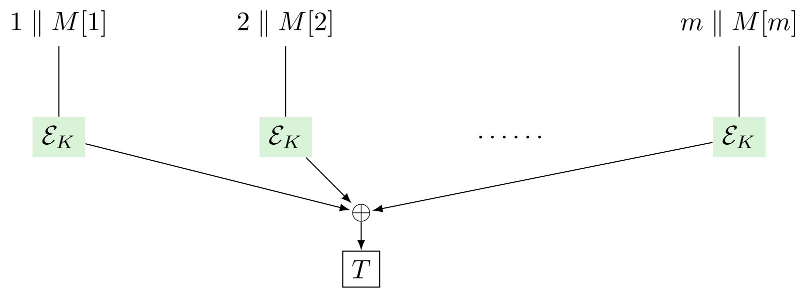


Figure 2.1: A simple MAC algorithm.

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

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

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

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

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 $\mathcal{M}\text{AC}$).

2.3 Mode of Operation: CBC-MAC

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

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

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

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

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.

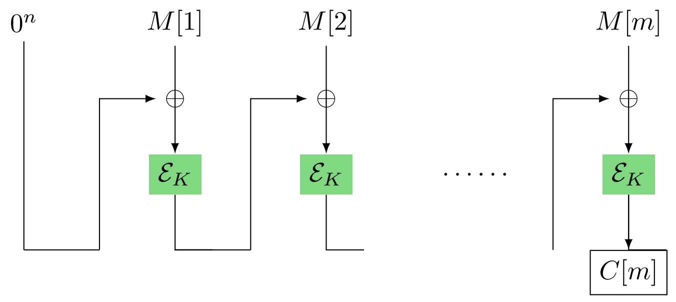


Figure 2.2: The CBC-MAC authentication mode.

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

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

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

(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

An essential part of modern cryptography, **hash functions** 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 3.1. 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 3.1: A list of some modern hash functions and their output digest length.

3.1 Collision Resistance

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

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

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.

[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.

2. Hash functions have collisions *by definition*, so the probability of finding one is *always* one. Even if we, as humans, don't *know* how to find the collision, this is a separate issue.

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

> *This is unfortunate in some ways, because it distances us from concrete hash functions like* SHA1*. But no alternative is known.*
>
> — Introduction to Modern Cryptography *(pp. 141)*

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

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

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

Definition 3.1: Collision Resistance

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

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

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:

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

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

3.2 Building Hash Functions

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

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

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

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

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

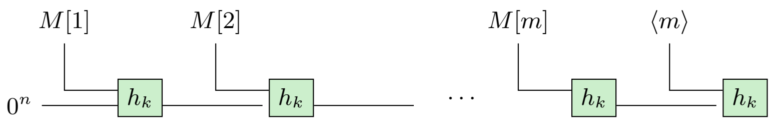


Figure 3.1: A visualization of the Merkle-Damgård transform.

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

Input: $h(\cdot)$, a limited-range compression function operating on b-bit inputs.
Input: M, the arbitrary-length input message to compress.
Result: M compressed to an n-bit digest.

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

The good news of this transform is the following:

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

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

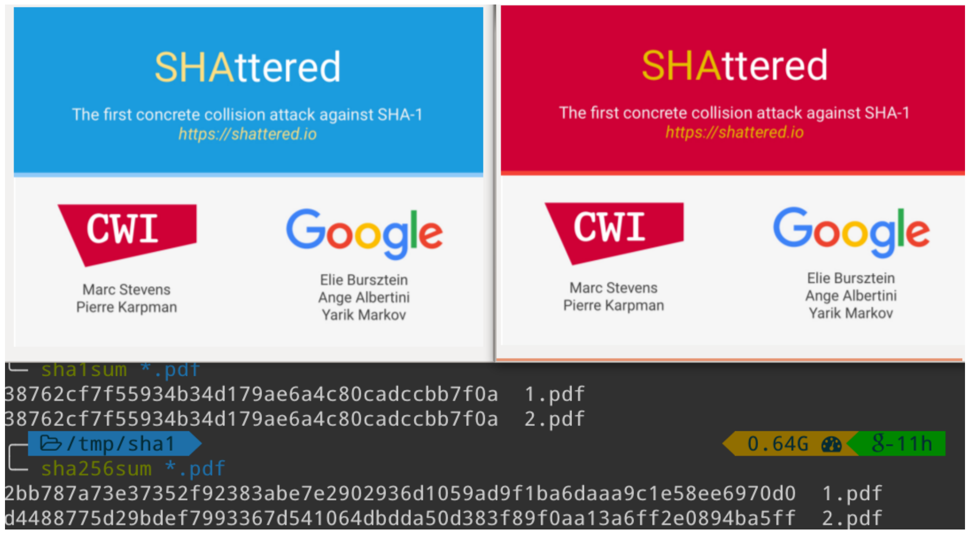
Birthday Attacks Recall the birthday paradox: as the number of samples from a range increases, the probability of any two of those samples being equal grows rapidly (there's a 95% chance that two people at a 50-person party will have the same birthday).

