The Basics of Provable Security

Until very recently, cryptography seemed doomed to be a cat-and-mouse game. Someone would come up with an encryption method, someone else would find a way to break it, and this process would repeat again and again. Crypto-enthusiast Edgar Allen Poe wrote in 1840,

"Human ingenuity cannot concoct a cypher which human ingenuity cannot resolve."

With the benefit of 21st-century knowledge, I would argue that Poe’s sentiment is not true. The code-makers can win against the code-breakers. Modern cryptography is full of schemes that we can prove are secure in a very specific sense.

In order to prove things about security, we must be very precise about what exactly we mean by “security.” Our study revolves around formal security definitions. In this chapter, we will learn how to write, understand, and interpret the meaning of a security definition; how to prove security using the technique of hybrids; and how to demonstrate insecurity by showing an attack violating a security definition.

!! A note to the reader: this chapter is rather theoretical/abstract since it introduces very general concepts that apply to the entire book.

2.1 Reasoning about Information Hiding via Code Libraries

Motivation

We begin with a simple example. Imagine that during a role-playing campaign, it becomes necessary to roll an 8-sided die. An 8-sided die is just a physical mechanism to sample uniformly from the set $\mathbb{Z}_8$ (in my world dice are numbered starting from 0). To be extremely precise, you would like to call this randomized subroutine:

```
ROLL-D8():
  d ← $\mathbb{Z}_8$
  return d
```

Unfortunately, you’ve misplaced your 8-sided die! Your only source of randomness is a coin (which you can think of as a 2-sided die). After a moment’s thought, you realize that you could just toss the coin 3 times to emulate an 8-sided die. That is, you could use your
coin to realize the following subroutine:

<table>
<thead>
<tr>
<th>ROLL-D8()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \leftarrow {0, 1}$</td>
</tr>
<tr>
<td>$b \leftarrow {0, 1}$</td>
</tr>
<tr>
<td>$c \leftarrow {0, 1}$</td>
</tr>
<tr>
<td>return $4a + 2b + c$</td>
</tr>
</tbody>
</table>

Although these two subroutines are obviously very different internally, they have the same external behavior. Specifically, both subroutines give a return value that is distributed uniformly in $\mathbb{Z}_8$.

The fact that the two subroutines have the same external behavior means:

> Just by calling the ROLL-D8 subroutine, you cannot tell which of these two implementations is being used.

As another example, if you have access to a sorting algorithm, then just by making calls to that algorithm you would not be able to tell whether it was sorting using quicksort or mergesort.

Notice that we are making statements about information being hidden. In the above examples, the input-output behavior of the subroutines completely hides (from the calling program) the choice of which subroutine is actually being run. The most fundamental goal in cryptography is to hide information, and we can use the terminology of subroutines, input-output behavior, calling programs, etc. to reason about security.

**Libraries & Interfaces**

All of the security definitions in this course are defined using a common methodology, which is based on familiar concepts from programming. The main idea is to formally define the “allowed” usage of a cryptographic scheme through a programmatic interface, and to define what information is hidden in terms of two implementations of that interface, called libraries.

**Definition 2.1 (Libraries)**

A library $L$ is a collection of subroutines and private/static variables. A library’s interface consists of the names, argument types, and output type of all of its subroutines. If a program $A$ includes calls to subroutines in the interface of $L$, then we write $A \circ L$ to denote the result of linking $A$ to $L$ in the natural way (answering those subroutine calls using the implementation specified in $L$). We write $A \circ L \Rightarrow z$ to denote the event that program $A \circ L$ outputs the value $z$.

**Discussion**

- If $A \circ L$ is a program that makes random choices, then its output is also a random variable, and it may often make sense to consider expressions like $\Pr[A \circ L \Rightarrow z]$.

- We can consider compound programs like $A \circ L_1 \circ L_2$. Our convention is that subroutine calls only happen from left to right across the $\circ$ symbol, so in this example, $L_2$ doesn’t call subroutines of $A$. We can then think of $A \circ L_1 \circ L_2$ as $(A \circ L_1) \circ L_2$.
(a compound program linked to $L_2$) or as $A \circ (L_1 \circ L_2)$ ($A$ linked to a compound library), whichever is convenient.

- Hopefully, security properties will be less daunting when they are expressed in terms of familiar concepts from CS like libraries, scope, etc. But a primary goal of security definitions is to be mathematically precise enough that we can actually prove things about security.

There is one way in which these familiar CS concepts don’t exactly match what we want when formulating a security definition using libraries. In real-world software, when a calling program is linked to a library, there are ways for the calling program to get information stored in the library, beyond just the advertised interface. For example, a calling program might be able to peek into a library’s internal memory, or measure the response time of a subroutine call, or see whether some memory access triggers a cache miss / page fault, etc.\(^1\)

In contrast, when using the language of libraries to reason about security, we always assume that the library’s explicit interface is the only way information gets in and out of the library. In that sense, you should think of security-definition libraries more as mathematical abstractions than realistic software.

This doesn’t mean that it’s impossible to reason about attacks where an adversary has side-channels of information on our cryptographic implementations. It’s just that if you want to prove something about what an adversary can do in the presence of such a side channel, then that side channel has to be explicit in the security definition, even if its purpose is to model a channel that is implicit in the real world.

### Interchangeability

Returning to our first examples, we know that there might be many different libraries that have the same input-output behavior. A library that provides a sort algorithm might implement it using quicksort or mergesort. A library that provides a roll-d8 algorithm might implement it using an 8-sided die or 3 coin flips.

