2 The Basics of Provable Security

Edgar Allan Poe was not only an author, but also a cryptography enthusiast. He once wrote, in a discussion on the state of the art in cryptography:[1]

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

This seems an accurate assessment of cryptography as it existed in 1841. Whenever someone would come up with an encryption method, someone else would inevitably find a way to break it, and the cat-and-mouse game would repeat again and again.

Modern 21st-century cryptography, however, is different. This book will introduce you to many schemes whose security we can **prove** in a very specific sense. The code-makers *can* win against the code-breakers.

The core concept that allows us to actually *prove* things about security is the *security definition*. It formalizes exactly what we mean by "security." In this chapter, we will start learning important skills that all revolve around security definitions: how to write them, how to understand & interpret them, how to prove security using the *hybrid technique*, and how to demonstrate insecurity using attacks against the security definition.

2.1 Generalizing and Abstracting One-Time Pad

Abstraction: What & Why?

In this course we will certainly learn about specific encryption schemes (like one-time pad). However, it is often necessary to talk about encryption (and other primitives) at a higher level of abstraction. For example, we will eventually be interested in building more complicated things, using encryption as a component. In such a situation, it is convenient to be able to say something like, "*any* encryption scheme, when combined with this other thing in this specific way, results in a system with security property X, as long as the encryption scheme satisfies property Y." By talking about encryption schemes in the abstract, we can ignore all of their insignificant details and focus on which properties are actually important (in this example, security property Y).

Abstraction also leads to modularity. Suppose you build a system that uses encryption scheme A as a component, but a new attack is later discovered against that scheme. If the design of the system has been abstracted well, then you might be able to swap encryption scheme A for some encryption scheme B, as long as scheme B satisfies all of the security requirements. In a system that was designed in a monolothic way (i.e., taking into account all specifics of all components), you might not be guaranteed that swapping scheme B for scheme A is safe.

[1] Edgar Allan Poe, "A Few Words on Secret Writing," *Graham's Magazine*, July 1841, v19.

Syntax & Correctness

Our goal in this chapter is to identify a good abstraction for encryption. You can think of this as answering the question: what properties of one-time pad are actually relevant (to a system that uses it as a component)? For example, is it fundamental that one-time pad uses XOR as its underlying operation, or are there other schemes that avoid XOR but are "just as good?"

In Chapter 1 we have already argued that any method of encryption should involve keys, should consist of 3 algorithms, and that decryption should recover the original message. These properties are straight-forward to abstract. The definition below does so, and introduces new terminology that we will use.

Definition 2.1 (Encryption syntax)

*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. Sometimes we refer to the entire scheme (all algorithms) by a single variable Σ. When we do so, we write Σ.KeyGen, Σ.Enc, Σ.Dec, $\Sigma.\mathcal{K}$, $\Sigma.\mathcal{M}$, and $\Sigma.\mathcal{C}$ to refer to its components.*

We call this a **syntax** definition because it specifies the "type signature" of all of the algorithms of an encryption scheme, but doesn't say anything about the functional behavior of those algorithms. For example, the syntax definition allows for KeyGen, Enc, Dec to always output a string of all zeroes (on every input), but this would not be a very useful encryption scheme. We clearly need to make some functional requirements on these algorithms.

Definition 2.2

*An encryption scheme Σ satisfies **correctness** if for all $k \in \Sigma.\mathcal{K}$ and all $m \in \Sigma.\mathcal{M}$,*

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

The definition is expressed in terms of a probability, because Enc *is allowed to be a randomized algorithm. In other words, decrypting a ciphertext, using the same key that was used for encryption, **always** results in the original plaintext.*

Notice that an encryption scheme defined by $\mathsf{Enc}(k, m) = m$ satisfies this correctness property (with an appropriately defined Dec), but is also quite uninteresting. In general, when a definition does not involve any adversarial behavior (like this one), we call it a **correctness** property. Only when a definition involves adversarial behavior do we call it a **security** property.

2.2 Towards an Abstract Security Definition

It is a relatively easy matter to formalize the syntax & correctness of encryption. Formalizing a security property is the hard part. Fortunately, we already have a head start from our discussion of one-time pad in Chapter 1. We showed a property that one-time pad satisfies, which we will refer to now as "Attempt #1" at a security definition:

Attempt 1 *For all* $m \in \{0,1\}^\lambda$*, the output of the following subroutine is uniformly distributed over* $\{0,1\}^\lambda$*:*

```
EAVESDROP(m ∈ {0,1}^λ):
  k ← {0,1}^λ
  c := k ⊕ m
  return c
```

.