A hash function is **regular** if every range point has the same number of pre-images (that is, if every output has the same number of possible inputs). For such a function, the "birthday attack" finds a collision in $\approx 2^{n/2}$ trials. For a hash function that is *not* regular, such an attack could succeed even sooner.

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

[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.

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.



```
└ sha1sum *.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  1.pdf
38762cf7f55934b34d179ae6a4c80cadccbb7f0a  2.pdf
┌ /tmp/sha1                                              0.64G   8-11h
└ sha256sum *.pdf
2bb787a73e37352f92383abe7e2902936d1059ad9f1ba6daaa9c1e58ee6970d0  1.pdf
d4488775d29bdef7993367d541064dbdda50d383f89f0aa13a6ff2e0894ba5ff  2.pdf
```

Figure 3.2: 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,

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

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.

3.3 One-Way Functions

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

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

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

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 collision resistance.

Definition 3.2: One-Way Function

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

$$\text{collision resistance} \implies \text{one-wayness} \\ \text{one-wayness} \implies \text{collision resistance}$$

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:

$$h(x_1 x_2 \cdots x_n) = g(x_1 x_2 \cdots x_{n-1})$$

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

$$h(m_1 m_2 \cdots m_{n-1} 0) = h(m_1 m_2 \cdots m_{n-1} 1)$$

3.4 Hash-Based MACs

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

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

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

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

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

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

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.

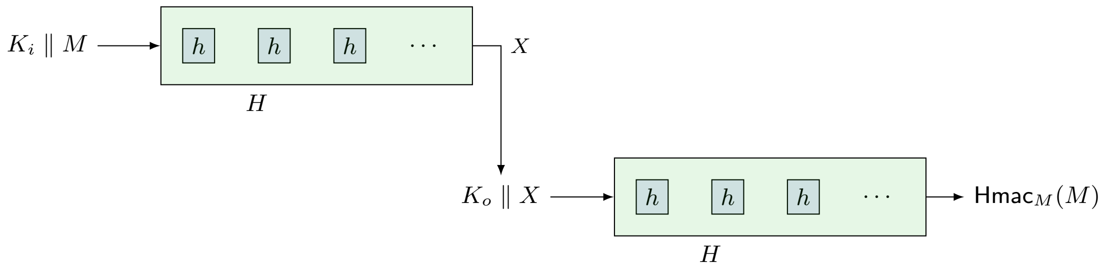


Figure 3.3: A visualization of the two-tiered structure of HMAC, the standard keyed message authentication code scheme.

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

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

Authenticated Encryption

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

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

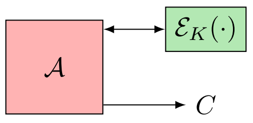
4.1 INT-CTXT: Integrity of Ciphertexts

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

Definition 4.1: INT-CTXT Security

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

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



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 4.1. *If a symmetric encryption scheme is secure under IND-CPA and INT-CTXT, then 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.

4.2 Generic Composite Schemes

Wikipedia

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

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

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

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

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

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

4.2.1 Encrypt-and-MAC

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

Algorithm 4.1: The encrypt-then-MAC encryption algorithm.

```
Input: $\mathcal{SE} = (\mathcal{E}_{K_m}, \mathcal{D}_{K_m})$
$C' \xleftarrow{\$} \mathcal{E}'_K(M)$
$T = F_{K_m}(M)$
return $C' \parallel T$
```

Algorithm 4.2: The encrypt-then-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

[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.

and can break IND-CPA (via Theorem 1.1).[2]

4.2.2 MAC-then-encrypt

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

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

Algorithm 4.3: The MAC-then-encrypt encryption algorithm.

```
Input: $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$
$T = F_{K_m}(M)$
$C' \xleftarrow{\$} \mathcal{E}'_{K_e}(M \parallel T)$
return $C'$
```

Algorithm 4.4: The MAC-then-encrypt decryption algorithm.

```
Input: $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$
$M \parallel T = \mathcal{D}'_{K_e}(C')$
if $T = F_{K_m}(M)$ then
    return $M$
end
return $\perp$
```

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

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

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

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

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*.

4.2.3 Encrypt-then-MAC

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

Caveat: Timing Attacks

Notice the key, extremely important nuance of the decryption routine in algorithm 4.6: the message is decrypted regardless of whether or not the tag is valid. From a performance perspective, we would ideally check the tag first, right? Unfortunately, this leads to the potential for an advanced **timing attack**: decryption of invalid messages now *takes less time* than valid ones, and this lets

[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.

the attacker to learn secret information about the scheme. Now, they can differentiate between an invalid tag and an invalid ciphertext.