Since we will consider libraries that use internal randomness, the outputs of subroutines may be probabilistic and we have to be careful about what we mean by “same input-output behavior.” A particularly convenient way to express this is to say that two libraries $L_{\text{left}}$ and $L_{\text{right}}$ have the same input-output behavior if no calling program behaves differently when linking to $L_{\text{left}}$ vs. $L_{\text{right}}$. More formally,

**Definition 2.2** (Interchangeable)

Let $L_{\text{left}}$ and $L_{\text{right}}$ be two libraries with a common interface. We say that $L_{\text{left}}$ and $L_{\text{right}}$ are **interchangeable**, and write $L_{\text{left}} \equiv L_{\text{right}}$, if for all programs $A$ that output a single bit,

$$\Pr[A \circ L_{\text{left}} \Rightarrow 1] = \Pr[A \circ L_{\text{right}} \Rightarrow 1].$$

**Discussion**

- We consider calling programs that produce only a single bit of output, which might seem unnecessarily restrictive. However, the definition says that the two libraries

\(^1\)In my experience, students who are interested in cryptography are also the most likely to be interested in these kinds of side channels.
have the same effect on all calling programs. In particular, the libraries must have the same effect on a calling program $A$ whose only goal is to distinguish between these particular libraries. A single output bit is necessary for this distinguishing task — just interpret the output bit as a “guess” for which library $A$ thinks it is linked to. For this reason, we will often refer to the calling program $A$ as a **distinguisher**.

- There is nothing special about defining interchangeability in terms of the calling program giving output 1. Since the only possible outputs are 0 and 1, we have:

$$\Pr[A \diamond L_{\text{left}} \Rightarrow 1] = \Pr[A \diamond L_{\text{right}} \Rightarrow 1]$$

$$\iff 1 - \Pr[A \diamond L_{\text{left}} \Rightarrow 1] = 1 - \Pr[A \diamond L_{\text{right}} \Rightarrow 1]$$

$$\iff \Pr[A \diamond L_{\text{left}} \Rightarrow 0] = \Pr[A \diamond L_{\text{right}} \Rightarrow 0].$$

- It is a common pitfall to imagine the program $A$ being simultaneously linked to both libraries. But in the definition, calling program $A$ is only ever linked to one of the libraries at a time.

- Taking the previous observation even further, the definition applies against calling programs $A$ that “know everything” about (more formally, whose code is allowed to depend arbitrarily on) the two libraries. This is a reflection of Kerckhoffs’ principle, which roughly says “assume that the attacker has full knowledge of the system.”

There is, however, a subtlety that deserves some careful attention, though. Our definitions will typically involve libraries that use internal randomness. Kerckhoffs’ principle allows the calling program to know which libraries are used, which in this case corresponds to how a library will choose randomness (i.e., from which distribution). It doesn’t mean that the adversary will know the result of the libraries’ choice of randomness (i.e., the values of all internal variables in the library). It’s the difference between knowing that you will choose a random card from a deck (i.e., the uniform distribution on a set of 52 items) versus reading your mind to know exactly what card you chose.

This subtlety is reflected in Definition 2.2 in the following way. First, we specify two libraries, then we consider a particular distinguisher, and only then do we link and execute the distinguisher with a library. The distinguisher cannot depend on the random choices made by the library, since the choice of randomness “happens after” the distinguisher is fixed.

**Kerckhoff’s Principle, adapted to our terminology:**

Assume that the distinguisher knows every fact in the universe, except for:

1. which of the two libraries it is linked to, and
2. the outcomes of random choices made by the library (often assigned to privately-scoped variables within the library).

---

\[\text{“Il faut qu’il n’exige pas le secret, et qu’il puisse sans inconvénient tomber entre les mains de l’ennemi.”}\]

Auguste Kerckhoffs, 1883. Translation: [The method] must not be required to be secret, and it can fall into the enemy’s hands without causing inconvenience.

13
The definitions here are general enough that they can apply to many future situations. Our first examples of libraries will be very simple, consisting of just a single subroutine. Later, our discussions of security will require libraries with multiple subroutines and persistent state between different subroutine calls (via static variables).

Defining interchangeability in terms of distinguishers may not seem entirely natural, but this definition allows us to ease into future concepts as well. Instead of requiring two libraries to have identical input-output behavior, we will eventually consider libraries that are “similar enough.” Distinguishers provide a conceptually simple way to measure the similarity between two libraries.

Suppose two libraries \( L_{\text{left}} \) and \( L_{\text{right}} \) are interchangeable: two libraries that are different but appear the same to all calling programs. Think of the internal differences between the two libraries as information that is perfectly hidden to the calling program. If the information weren’t perfectly hidden, then the calling program could get a whiff of whether it was linked to \( L_{\text{left}} \) or \( L_{\text{right}} \), and use it to act differently in those two cases. But such a calling program would contradict the fact that \( L_{\text{left}} \equiv L_{\text{right}} \).

This line of reasoning leads to:

\[
\text{The Prime Directive of Security Definitions:}
\]

\[
\text{If two libraries are interchangeable, then their common interface leaks no information about their internal differences.}
\]

We can use this principle to define security, as we will see shortly (and throughout the entire course). It is typical in cryptography to want to hide some sensitive information. To argue that the information is really hidden, we define two libraries with a common interface, which formally specifies what an adversary is allowed to do & learn. The two libraries are typically identical except in the choice of the sensitive information. The Prime Directive tells us that when the resulting two libraries are interchangeable, the sensitive information is indeed hidden when an adversary is allowed to do the things permitted by the libraries’ interface.

\[\text{An example: One-time pad’s uniformity property}\]

We have already seen an example of interchangeable libraries! In Claim 1.3 we proved that a particular subroutine \( \text{VIEW} \) generates uniformly distributed output, no matter what input is given to it.

