2 The Basics of Provable Security

Until very recently, cryptography seemed doomed to be a cat-and-mouse game. Someone would come up with a way to encrypt, someone else would find a way to break the encryption, 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 hindsight, we can now see that Poe's sentiment is not true. Modern cryptography is full of schemes that we can **prove** are secure in a very specific sense.

If we wish 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 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.

2.1 Reasoning about Information Hiding via Code Libraries

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 (libraries).

Definition 2.1 (Libraries)

*A **library** $\mathcal{L}$ 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 $\mathcal{A}$ includes calls to subroutines in the interface of $\mathcal{L}$, then we write $\mathcal{A} \diamond \mathcal{L}$ to denote the result of **linking** $\mathcal{A}$ to $\mathcal{L}$ in the natural way (answering those subroutine calls using the implementation specified in $\mathcal{L}$). We write $\mathcal{A} \diamond \mathcal{L} \Rightarrow z$ to denote the event that program $\mathcal{A} \diamond \mathcal{L}$ outputs the value z.*

Some more specifics:

- If $\mathcal{A} \diamond \mathcal{L}$ is a program that makes random choices, then its output is also a random variable.
- We can consider compound programs like $\mathcal{A} \diamond \mathcal{L}_1 \diamond \mathcal{L}_2$. Our convention is that subroutine calls only happen from left to right across the $\diamond$ symbol, so in this example, $\mathcal{L}_2$ doesn't call subroutines of $\mathcal{A}$. We can then think of $\mathcal{A} \diamond \mathcal{L}_1 \diamond \mathcal{L}_2$ as $(\mathcal{A} \diamond \mathcal{L}_1) \diamond \mathcal{L}_2$ (a compound program linked to $\mathcal{L}_2$) or as $\mathcal{A} \diamond (\mathcal{L}_1 \diamond \mathcal{L}_2)$ ($\mathcal{A}$ linked to a compound library), whichever is convenient.

- We try to make formal security definitions less daunting by expressing them in terms of elementary CS concepts like libraries, scope, etc. But one must not forget that the ultimate goal of security definitions is to be mathematically precise enough that we can actually *prove* things about security.

  For this reason, we need to have a handle on exactly what information the calling program can obtain from the library. We assume that the library's *explicit interface* is the only way information gets in and out of the library. This is at odds with real-world software, where you can find *implicit* channels of information (*e.g.*, peeking into a library's internal memory, measuring the response time of a subroutine call, etc.).[1] In short, don't get too carried away with the terminology of real-world software — you should think of these libraries more as *mathematical abstractions* than software.

Imagine the interface of a sorting library: when you call a subroutine $\textsc{sort}(A)$, you expect back a list that contains a sorted arrangement of the contents of A. As you know, there might be many ways to *implement* such a subroutine SORT. If you have access to only the input/output of SORT, then you would not be able to determine whether SORT was being implemented by mergesort or quicksort, for example, because both algorithms realize the same input-output behavior (sorting a list).

This kind of idea (that there can be two implementations of the same input-output behavior) is the basis for all of our security definitions. 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 $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ have the same input-output behavior if *no* calling program behaves differently when linking to $\mathcal{L}_{\text{left}}$ vs. $\mathcal{L}_{\text{right}}$. More formally,

Definition 2.2 (Interchangeable)
*Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be two libraries with a common interface. We say that $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are **interchangeable**, and write $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$, if for all programs $\mathcal{A}$ that output a single bit, $\Pr[A \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] = \Pr[A \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]$.*

Discussion

- 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:

$$\begin{aligned} & \Pr[A \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] = \Pr[A \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] \\ \Leftrightarrow \quad & 1 - \Pr[A \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] = 1 - \Pr[A \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] \\ \Leftrightarrow \quad & \Pr[A \diamond \mathcal{L}_{\text{left}} \Rightarrow 0] = \Pr[A \diamond \mathcal{L}_{\text{right}} \Rightarrow 0]. \end{aligned}$$

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

[1] This doesn't mean that it's impossible to reason about attacks where an adversary has implicit channels of information on our cryptographic implementations. It's just that such channels must be made *explicit as far as the definitions go*, if one wishes to prove something about what an adversary can learn by that channel. You can't reason about information that your definition has no way of expressing.

- We have restricted the calling program $\mathcal{A}$ to a single bit of output, which might seem unnecessarily restrictive. However, the definition says that the two libraries have the same effect on *all* calling programs. In particular, the libraries must have the same effect on a calling program $\mathcal{A}$ 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 $\mathcal{A}$ thinks it is linked to. For this reason, we will often refer to the calling program $\mathcal{A}$ as a **distinguisher**.

- Taking the previous observation even further, the definition applies against calling programs $\mathcal{A}$ 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."[2]

  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 libararies' 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 shows up in our definitions in that our definition specifies two libaries, *then* we consider a particular distinguisher, and *only then* 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.