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

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

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

Algorithm 4.5: The encrypt-then-MAC encryption algorithm.

```
Input: $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$
Input: $M$, an input plaintext message.
$C \xleftarrow{\$} \mathcal{E}'_{K_e}(MT)$
$T = F_{K_m}(C)$
return $C \parallel T$
```

Algorithm 4.6: The encrypt-then-MAC decryption algorithm.

```
Input: $\mathcal{SE} = (\mathcal{E}_{K_e}, \mathcal{D}_{K_e})$
Input: $C$, an input ciphertext message.
$M = \mathcal{D}'_{K_e}(C)$
if $T = F_{K_m}(C)$ then
    return $M$
end
return $\perp$
```

4.2.4 In Practice...

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

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

4.2.5 Dedicated Authenticated Encryption

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

The answer is yes, and the **offset codebook** mode is such a scheme. It's a one-pass, heavily parallelizable scheme.[4]

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

4.3 AEAD: Associated Data

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

4.3.1 GCM: Galois/Counter Mode

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

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

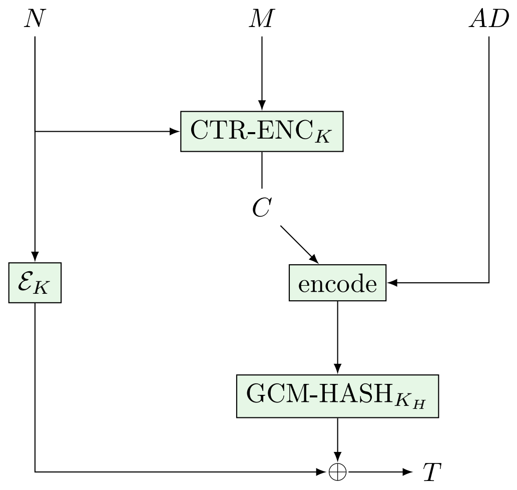


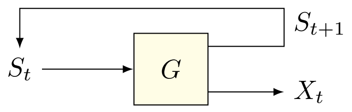
Figure 4.1: A visualization of the Galois/Counter mode (GCM) of AEAD encryption.

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.

5.1 Generators

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



We'll use the shorthand notation:

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

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

5.1.1 PRGs for Encryption

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

5.2 Evaluating PRGs

Creating a generator with unpredictable, random output is quite difficult. Functions build on **linear congruential generators** (LRGs) and **linear feedback shift registers** (LFSRs) have good distributions (equal numbers of 1s and 0s in the output) but are predictable given enough output. However, stream ciphers like RC4 (the 4th 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:

$$\left(\mathbf{X}^1, S_t\right) = \left(X_0^1 X_1^1 \cdots X_m^1, S_t\right) = G\left(S_0, m\right)$$
$$\mathbf{X}^0 = X_0^0 X_1^0 \cdots X_m^0 \xleftarrow{\$} \{0,1\}^m$$

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

Definition 5.1: INDR Security

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

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

5.3 Creating Stream Ciphers

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

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

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

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

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

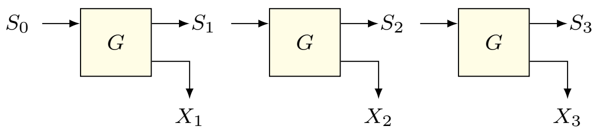
Input: *S_t, the current PRG input.*
Result: *(X, S_{t+1}), the pseudorandom value and the new PRG state.*

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

5.3.1 Forward Security

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



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

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

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

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

```
Input: $S_t$, the current PRG input.
Result: $(X, S_{t+1})$, the pseudorandom value and the new PRG state.
$X = E_K(0)$
$K = E_K(1)$
return $(X, K)$
```

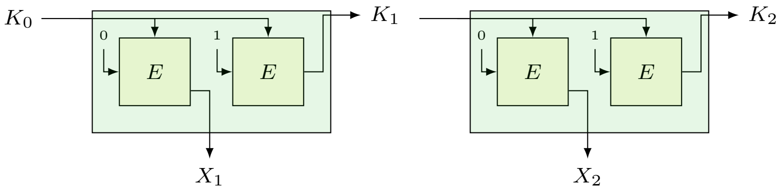


Figure 5.1: A visualization of a PRG with forward secrecy.

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.

5.3.2 Considerations

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

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

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

Common Implementation Mistakes

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

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

(Lack of) Security Proofs Using a mode of operation with no proof of security—or worse, modes with proofs of *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 1.4) had the following adversarial advantage:

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

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.

Index of Terms