Let’s restate that claim in the new terminology of libraries:

\[\text{Claim 2.3 (OTP rule)}\]

\[
\text{The following two libraries are interchangeable (i.e., } L_{\text{otp-real}} \equiv L_{\text{otp-rand}}):\]

\[
\begin{array}{|c|}
\hline
L_{\text{otp-real}} \\
\hline
\text{QUERY}(m \in \{0, 1\}^\lambda): \quad k \leftarrow \{0, 1\}^\lambda \\
\hline
\text{return } k \oplus m \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
L_{\text{otp-rand}} \\
\hline
\text{QUERY}(m \in \{0, 1\}^\lambda): \quad c \leftarrow \{0, 1\}^\lambda \\
\hline
\text{return } c \\
\hline
\end{array}
\]
You can tell just by looking that the two libraries $\mathcal{L}_{\text{otp-}}$ have the same interface. Claim 2.3 says something more specific: the two libraries in fact have the same input-output behavior.

### 2.2 A General-Purpose Security Definition for Encryption

It’s important and useful to be able to talk about security definitions that can apply to any encryption scheme. With such a general-purpose security definition, we can design a system in a modular way, saying “my system is secure as long as the encryption scheme being used has such-and-such property.” If concerns arise about a particular choice of encryption scheme, then we can easily swap it out for a different one, thanks to the clear abstraction boundary.

Claim 2.3 is a good security property, but it is rather specific to one-time pad. In this section, we develop a general-purpose security definition for encryption. That means it’s time to face the question, what are we really asking for when we want a “secure encryption method?”

#### Syntax of Encryption

Before tackling what it means to be secure, it is helpful to define what it means to be an encryption method in the first place.

**Definition 2.4**

A symmetric-key encryption (SKE) scheme consists of the following algorithms:

- **KeyGen**: a randomized algorithm that outputs a key $k \in \mathcal{K}$.
- **Enc**: a (possibly randomized) algorithm that takes a key $k \in \mathcal{K}$ and plaintext $m \in \mathcal{M}$ as input, and outputs a ciphertext $c \in \mathcal{C}$.
- **Dec**: a deterministic algorithm that takes a key $k \in \mathcal{K}$ and ciphertext $c \in \mathcal{C}$ as input, and outputs a plaintext $m \in \mathcal{M}$.

We call $\mathcal{K}$ the key space, $\mathcal{M}$ the message space, and $\mathcal{C}$ the ciphertext space of the scheme. When we use a single variable — say, $\Sigma$ — to refer to the scheme as a whole and distinguish one scheme from another, we write $\Sigma$.KeyGen, $\Sigma$.Enc, $\Sigma$.Dec, $\Sigma$.K, $\Sigma$.M, and $\Sigma$.C to refer to its components.

Because the same key is used for encryption and decryption, we refer to this style of encryption scheme as symmetric-key. It’s also sometimes referred to as secret-key or private-key encryption; these terms are somewhat confusing because even other styles of encryption involve things that are called secret/private keys.

This definition only says what kind of object an “encryption scheme” is. This information is the syntax of encryption.


**Correctness**

The syntax definition states what algorithms comprise an encryption scheme, but it does not say what behaviors the scheme should have. In particular, the syntax definition allows for KeyGen, Enc, Dec to always output all zeroes, but this would not be a very useful encryption scheme. An encryption scheme is only useful for communication if the receiver can learn the sender’s intended plaintext:

**Definition 2.5** An encryption scheme $\Sigma$ satisfies correctness if for all $k \in \Sigma.K$ and all $m \in \Sigma.M$,

$$
\Pr[\Sigma.\text{Dec}(k, \Sigma.\text{Enc}(k, m)) = m] = 1.
$$

The definition is expressed in terms of a probability, because Enc is allowed to be a randomized algorithm.

Note that correctness has little to do with security. An encryption scheme where $\text{Enc}(k, m) = m$ has no security at all but still satisfies the correctness property. One way to see that the definition has nothing to do with security is to recognize the absence of an adversary in the definition. A security property must refer to something that happens in the presence of an adversary.

**One-Time Secrecy**

We now develop a general-purpose security definition that makes sense for any encryption scheme. We will be considering an arbitrary, unspecified encryption scheme $\Sigma$. Let’s first consider a very simplistic scenario in which an eavesdropper sees the encryption of some plaintext. It should make sense that we are considering an eavesdropper who sees encrypted messages — if you are confident that no attacker will ever see your ciphertexts, then what is the point of encrypting? We will start with the following informal idea:

seeing a ciphertext should leak no information about the choice of plaintext.

Our goal is to formalize this property as a statement about interchangeable libraries.

We can work backwards from the Prime Directive. We will show two libraries whose common interface allows the calling program to see a ciphertext, and whose only internal difference was in the choice of plaintext that was encrypted. Saying that these two libraries are interchangeable is equivalent to saying that their common interface (seeing a ciphertext) leaks no information about the internal differences (the choice of plaintext).

The two libraries should look something like:

<table>
<thead>
<tr>
<th>QUERY(??):</th>
<th>QUERY(??):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \leftarrow \Sigma.\text{KeyGen}$</td>
<td>$k \leftarrow \Sigma.\text{KeyGen}$</td>
</tr>
<tr>
<td>$c \leftarrow \Sigma.\text{Enc}(k, m_L)$</td>
<td>$c \leftarrow \Sigma.\text{Enc}(k, m_R)$</td>
</tr>
<tr>
<td>return $c$</td>
<td>return $c$</td>
</tr>
</tbody>
</table>

Indeed, the common interface of these libraries allows the calling program to learn a ciphertext, and the libraries differ only in the choice of plaintext $m_L$ vs. $m_R$ (highlighted). We are getting very close! However, the libraries are still underspecified. Variables $m_L$
and \( m_R \) are undefined — where do they come from? Should they be fixed, hard-coded into the libraries? Should the libraries choose them randomly?