  In the context of security definitions, think of Kerckhoffs' principle as: *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 values of privately-scoped variables within the library (which could be the result of random choices).*

- The definitions here have been chosen to ease gently into future concepts. We will eventually require libraries that have multiple subroutines and maintain state (via static variables) between different subroutine calls.

  Defining interchangeability in terms of distinguishers may not seem entirely natural, but it allows us to later have more granularity. Instead of requiring two libraries to have identical input-output behavior, we will be content with libraries that are "similar enough", and distinguishers provide a conceptually simple way to measure exactly how similar two libraries are.

The following lemma will be useful throughout the course as we deal with libraries in security proofs:

[2] *"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.

Lemma 2.3 (Composition) *If $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$ then $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$ for any library $\mathcal{L}^*$.*

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

$$\begin{aligned}
\Pr[\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_{\text{left}}) \Rightarrow 1] &= \Pr[(\mathcal{A} \diamond \mathcal{L}^*) \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] && \text{(change of perspective)} \\
&= \Pr[(\mathcal{A} \diamond \mathcal{L}^*) \diamond \mathcal{L}_{\text{right}} \Rightarrow 1] && \text{(since } \mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}\text{)} \\
&= \Pr[\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_{\text{right}}) \Rightarrow 1]. && \text{(change of perspective)}
\end{aligned}$$

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

What Does Any of This Have to do with Security Definitions?

Suppose two libraries $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are interchangeable: two libraries that are different but appearing 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 $\mathcal{L}_{\text{left}}$ or $\mathcal{L}_{\text{right}}$, and use it to act differently in those two cases. But such a calling program would contradict the fact that $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$.

This line of reasoning leads to:

> **The Central Principle of Security Definitions™:**
>
> *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 Principle 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.

2.2 One-time Secrecy for Encryption

We now have all the technical tools needed to revisit the security of one-time pad. First, we can restate Claim 1.3 in terms of libraries:

Claim 2.4 (OTP rule) *The following two libraries are interchangeable* (i.e., $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$)*:*

$\mathcal{L}_{\text{otp-real}}$
QUERY($m \in \{0,1\}^\lambda$):
$k \leftarrow \{0,1\}^\lambda$
return $k \oplus m$

$\mathcal{L}_{\text{otp-rand}}$
QUERY($m \in \{0,1\}^\lambda$):
$c \leftarrow \{0,1\}^\lambda$
return c

Note that the two libraries $\mathcal{L}_{\text{otp-}\star}$ indeed have the same interface. Claim 2.4 is that the two libraries are have the same input-output behavior.

Claim 2.4 is specific to one-time pad, and in the previous chapter we have argued why it has some relevance to security. However, it's important to also have a standard definition of security 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.

In this section, we develop such a general-purpose security definition for encryption. That means it's time to face the question,

what does it mean for an encryption scheme to be "secure?"

To answer that question, let's first consider a very simplistic scenario in which an eavesdropper sees the encryption of some plaintext (using some unspecified encryption scheme Σ). We will start with the following informal idea:

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

Our goal is to somehow formalize this property as a statement about interchangeable libraries. Working backwards from The Central Principle of Security Definitions™, suppose we had two libraries whose interface allowed the calling program to learn a ciphertext, and whose only internal difference was in the choice of plaintext that was encrypted. If those two libraries were interchangeable, then their common interface (seeing a ciphertext) would leak no information about the internal differences (the choice of plaintext).

Hence, we should consider two libraries that look something like:

QUERY(??):
$k \leftarrow \Sigma$.KeyGen
$c \leftarrow \Sigma$.Enc(k, m_L)
return c

and

QUERY(??):
$k \leftarrow \Sigma$.KeyGen
$c \leftarrow \Sigma$.Enc(k, m_R)
return c

.

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 not quite well-defined — it's not clear where these plaintexts m_L and m_R come from. Should they be fixed, hard-coded constants? Should they be chosen randomly?

A good approach here 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 — in particular, even if you knew that it was one of only two options, and even if you got to *choose* those two options.

Putting these ideas together, we obtain the following definition:

Definition 2.5 (One-time secrecy) *Let Σ be an encryption scheme. We say that Σ is **(perfectly) one-time secret** if $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$
$\underline{\text{QUERY}(m_L, m_R \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_L)$
return c

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
$\underline{\text{QUERY}(m_L, m_R \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_R)$
return c

This security notion is often called *perfect secrecy* in other sources.[3] The definition is deceptively simple, and we will explore some of its subtleties through some more examples.