This property is far too specific to one-time pad. What we really want is a general-purpose security definition that says something like *"An encryption scheme* Σ *is secure if . . . "* and refers to some behaviors of Σ.Enc and so on.

In this section we will slowly build up to such a general-purpose definition. Think back to how this property of one-time pad was motivated in Chapter 1. The EAVESDROP subroutine should take as input a plaintext, and give as output a ciphertext which is the result of encrypting the input with an appropriately sampled key. If we re-write the subroutine in terms of a totally generic encryption scheme Σ, we get:

```
EAVESDROP(m ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m)
  return c
```

.

We want to say that, for all inputs m, the output of this subroutine is uniformly distributed, but uniformly distributed *over what set?* In the case of one-time pad the output was uniform over $\{0,1\}^\lambda$, but $\{0,1\}^\lambda$ could refer to the plaintext space, key space, or ciphertext space! In an arbitrary encryption scheme, these three spaces may not all be the same, and we need our general-purpose security definition to specify which one. In this case we are talking about the output distribution of EAVESDROP, which is a distribution over *ciphertexts.* Hence:

Attempt 2 Σ *is "secure" if, for all* $m \in \Sigma.\mathcal{M}$*, the output of the following subroutine is uniformly distributed over* $\Sigma.C$ *:*

```
EAVESDROP(m ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m)
  return c
```

.

Adversaries as Distinguishers

Attempt #2 is actually a reasonable security definition, but it turns out that there is a more useful way to conceptualize it. First, let's re-frame it in terms of an explicit *comparison between the input-output behavior of two subroutines:*

Attempt 3 *Σ is "secure" if the following two implementation of an* EAVESDROP *subroutine have the same input-output behavior (i.e., on every input, both subroutines generate the same output distribution):*

```
EAVESDROP(m ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m)
  return c
```

;

```
EAVESDROP(m ∈ Σ.M):
  c ← Σ.C
  return c
```

.

It's now time for a very important conceptual leap. Suppose we play a game where you play the role of a calling program and I play the role of a subroutine called EAVESDROP. You can send me any input, and I will either run the left implementation of EAVESDROP or the right implementation, but I won't tell you which implementation I'm using. Is there anything you can do to figure out which implementation I am using? No! In fact, my choice of which implementation to use has *no effect on you at all.*

In particular, the choice of left/right implementation of EAVESDROP has no effect on your output. Suppose after playing this game you output a single bit $b \in \{0, 1\}$; think of this as a guess of which implementation I have chosen. Your choice to output 0 or 1 is a random variable since it might depend on your own random choices and the randomness in EAVESDROP. Then the following two probabilities are the same:

$$\begin{aligned}&\Pr[\text{you output 1 when I respond using left implementation of EAVESDROP}]\\ =\ &\Pr[\text{you output 1 when I respond using right implementation of EAVESDROP}].\end{aligned}$$

It turns out that this game ("guess which of these two subroutine implementations you are connected to") is a convenient way to **define** what it means for two subroutine implementations to have "identical input-output behavior," especially when the implementations involve randomized behavior. The implementations have identical behavior if *no calling program can tell them apart*; i.e., if no calling program can find a way to behave differently (in terms of its output distribution) in the presence of the two implementations.

Attempt 4 *Σ is "secure" if, for all calling programs $\mathcal{A}$, connecting $\mathcal{A}$ with either the left or right version of* EAVESDROP *(below) does not change the output probability of $\mathcal{A}$.*

```
EAVESDROP(m ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m)
  return c
```

;

```
EAVESDROP(m ∈ Σ.M):
  c ← Σ.C
  return c
```

.

This style of definition is a little strange at first, but it is the basis for the way we talk about security in the entire class. The entire remainder of this chapter is devoted to understanding this style in depth, but for now just keep in mind:

- The adversary is an arbitrary program that gets to choose plaintexts to send to an EAVESDROP subroutine, but doesn't know whether the response is coming from the left or right implementation given above.
- The adversary's only goal in life is to **distinguish** whether it is connected to the left or right implementation of EAVESDROP. In this case, "distinguishing" means behaving differently when receiving encryptions of chosen plaintexts vs. totally random ciphertext (unrelated to the chosen plaintext).