A good approach is actually to let the calling program itself choose \( m_L \) and \( m_R \). Think of this as giving the calling program control over precisely what the difference is between the two libraries. If the libraries are still interchangeable, then seeing a ciphertext leaks no information about the choice of plaintext, even if you already knew some partial information about the choice of plaintext, even if you knew that it was one of only two options, even if you got to choose those two options!

Putting these ideas together, we obtain the following definition:

**Definition 2.6 (One-time secrecy)**

Let \( \Sigma \) be an encryption scheme. We say that \( \Sigma \) is *(perfectly) one-time secret* if \( L_{\text{ots-L}}^\Sigma \equiv L_{\text{ots-R}}^\Sigma \), where:

\[
\begin{align*}
L_{\text{ots-L}}^\Sigma & \quad \text{QUERY}(m_L, m_R \in \Sigma.M) : \\
& k \leftarrow \Sigma.\text{KeyGen} \\
& c \leftarrow \Sigma.\text{Enc}(k, m_L) \\
& \text{return } c
\end{align*}
\]

\[
\begin{align*}
L_{\text{ots-R}}^\Sigma & \quad \text{QUERY}(m_L, m_R \in \Sigma.M) : \\
& k \leftarrow \Sigma.\text{KeyGen} \\
& c \leftarrow \Sigma.\text{Enc}(k, m_R) \\
& \text{return } c
\end{align*}
\]

This security notion is often called *perfect secrecy* in other sources.\(^3\)

**Interpreting/Critiquing a Security Definition**

Just because some author writes something down and calls it a security definition, there’s no guarantee that it’s a *good* definition! In fact, some security definitions are weak, some are strong, some are trivially weak, some are impossibly strong (no scheme can satisfy them), some are a good model of real-world applications of encryption, some are unrealistic. *All* security definitions have limitations.

In short, *security definitions should always be viewed critically.* A security definition is a contract that says, “a cryptographic scheme provides a particular guarantee, when it is used in a very specific context.” You might want to use the scheme in a different context than the one in the security definition, in which you might not get *any* security guarantee. Even in the right context, you might want a different guarantee than the one promised by the definition.

Never simply say, *this scheme satisfies a “security definition,” so I can safely use it for whatever I want.*

To make things concrete, let’s talk about the limitations of one-time secrecy as a security definition. Here are a few things that are *not* covered by the definition:

- The definition assumes that a key is used to encrypt only a single plaintext: each time \texttt{QUERY} is called, a fresh key is chosen as \( k \). Of course, it is *possible* for different calls to \texttt{QUERY} to use the same value \( k \) by chance. But there is no way for the calling program to *ensure* that two plaintexts are encrypted under the same key. This

\(^3\)Personally, I think that using the term “perfect” leads to an impression that one-time pad should always be favored over any other kind of encryption scheme (presumably with only “imperfect” security). But if you want encryption, then you should almost never favor plain old one-time pad.
means that one-time secrecy gives no guarantee about what happens when a key is
(purposefully, consistently — not just by chance) used to encrypt several plaintexts.

▶ The definition doesn’t involve the Dec algorithm of the scheme at all! This fact has
many consequences. One consequence is that one-time secrecy does not prevent
an adversary from generating valid-looking ciphertexts. That is, an adversary can
generate a ciphertext that a receiver would happily decrypt (and presumably act
upon). If you want a scheme where the receiver will detect/reject ciphertexts that
were not generated by the sender (who knows \( k \)), then you have to find a security
definition that captures that guarantee. It is not a guarantee provided by the one-
time security definition.

▶ The definition assumes that the adversary (calling program) has no information
about the key (other than its length, which is a publicly-known parameter): the
variable \( k \) is private and falls out of scope at the end of the call to \textsc{query}. One-time
secrecy gives no guarantee when the adversary has some partial information about
the key (e.g., the adversary knows that the key has an odd number of 1s).

▶ The definition does not guarantee that the ciphertext hides the fact that a plaintext
comes from the set \( M \). In both libraries, the plaintext (\( m_L \) or \( m_R \)) is an element of \( M \),
whereas only the differences between the libraries are hidden. More concretely, for
one-time pad we have \( M = \{0, 1\}^\lambda \). The one-time secrecy definition does not hide
the fact that the plaintext is \( \lambda \) bits long, and indeed, you can quite clearly deduce
the length of the plaintext from the length of the one-time pad ciphertext. If you
want to hide the length of a plaintext, then you need to find a security definition
that captures that guarantee. It is not a guarantee provided by the one-time secrecy
definition.

▶ Here is a very subtle issue. The definition assumes that the choice of plaintext does
not depend on the key: the calling program’s choice of \( m_L, m_R \) happens before the
library chooses \( k \). There is no way for the calling program to force the library to use
the key itself as a plaintext (again, unless it happens by chance). Therefore one-time
secrecy gives no guarantee about what happens when users (intentionally) encrypt
their own keys. Indeed, encryptions of the key can “look different” than encryptions
of other things (see Exercise 1.3), even though the scheme satisfies one-time secrecy.

▶ The definition doesn’t specify how the sender and receiver come to know a common
key \( k \) in the real-world. That problem is considered out of scope for encryption
(it is known as key distribution). You can think of the problem of (symmetric-key)
encryption as how two users can securely communicate once they have established
a shared key.