2.3 Hybrid Proofs of Security

We will now show that one-time pad satisfies the new security definition. More precisely,

Theorem 2.6 *Let* OTP *denote the one-time pad encryption scheme (Construction 1.2). Then* OTP *has one-time secrecy. That is, $\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}^{\text{OTP}}_{\text{ots-R}}$.*

Given what we already know about one-time pad, it's not out of the question that we could simply "eyeball" the claim $\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}^{\text{OTP}}_{\text{ots-R}}$. Indeed, we have already shown that in both libraries the 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:

$$\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}^{OT}_{\text{ots-R}},$$

where $\mathcal{L}_{\text{hyb-1}}, \ldots, \mathcal{L}_{\text{hyb-4}}$ 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 $\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}^{\text{OTP}}_{\text{ots-R}}$.) This style of proof is called a **hybrid proof** and it will be the standard way to prove security throughout this course.

For the security of one-time pad, such a hybrid proof is likely overkill. But we use this opportunity to introduce the technique, since nearly every security proof we will see in this class will use the hybrid technique. Hybrid proofs have the advantage that it can be quite easy to justify that *adjacent* hybrids (e.g., $\mathcal{L}_{\text{hyb-}i}$ and $\mathcal{L}_{\text{hyb-}(i+1)}$) are interchangeable, so the method scales well even in proofs where the "endpoints" of the hybrid sequence are quite different.

[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.

Proof As described above, we will prove that

$$\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}^{OT}_{\text{ots-R}},$$

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

$\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$:

```
L^OTP_ots-L
QUERY(m_L, m_R ∈ {0,1}^λ):
  k ← {0,1}^λ
  c = k ⊕ m_L
  return c
```

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

$\mathcal{L}_{\text{hyb-1}}$:

```
QUERY(m_L, m_R ∈ {0,1}^λ):
  c = QUERY'(m_L)
  return c
```

◇

```
L_otp-real
QUERY'(m ∈ {0,1}^λ):
  k ← {0,1}^λ
  return k ⊕ m
```

Factoring out a block of statements into a subroutine does not affect the library's behavior. Note that the new subroutine is exactly the $\mathcal{L}_{\text{otp-real}}$ library from Claim 2.4 (with the subroutine name changed to avoid naming conflicts). This is no accident!

$\mathcal{L}_{\text{hyb-2}}$:

```
QUERY(m_L, m_R ∈ {0,1}^λ):
  c = QUERY'(m_L)
  return c
```

◇

```
L_otp-rand
QUERY'(m ∈ {0,1}^λ):
  c ← {0,1}^λ
  return c
```

$\mathcal{L}_{\text{otp-real}}$ has been replaced with $\mathcal{L}_{\text{otp-rand}}$. From Claim 2.4 along with the library composition lemma Lemma 2.3, this change has no effect on the library's behavior.

$\mathcal{L}_{\text{hyb-3}}$:

```
QUERY(m_L, m_R ∈ {0,1}^λ):
  c = QUERY'(m_R)
  return c
```

◇

```
L_otp-rand
QUERY'(m ∈ {0,1}^λ):
  c ← {0,1}^λ
  return c
```

The argument to QUERY′ has been changed from m_L to m_R. This has no effect on the library's behavior since 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 — $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$ and $\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$ — we see that they differ only in swapping m_L for m_R. There is (arguably) no *direct* way to argue that these values can be simply switched. However, the one-time pad rule (Claim 2.4) shows that $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$ in fact has the same behavior as a library $\mathcal{L}_{\text{hyb-2}}$ 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.

$\mathcal{L}_{\text{hyb-4}}$:

```
QUERY(m_L, m_R ∈ {0,1}^λ):
  c = QUERY'(m_R)
  return c
```

◇

$\mathcal{L}_{\text{otp-real}}$

```
QUERY'(m ∈ {0,1}^λ):
  k ← {0,1}^λ
  return k ⊕ m
```

$\mathcal{L}_{\text{otp-rand}}$ has been replaced with $\mathcal{L}_{\text{otp-real}}$. Again, this has no effect on the library's behavior, due to Claim 2.4. Note that the library composition lemma is being applied with respect to a different common $\mathcal{L}^*$ than before (now m_R instead of m_L is being used).

$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$:

$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$

```
QUERY(m_L, m_R ∈ {0,1}^λ):
  k ← {0,1}^λ
  c = k ⊕ m_R
  return c
```

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

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

Discussion

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

- Proving security amounts to showing that two particular libraries, say $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$, are interchangeable.
- To show this, we show a sequence of hybrid libraries, beginning with $\mathcal{L}_{\text{left}}$ and ending with $\mathcal{L}_{\text{right}}$. 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 something that uses a one-time-secret encryption Σ as one of its components, then we are allowed to replace $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ with $\mathcal{L}^{\Sigma}_{\text{ots-R}}$ as one of the steps in the hybrid proof.