Critically Analyzing a Security Definition

In math, a definition can't really be "wrong," but it can be "not as useful as you hoped" or it can "fail to adequately capture your intuition" about the concept.

Security definitions are no different. Our attempt #4 above is a useful security definition. However, one can argue that it doesn't *quite* perfectly capture our intuition about security for encryption.

Let's discuss the pros/cons of this security definition. The way to do this is by considering some (possibly strange) encryption schemes and seeing whether they satisfy the definition. If the scheme seems intuitively "secure" but does not meet the security definition (or if it's intuitively "insecure" but does meet the definition), then we need to stop and think. Either our intuitions or the definition needs to be re-evaluated.

Suppose you are worried about detecting errors during the transmission of ciphertext.[2] In order to add some redunancy to the data, you might modify one-time pad so that it sends two copies of the ciphertext (so, a ciphertext twice as long as before):

Construction 2.3 (Doubled OTP)

$\mathcal{K} = \{0,1\}^\lambda$
$\mathcal{M} = \{0,1\}^\lambda$
$\mathcal{C} = \{0,1\}^{2\lambda}$

KeyGen:
$k \leftarrow \{0,1\}^\lambda$
return k

$\mathrm{Enc}(k, m \in \{0,1\}^\lambda)$:
$c' := k \oplus m$
$c := c' \| c'$
return c

$\mathrm{Dec}(k, c \in \{0,1\}^{2\lambda})$:
$c' :=$ first λ bits of c
return $k \oplus c'$

Intuitively, this new scheme is just as secure as original one-time pad. Think of it this way: duplicating the ciphertext in this way ($c \mapsto c\|c$) is something that an eavesdropper can imagine even when attacking original OTP. So if this is a dangerous thing to do, then an attacker could also do it while eavesdropping on OTP ciphertexts.

However, this doubled OTP does not satisfy the security definition attempt #4, because its ciphertexts are not uniformly distributed in $C = \{0,1\}^{2\lambda}$. More formally, the definition requires that the following two subroutine implementations have the same input-output behavior:

EAVESDROP(m):
$k \leftarrow \{0,1\}^\lambda$
$c' := k \oplus m$
return $c'\|c'$

;

EAVESDROP(m):
$c \leftarrow \{0,1\}^{2\lambda}$
return c

.

But we can write a calling program that behaves differently in the presence of these two implementations. In particular, if we want to know whether we are talking to the left or

[2] This is actually a security concern if the transmission errors are adversarial, but at least in this chapter this issue is out of scope in terms of security.

right implementation of EAVESDROP, we should just call EAVESDROP (the choice of input won't matter), and check whether the first half of the output equals the second half of the output:

$\mathcal{A}$:
$c :=$ EAVESDROP$(\mathtt{0}^\lambda)$
$L :=$ first half of c
$R :=$ second half of c
return $L \stackrel{?}{=} R$

When $\mathcal{A}$ is connected to the left implementation of EAVESDROP, the output c always has equal first/second halves, so $\Pr[\mathcal{A}$ outputs `true`$] = 1$.

When $\mathcal{A}$ is connected to the right implementation of EAVESDROP, c is chosen as a uniform element of $\{\mathtt{0},\mathtt{1}\}^{2\lambda}$. What is the probability that such a string has equal first/second halves? It is only $1/2^\lambda$. In this case, $\Pr[\mathcal{A}$ outputs `true`$] = 1/2^\lambda < 1$.

The output probability of $\mathcal{A}$ is different in the two cases, so $\mathcal{A}$ successfully *distinguishes* the implementations. The scheme does not satisfy the security definition.

You might be thinking, surely this can be fixed by redefining the ciphertext space as $C = \{2\lambda$-bit strings with identical first/second halves$\}$. This is a clever idea, and indeed it would work. The ciphertexts in this scheme are not uniform in $\{\mathtt{0},\mathtt{1}\}^{2\lambda}$ but they are uniform in this suggestion for C.

However, isn't it weird that the security of an encryption scheme should so crucially rest on how narrowly you define the set C of ciphertexts? When we change C, it really has no effect on the functional properties of KeyGen and Enc, which are the important algorithms here. I hope you will agree that this is a somewhat inelegant way of fixing the problem.

Chosen-Plaintext Attack Template

In Chapter 1, we reasoned about security in the following way: An adversary sees a sample of EAVESDROP(m), for some m that was chosen (somehow) by Alice. We argued that the eavesdropper gets no information about m is because it could sample from the same distribution without Alice's involvement, by choosing an arbitrary m' and running EAVESDROP(m').