The point of all this is not to pick on one-time secrecy, even though it is a relatively
weak security definition. The point is merely to show that every security definition has
limitations, and to get you in the habit of interpreting/understanding security definitions
in this way. In fact, only the first two limitations in this list are specific to one-time secrecy,
while the other limitations are shared by even the strongest security definitions in this
book.
2.3 How to Prove Security with the Hybrid Technique

We now have a general-purpose security definition (one-time secrecy) and we know of one encryption scheme (one-time pad). The natural next step is to show that one-time pad satisfies one-time secrecy.

A Useful Chaining Lemma

Before proving the security of one-time pad, we first show a helpful lemma that will be used in essentially every security proof that deals with libraries.

Lemma 2.7

If \( L_{\text{left}} \equiv L_{\text{right}} \) then \( L^* \circ L_{\text{left}} \equiv L^* \circ L_{\text{right}} \) for any library \( L^* \).

Proof

Take an arbitrary calling program \( \mathcal{A} \) and consider the compound program \( \mathcal{A} \circ L^* \circ L_{\text{left}} \). We can interpret this program as a calling program \( \mathcal{A} \) linked to the library \( L^* \circ L_{\text{left}} \), or alternatively as a calling program \( \mathcal{A} \circ L^* \) linked to the library \( L_{\text{left}} \). After all, \( \mathcal{A} \circ L^* \) is some program that makes calls to the interface of \( L_{\text{left}} \). Since \( L_{\text{left}} \equiv L_{\text{right}} \), swapping \( L_{\text{left}} \) for \( L_{\text{right}} \) has no effect on the output of any calling program. In particular, it has no effect when the calling program happens to be \( \mathcal{A} \circ L^* \). Hence we have:

\[
\Pr[\mathcal{A} \circ (L^* \circ L_{\text{left}}) \Rightarrow 1] = \Pr[\mathcal{A} \circ L^* \circ L_{\text{left}} \Rightarrow 1] \quad \text{(change of perspective)}
\]

\[
= \Pr[\mathcal{A} \circ L^* \circ L_{\text{right}} \Rightarrow 1] \quad \text{(since } L_{\text{left}} \equiv L_{\text{right}}) \]

\[
= \Pr[\mathcal{A} \circ (L^* \circ L_{\text{right}}) \Rightarrow 1]. \quad \text{(change of perspective)}
\]

Since \( \mathcal{A} \) was arbitrary, we have proved the lemma. ■

One-Time Secrecy of One-Time Pad

Remember that to prove a security property we must show that two libraries are interchangeable.

Theorem 2.8

Let \( \text{OTP} \) denote the one-time pad encryption scheme (Construction 1.1). Then \( \text{OTP} \) has one-time secrecy. That is, \( L_{\text{ots-L}}^{\text{OTP}} \equiv L_{\text{ots-R}}^{\text{OTP}} \).

Given what we already know about one-time pad, it’s not out of the question that we could simply “eyeball” the claim \( L_{\text{ots-L}}^{\text{OTP}} \equiv L_{\text{ots-R}}^{\text{OTP}} \). Indeed, we have already shown that in both libraries the \text{QUERY} subroutine simply returns a uniformly random string. A direct proof along these lines is certainly possible.

Instead of directly relating the behavior of the two libraries, however, we will instead show that:

\[
L_{\text{ots-L}}^{\text{OTP}} \equiv L_{\text{hyb-1}} \equiv L_{\text{hyb-2}} \equiv L_{\text{hyb-3}} \equiv L_{\text{hyb-4}} \equiv L_{\text{ots-R}}^{\text{OTP}},
\]

where \( L_{\text{hyb-1}}, \ldots, L_{\text{hyb-4}} \) are a sequence of what we call hybrid libraries. (It is not hard to see that the “\( \equiv \)” relation is transitive, so this proves that \( L_{\text{ots-L}}^{\text{OTP}} \equiv L_{\text{ots-R}}^{\text{OTP}} \).) This proof technique is called the hybrid technique.

Again, the hybrid technique is likely overkill for proving the security of one-time pad. The reason for going to all this extra effort is that it is how all proofs in this book are done.
We use one-time pad as an opportunity to gently introduce the technique. Hybrid proofs have the advantage that it can be quite easy to justify that adjacent hybrids (e.g., $L_{\text{hyb-}i}$ and $L_{\text{hyb-}(i+1)}$) are interchangeable, so the method scales well even in proofs where the “endpoints” of the hybrid sequence are quite different.

Proof

As described above, we will prove that

$$L_{\text{OTP-ots-L}} \equiv L_{\text{hyb-1}} \equiv L_{\text{hyb-2}} \equiv L_{\text{hyb-3}} \equiv L_{\text{hyb-4}} \equiv L_{\text{OTP-ots-R}},$$

for a particular sequence of $L_{\text{hyb-}i}$ libraries that we choose. For each hybrid, we highlight the differences from the previous one, and argue why adjacent hybrids are interchangeable.

As promised, the hybrid sequence begins with $L_{\text{OTP-ots-L}}$. The details of one-time pad have been filled in and highlighted.

Factoring out a block of statements into a subroutine does not affect the library’s behavior. Note that the new subroutine is exactly the $L_{\text{otp-real}}$ library from Claim 2.3 (with the subroutine name changed to avoid naming conflicts). This is no accident!

$L_{\text{otp-real}}$ has been replaced with $L_{\text{otp-rand}}$. From Claim 2.3 along with the chaining lemma Lemma 2.7, this change has no effect on the library’s behavior.

The argument to $\text{QUERY'}$ has been changed from $m_L$ to $m_R$. This has no effect on the library’s behavior since $\text{QUERY'}$ does not actually use its argument in these hybrids.