2.4 Demonstrating Insecurity with Attacks

We have seen an example of proving the security of a construction. To show that a construction is *insecure*, we demonstrate an **attack**. An attack means a counterexample to the definition of security. Since we define security in terms of two interchangeable libraries,

an attack is a **distinguisher** (calling program) that behaves as differently as possible when linked to the two libraries.

Below is an example of an insecure construction:

Construction 2.7

$\mathcal{K} = \left\{ \begin{array}{c} \textit{permutations} \\ \textit{of} \{1, \ldots, \lambda\} \end{array} \right\}$
$\mathcal{M} = \{0,1\}^\lambda$
$\mathcal{C} = \{0,1\}^\lambda$

KeyGen:
$k \leftarrow \mathcal{K}$
return k

Enc(k, m):
for $i := 1$ to λ:
  $c_{k(i)} := m_i$
return $c_1 \cdots c_\lambda$

Dec(k, c):
for $i := 1$ to λ:
  $m_i := c_{k(i)}$
return $m_1 \cdots m_\lambda$

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

Claim 2.8 *Construction 2.7 does **not** have one-time secrecy.*

Proof Our goal is to construct a program $\mathcal{A}$ so that $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-L}} \Rightarrow 1]$ and $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-R}} \Rightarrow 1]$ are different, where Σ refers to Construction 2.7. 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 $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-L}} \Rightarrow 1]$ and $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-R}} \Rightarrow 1]$ 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 $\mathcal{L}_{\text{ots-}\star}$ libraries; in other words, we should expect the distinguisher to call the 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:

$\mathcal{A}$
$c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

.

Here is what it looks like when $\mathcal{A}$ is linked to $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ (we have filled in the details of Construction 2.7 in $\mathcal{L}^{\Sigma}_{\text{ots-L}}$):

$\mathcal{A}$
$c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$
QUERY(m_L, m_R):
  $k \leftarrow$ {permutations of $\{1, \ldots, \lambda\}$}
  for $i := 1$ to λ:
    $c_{k(i)} := (m_L)_i$
  return $c_1 \cdots c_\lambda$

.

We can see that m_L takes on the value 0^λ, 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:

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

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

$\mathcal{A}$
$c \leftarrow \text{QUERY}(0^\lambda, 1^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
QUERY(m_L, m_R):
  $k \leftarrow$ {permutations of $\{1, \ldots, \lambda\}$}
  for $i := 1$ to λ:
    $c_{k(i)} := (m_R)_i$
  return $c_1 \cdots c_\lambda$

.

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:

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

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 2.7 does not satisfy the definition of one-time secrecy. ■

Exercises

2.1. Consider the following encryption scheme:

$\mathcal{K} = \mathbb{Z}_n$	KeyGen:	Enc(k, m):	Dec(k, c):
$\mathcal{M} = \mathbb{Z}_n$	$k \leftarrow \mathbb{Z}_n$	return $(k + m)$ % n	??
$\mathcal{C} = \mathbb{Z}_n$	return k		

(a) Fill in the details of the Dec algorithm so that the scheme satisfies correctness.

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

★ 2.2. 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.

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

$\mathcal{K} = \mathbb{G}$
$\mathcal{M} = \mathbb{G}$
$\mathcal{C} = \mathbb{G}$

KeyGen:
$k \leftarrow \mathbb{G}$
return k

Enc(k, m):
return $k \otimes m$

Dec(k, c):
??

(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.

2.3. 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.

$\mathcal{K} = \mathbb{Z}_{10}$
$\mathcal{M} = \mathbb{Z}_{10}$
$\mathcal{C} = \mathbb{Z}_{10}$

KeyGen:
$k \leftarrow \mathbb{Z}_{10}$
return k

Enc(k, m):
return $k \times m \,\%\, 10$

2.4. Prove that if an encryption scheme Σ satisfies one-time secrecy, then $|\Sigma.\mathcal{K}| \geq |\Sigma.\mathcal{M}|$.

2.5. Let Σ denote an encryption scheme where $\Sigma.\mathcal{C} = \Sigma.\mathcal{M}$, and define Σ^2 to be the following **double-encryption** scheme:

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

KeyGen:
$k_1 \leftarrow \Sigma.\mathcal{K}$
$k_2 \leftarrow \Sigma.\mathcal{K}$
return (k_1, k_2)

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

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 Σ satisfies one-time secrecy, then so does Σ^2.

2.6. Let Σ denote an encryption scheme and define Σ^2 to be the following **encrypt-twice** scheme:

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

KeyGen:
$k_1 \leftarrow \Sigma.\mathcal{K}$
$k_2 \leftarrow \Sigma.\mathcal{K}$
return (k_1, k_2)

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

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

Prove that if Σ satisfies one-time secrecy, then so does Σ^2.

2.7. 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.