Note that in this discussion, there is nothing particularly special about EAVESDROP(m) being the *uniform* distribution. The important property is that EAVESDROP(m) and EAVESDROP(m') are *the same distribution* for all $m, m' \in \mathcal{M}$. Although we didn't prove it, our "double OTP" has this property too (in this case the distribution is the uniform distribution over 2λ-bit strings that have identical first/second halves).

In a sense, our attempt #4 at a security definition was slightly **too strong.** It was demanding that EAVESDROP(m) was *uniform*, while the more important factor was that EAVESDROP(m) is the *same distribution for all m.* How can we make a formal definition that says this?

We *could* define a subroutine that generates encryptions of $m = \mathtt{000\cdots0}$, one that generates encryptions of $m = \mathtt{000\cdots1}$, and so on, and require that no calling program

can distinguish any two of them. But this approach gets out of hand quickly, with too many subroutines to list.

A better way is the following:

Attempt 5 *Σ is "secure" if, for all calling programs $\mathcal{A}$, connecting $\mathcal{A}$ with either the left or right version of* EAVESDROP *(below) does not change the output probability of $\mathcal{A}$.*

```
EAVESDROP( m_L, m_R ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m_L )
  return c
```

;

```
EAVESDROP( m_L, m_R ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m_R )
  return c
```

.

Note how the calling program chooses *two* plaintexts m_L and m_R as arguments to EAVESDROP, but each implementation of EAVESDROP ignores one of these arguments. In this style of definition, the adversary (calling program) is saying: *I want the left implementation to generate encryptions of my chosen m_L, and the right implementation to generate encryptions of my chosen m_R, and then I will attempt to distinguish whether I am talking to the left or right.*

The security definition considers *all possible* calling programs. That means it considers all possible strategies for choosing m_L and m_R. If there exist m_L, m_R that induce different ciphertext distributions, then a calling program that chooses those particular m_L, m_R should be able to distinguish between the left/right EAVESDROP variants (and therefore violate the security definition). On the other hand, if all plaintexts induce the same ciphertext distribution, then no choice of m_L, m_R would lead to different input/output behavior between the two EAVESDROP variants.

This style of security definition — where the calling program chooses two plaintext m_L, m_R and only one is encrypted — is the standard way in cryptography to model a **chosen-plaintext attack.** This may seem strange, since you probably think of Alice as the person who chooses what to encrypt, *not Eve.* A good way to think about this security definition is: 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! Of course, if in some real-world scenario an attacker had even less partial information or influence on the choice of plaintexts, it would only make an attack even harder.

2.3 Provable Security Fundamentals

So far, one of the main themes of this chapter is that two subroutines have identical input-output behavior if and only if no calling program can tell which one it is connected to. We now introduce some more formal notation and terminology surrounding this concept.

Libraries & Interfaces

Definition 2.4 (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.*

If $\mathcal{A}$ or $\mathcal{L}$ is a program that makes random choices, then the output of $\mathcal{A} \diamond \mathcal{L}$ is a random variable. It is often useful to consider probabilities like $\Pr[\mathcal{A} \diamond \mathcal{L} \Rightarrow \texttt{true}]$.

Example *Here are two libraries $\mathcal{L}_1, \mathcal{L}_2$ that we have considered before. They have a common interface:*

$\mathcal{L}_2$

EAVESDROP(m):
$k \leftarrow \{0,1\}^\lambda$
$c' := k \oplus m$
return $c' \| c'$

;

$\mathcal{L}_1$

EAVESDROP(m):
$c \leftarrow \{0,1\}^{2\lambda}$
return c

.

Here is a calling program $\mathcal{A}$ that we also considered before:

$\mathcal{A}$:
$c :=$ EAVESDROP(0^λ)
$L :=$ first half of c
$R :=$ second half of c
return $L \stackrel{?}{=} R$

Previously we argued that:

$$\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \mathit{true}] = 1,$$
$$\Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \mathit{true}] = 1/2^\lambda.$$

Example *A library can contain several subroutines and variables that are kept static between subroutine calls. For example, here is a simple library that picks a string s uniformly and allows the calling program to guess s.*

$\mathcal{L}$
$s \leftarrow \{0,1\}^\lambda$

RESET():
$s \leftarrow \{0,1\}^\lambda$

GUESS($x \in \{0,1\}^\lambda$):
return $x \stackrel{?}{=} s$

Our convention is that code outside of a subroutine (like the first line here) is run once at initialization time. Variables defined at initialization time (like s) are visible in all subroutine scopes.