The previous transition is the most important one in the proof, as it gives insight into how we came up with this particular sequence of hybrids. Looking at the desired endpoints of our sequence of hybrids — $L_{\text{OTP-ots-L}}$ and $L_{\text{OTP-ots-R}}$ — we see that they differ only in swapping $m_L$ for $m_R$. If we are not comfortable eyeballing things, we’d like a better justification for why it is “safe” to exchange $m_L$ for $m_R$. However, the one-time pad rule (Claim 2.3) shows
that $L_{\text{ots-L}}^{\text{OTP}}$ in fact has the same behavior as a library $L_{\text{hyb-2}}$ that doesn’t use either of $m_L$ or $m_R$. Now, in a program that doesn’t use $m_L$ or $m_R$, it is clear that we can switch them.

Having made this crucial change, we can now perform the same sequence of steps, but in reverse. $L_{\text{hyb-4}}$ has been replaced with $L_{\text{ots-R}}$. Again, this has no effect on the library’s behavior, due to Claim 2.3. Note that the chaining lemma is being applied with respect to a different common $L^*$ than before (now $m_R$ instead of $m_L$ is being used).

A subroutine call has been inlined, which has no effect on the library’s behavior. The result is exactly $L_{\text{ots-R}}^{\text{OTP}}$.

Putting everything together, we showed that $L_{\text{ots-L}}^{\text{OTP}} \equiv L_{\text{hyb-1}} \equiv \cdots \equiv L_{\text{hyb-4}} \equiv L_{\text{ots-R}}^{\text{OTP}}$. This completes the proof, and we conclude that one-time pad satisfies the definition of one-time secrecy.

### Summary of the Hybrid Technique

We have now seen our first example of the hybrid technique for security proofs. The example illustrates features that are common to all security proofs used in this course:

- Proving security amounts to showing that two particular libraries, say $L_{\text{left}}$ and $L_{\text{right}}$, are interchangeable.

- To show this, we show a sequence of hybrid libraries, beginning with $L_{\text{left}}$ and ending with $L_{\text{right}}$. The hybrid sequence corresponds to a sequence of allowable modifications to the library. Each modification is small enough that we can easily justify why it doesn’t affect the calling program’s output probability.

- Simple things like factoring out & inlining subroutines, changing unused variables, consistently renaming variables, removing & changing unreachable statements, or unrolling loops are always “allowable” modifications in a hybrid proof. As we progress in the course, we will add to our toolbox of allowable modifications. For instance, if we want to prove security of some complicated system that uses a one-time-secret encryption $\Sigma$ as one of its components, then we are allowed to replace $L_{\text{ots-L}}^{\Sigma}$ with $L_{\text{ots-R}}^{\Sigma}$ as one of the steps in the hybrid proof.
### 2.4 How to Demonstrate Insecurity with Attacks

We have seen an example of how to prove that an encryption scheme is secure. To show that a scheme is *insecure*, we just have to show that the two relevant libraries are not interchangeable. To show that two libraries are not interchangeable, all we have to do is show just one calling program that behaves differently in the presence of the two libraries!

To make the process sound more exciting, we refer to such a demonstration as an *attack*.

Below is an example of an insecure encryption scheme:

#### Construction 2.9

- **\( \mathcal{K} \)** = \{permutations of \( \{1, \ldots, \lambda\} \) \}
- **\( \mathcal{M} \)** = \( \{0, 1\}^\lambda \)
- **\( \mathcal{C} \)** = \( \{0, 1\}^\lambda \)

**KeyGen:**

\[
\frac{k \leftarrow \mathcal{K}}{\text{return } k}
\]

**Enc(} (k, m):**

\[
\text{for } i := 1 \text{ to } \lambda:
\]

\[
c_{k(i)} := m_i
\]

\[
\text{return } c_1 \cdots c_\lambda
\]

**Dec(} (k, c):**

\[
\text{for } i := 1 \text{ to } \lambda:
\]

\[
m_i := c_{k(i)}
\]

\[
\text{return } m_1 \cdots m_\lambda
\]

To encrypt a plaintext \( m \), the scheme simply rearranges its bits according to the permutation \( k \).

#### Claim 2.10

*Construction 2.9 does not have one-time secrecy.*

**Proof**

Our goal is to construct a program \( \mathcal{A} \) so that \( \Pr[\mathcal{A} \circ \mathcal{L}_{\text{ots-L}}^\Sigma \Rightarrow 1] \) and \( \Pr[\mathcal{A} \circ \mathcal{L}_{\text{ots-R}}^\Sigma \Rightarrow 1] \) are different, where \( \Sigma \) refers to Construction 2.9. There are probably many "reasons" why this construction is insecure, each of which leads to a different distinguisher \( \mathcal{A} \). We need only demonstrate one such \( \mathcal{A} \), and it’s generally a good habit to try to find one that makes the probabilities \( \Pr[\mathcal{A} \circ \mathcal{L}_{\text{ots-L}}^\Sigma \Rightarrow 1] \) and \( \Pr[\mathcal{A} \circ \mathcal{L}_{\text{ots-R}}^\Sigma \Rightarrow 1] \) as different as possible.

One immediate observation about the construction is that it only rearranges bits of the plaintext, without modifying them. In particular, encryption preserves (leaks) the number of 0s and 1s in the plaintext. By counting the number of 0s and 1s in the ciphertext, we know exactly how many 0s and 1s were in the plaintext. Let’s try to leverage this observation to construct an actual distinguisher.

Any distinguisher must use the interface of the \( \mathcal{L}_{\text{ots-}} \) libraries; in other words, we should expect the distinguisher to call the \text{QUERY} subroutine with *some* choice of \( m_L \) and \( m_R \), and then do something based on the answer that it gets. If we are the ones writing the distinguisher, we must specify how these arguments \( m_L \) and \( m_R \) are chosen. Following the observation above, we can choose \( m_L \) and \( m_R \) to have a different number of 0s and 1s. An extreme example (and why not be extreme?) would be to choose \( m_L = 0^\lambda \) and \( m_R = 1^\lambda \). By looking at the ciphertext, we can determine which of \( m_L, m_R \) was encrypted, and hence which of the two libraries we are currently linked with.

Putting it all together, we define the following distinguisher:

\[
\frac{c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)}{\text{return } c \equiv 0^\lambda}
\]
Here is what it looks like when $\mathcal{A}$ is linked to $L_{ots-L}^\Sigma$ (we have filled in the details of Construction 2.9 in $L_{ots-L}^\Sigma$):

<table>
<thead>
<tr>
<th>$\mathcal{A}$</th>
<th>$L_{ots-L}^\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)$</td>
<td>$\text{QUERY}(m_L, m_R)$:</td>
</tr>
<tr>
<td>return $c \equiv 0^\lambda$</td>
<td>$k \leftarrow {\text{permutations of } {1, \ldots, \lambda}}$</td>
</tr>
<tr>
<td></td>
<td>for $i := 1$ to $\lambda$:</td>
</tr>
<tr>
<td></td>
<td>$c_k(i) := (m_L)_i$</td>
</tr>
<tr>
<td></td>
<td>return $c_1 \cdots c_\lambda$</td>
</tr>
</tbody>
</table>

We can see that $m_L$ takes on the value $0^\lambda$, so each bit of $m_L$ is $0$, and each bit of $c$ is $0$. Hence, the final output of $\mathcal{A}$ is $1$ (true). We have:

$$\Pr[\mathcal{A} \diamond L_{ots-L}^\Sigma \Rightarrow 1] = 1.$$ 

Here is what it looks like when $\mathcal{A}$ is linked to $L_{ots-R}^\Sigma$:

<table>
<thead>
<tr>
<th>$\mathcal{A}$</th>
<th>$L_{ots-R}^\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)$</td>
<td>$\text{QUERY}(m_L, m_R)$:</td>
</tr>
<tr>
<td>return $c \equiv 0^\lambda$</td>
<td>$k \leftarrow {\text{permutations of } {1, \ldots, \lambda}}$</td>
</tr>
<tr>
<td></td>
<td>for $i := 1$ to $\lambda$:</td>
</tr>
<tr>
<td></td>
<td>$c_k(i) := (m_R)_i$</td>
</tr>
<tr>
<td></td>
<td>return $c_1 \cdots c_\lambda$</td>
</tr>
</tbody>
</table>

We can see that each bit of $m_R$, and hence each bit of $c$, is $1$. So $\mathcal{A}$ will output $0$ (false), giving:

$$\Pr[\mathcal{A} \diamond L_{ots-R}^\Sigma \Rightarrow 1] = 0.$$ 

The two probabilities are different, demonstrating that $\mathcal{A}$ behaves differently (in fact, as differently as possible) when linked to the two libraries. We conclude that Construction 2.9 does not satisfy the definition of one-time secrecy.

**Exercises**

2.1. In abstract algebra, a (finite) **group** is a finite set $\mathbb{G}$ of items together with an operator $\otimes$ satisfying the following axioms:

- **Closure**: for all $a, b \in \mathbb{G}$, we have $a \otimes b \in \mathbb{G}$.
- **Identity**: there is a special identity element $e \in \mathbb{G}$ that satisfies $e \otimes a = a$ for all $a \in \mathbb{G}$. We typically write "1" rather than $e$ for the identity element.
- **Associativity**: for all $a, b, c \in \mathbb{G}$, we have $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.
- **Inverses**: for all $a \in \mathbb{G}$, there exists an inverse element $b \in \mathbb{G}$ such that $a \otimes b = b \otimes a$ is the identity element of $\mathbb{G}$. We typically write "$a^{-1}$" for the inverse of $a$. 

23
Define the following encryption scheme in terms of an arbitrary group \((\mathcal{G}, \otimes)\):

| \(\mathcal{K}\) = \(\mathcal{G}\) | KeyGen: \(k \leftarrow \mathcal{G}\) | Enc\((k, m)\): return \(k \otimes m\) | Dec\((k, c)\): ?? |
| \(\mathcal{M}\) = \(\mathcal{G}\) | return \(k\) |
| \(\mathcal{C}\) = \(\mathcal{G}\) |

(a) Prove that \(\{0,1\}^\lambda\) is a group with respect to the xor operator. What is the identity element, and what is the inverse of a value \(x \in \{0,1\}^\lambda\)?

(b) Fill in the details of the Dec algorithm and prove (using the group axioms) that the scheme satisfies correctness.

(c) Prove that the scheme satisfies one-time secrecy.

2.2. Show that the following encryption scheme does not have one-time secrecy, by constructing a program that distinguishes the two relevant libraries from the one-time secrecy definition.

| \(\mathcal{K}\) = \{1, \ldots, 9\} | KeyGen: \(k \leftarrow \{1, \ldots, 9\}\) | Enc\((k, m)\): return \(k \times m \mod 10\) |
| \(\mathcal{M}\) = \{1, \ldots, 9\} | return \(k\) |
| \(\mathcal{C}\) = \(\mathbb{Z}_{10}\) |

2.3. Consider the following encryption scheme. It supports plaintexts from \(\mathcal{M} = \{0,1\}^\lambda\) and ciphertexts from \(\mathcal{C} = \{0,1\}^{2\lambda}\). Its keyspace is:

\[\mathcal{K} = \{k \in \{0,1,\_\}^{2\lambda} \mid k \text{ contains exactly } \lambda \text{ " } \_ \text{ " characters}\}\]

To encrypt plaintext \(m\) under key \(k\), we “fill in” the \(\_\) characters in \(k\) using the bits of \(m\).

Show that the scheme does not have one-time secrecy, by constructing a program that distinguishes the two relevant libraries from the one-time secrecy definition.

Example: Below is an example encryption of \(m = 1101100001\).

\[k = 1\_0\_11010\_100\_0\_\_\_\]
\[m = 11 \ 01 \ 1 \ 0 \ 0\ 0\]
\[\Rightarrow \text{Enc}(k, m) = 11100111010110000001\]

2.4. Suppose we modify the scheme from the previous problem to first permute the bits of \(m\) (as in Construction 2.9) and then use them to fill in the \(\_\) characters in a template string. In other words, the key specifies a random permutation on positions \(\{1, \ldots, \lambda\}\) as well as a random template string that is \(2\lambda\) characters long with \(\lambda\) \(\_\) characters.

Show that even with this modification the scheme does not have one-time secrecy.

2.5. Prove that if an encryption scheme \(\Sigma\) has \(|\Sigma.\mathcal{K}| < |\Sigma.\mathcal{M}|\) then it cannot satisfy one-time secrecy. Try to structure your proof as an explicit attack on such a scheme.
2.6. Let $\Sigma$ denote an encryption scheme where $\Sigma.C \subseteq \Sigma.M$ (so that it is possible to use the scheme to encrypt its own ciphertexts). Define $\Sigma^2$ to be the following **nested-encryption** scheme:

- $\mathcal{K} = (\Sigma.K)^2$
- $\mathcal{M} = \Sigma.M$
- $C = \Sigma.C$

\[
\text{KeyGen:} \quad k_1 \leftarrow \Sigma.K \quad k_2 \leftarrow \Sigma.K \\
\text{return } (k_1, k_2)
\]

- $\text{Enc}((k_1, k_2), m)$:
  - $c_1 := \Sigma.\text{Enc}(k_1, m)$
  - $c_2 := \Sigma.\text{Enc}(k_2, c_1)$
  - return $c_2$

- $\text{Dec}((k_1, k_2), c_2)$:
  - $c_1 := \Sigma.\text{Dec}(k_2, c_2)$
  - $m := \Sigma.\text{Dec}(k_1, c_1)$
  - return $m$

Prove that if $\Sigma$ satisfies one-time secrecy, then so does $\Sigma^2$.

2.7. Let $\Sigma$ denote an encryption scheme and define $\Sigma^2$ to be the following **encrypt-twice** scheme:

- $\mathcal{K} = (\Sigma.K)^2$
- $\mathcal{M} = \Sigma.M$
- $C = \Sigma.C$

\[
\text{KeyGen:} \quad k_1 \leftarrow \Sigma.K \quad k_2 \leftarrow \Sigma.K \\
\text{return } (k_1, k_2)
\]

- $\text{Enc}((k_1, k_2), m)$:
  - $c_1 := \Sigma.\text{Enc}(k_1, m)$
  - $c_2 := \Sigma.\text{Enc}(k_2, c_1)$
  - return $(c_1, c_2)$

- $\text{Dec}((k_1, k_2), (c_1, c_2))$:
  - $m_1 := \Sigma.\text{Dec}(k_1, c_1)$
  - $m_2 := \Sigma.\text{Dec}(k_2, c_2)$
  - if $m_1 \neq m_2$ return $\text{err}$
  - return $m_1$

Prove that if $\Sigma$ satisfies one-time secrecy, then so does $\Sigma^2$.

2.8. Formally define a variant of the one-time secrecy definition in which the calling program can obtain two ciphertexts (on chosen plaintexts) encrypted under the same key. Call it two-time secrecy.

(a) Suppose someone tries to prove that one-time secrecy implies two-time secrecy. Show where the proof appears to break down.

(b) Describe an attack demonstrating that one-time pad does not satisfy your definition of two-time secrecy.

2.9. Show that the following libraries are interchangeable:

<table>
<thead>
<tr>
<th>$\mathcal{L}_1$</th>
<th>$\mathcal{L}_2$</th>
</tr>
</thead>
</table>
| **QUERY**($m \in \{0, 1\}^\lambda$):
  - $x \leftarrow \{0, 1\}^\lambda$
  - $y := x \oplus m$
  - return $(x, y)$ |
| **QUERY**($m \in \{0, 1\}^\lambda$):
  - $y \leftarrow \{0, 1\}^\lambda$
  - $x := y \oplus m$
  - return $(x, y)$ |

Note that $x$ and $y$ are swapped in the first two lines, but not in the return statement.
2.10. In this problem we consider modifying one-time pad so that the key is not chosen uniformly. Let $D_\lambda$ denote the probability distribution over $\{0, 1\}^\lambda$ where we choose each bit of the result to be 0 with probability 0.4 and 1 with probability 0.6.

Let $\Sigma$ denote one-time pad encryption scheme but with the key sampled from distribution $D_\lambda$ rather than uniformly in $\{0, 1\}^\lambda$.

(a) Consider the case of $\lambda = 5$. A calling program $A$ for the $L_{ots-\star}$ libraries calls $\text{QUERY}(01011, 10001)$ and receives the result 01101. What is the probability that this happens, assuming that $A$ is linked to $L_{ots-L}$? What about when $A$ is linked to $L_{ots-R}$?

(b) Turn this observation into an explicit attack on the one-time secrecy of $\Sigma$. 