Semantics & Scope

We will use a pseudocode to specify libraries, and most aspects of that pseudocode will (hopefully) be straight-forward and self-explanitory. But we will make one important assumption/axiom about the meaning of these programs & libraries:

> The **only** thing a calling program can do with a library is to call its subroutines (on any arguments of its choice) and receive the output of subroutines.

One important consequence of this is that we assume all variables in a library to be *privately scoped* to the library, so that the calling program cannot access them directly. For example, the calling program has no way of learning the s value in the previous example, apart from eventually guessing it via the GUESS subroutine. If we want a calling program to have access to some internal variables, we must explicitly add an "accessor" subroutine to the library.

This is where the analogy to a "real-world software library" breaks down somewhat. In real-world software, when a program is linked to a library there are sneaky 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.[3]

In this course, we use the libraries to precisely model what an attacker can do in some situation, and then reason about the consequences. It simply works out best if all the adversary's capabilities are *explicit* in the library. So it's best to think of the libraries more as *mathematical abstractions* than realistic software.

We can still use these libraries to reason about attacks where an adversary has side-channels of information on our cryptographic implementations. The only catch is 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 library you're reasoning about*, even if its purpose is to model a channel that is implicit in the real world.

Interchangeability

The usual question to ask about two libraries is whether they have the same input-output behavior. As you have seen, this question can be framed in terms of whether any calling program can behave differently when connected to the two libraries.

Definition 2.5 (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,* $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow 1] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow 1]$.

At the risk of insulting the reader's intelligence, some ways that two libraries can be interchangeable include:

- Their only difference happens in an *unreachable block of code*

[3] In my experience, students who are interested in cryptography are also the most likely to be interested in these kinds of side channels.

- Their only difference is the value they assign to a *variable that is never actually used*
- Their only difference is that one library *unrolls a loop* that occurs in the other library
- Their only difference is that one library *inlines a subroutine* that occurs in the other library

We can all agree that these differences clearly have no effect on the input/output behavior of the library, and therefore they will have no effect on any calling program. Still, each of these examples shows up in real security proofs in this book, sometimes in surprising ways.

Here are some more simple examples of interchangeable libraries that deal specifically with randomness:

Example *The following two libraries are interchangeable. This example is essentially stating that, in the uniform distribution on* $\{0,1\}^{n+m}$*, each of the individual bits is distributed* independently *of the others.*

```
SAMPLE():
  x ← {0,1}^n
  y ← {0,1}^m
  return x‖y
```

```
SAMPLE():
  z ← {0,1}^{n+m}
  return z
```

Example *The following two libraries are interchangeable. The library on the left samples* s *"eagerly" — as soon as it can. The library on the right samples* s *"lazily" — only at the last possible moment.*

```
s ← {0,1}^n

GET():
  return s
```

```
GET():
  if s not defined:
    s ← {0,1}^n
  return s
```

We define interchangeability in terms of calling programs that produce only a single bit of output. You might think this is strange or somehow 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**.

Similarly, 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:

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

Our first examples of libraries are all very simple, consisting of just a single subroutine, but our definitions allow for more complicated libraries that have several subroutines and

persistent state between the subroutines (*i.e.*, static variables). These additional features will become important.

Looking even farther ahead, we will eventually consider libraries that do not have exactly identical input-output behavior, but which are only "similar enough." Because we have defined the similarity of libraries in terms of distinguishers (calling programs), these advanced definitions will still have mostly the same structure. The major difference is that these definitions will allow the libraries to alter the calling program's behavior by a *very small* amount.

Security Definitions, Using New Terminology

We can re-state some of our previous concepts using this new terminology. Our first observation specifically about one-time pad (attempt #1 at a security definition) can be written in terms of interchangeable libraries:

Claim 2.6 (OTP rule) *The following two libraries are interchangeable (*i.e., $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$*):*

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

EAVESDROP($m \in \{0,1\}^\lambda$):
  $k \leftarrow \{0,1\}^\lambda$
  return $k \oplus m$

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

EAVESDROP($m \in \{0,1\}^\lambda$):
  $c \leftarrow \{0,1\}^\lambda$
  return c

This specific property of one-time pad is sometimes useful. For more abstract security definitions which are not so closely tied to one-time pad, we previously settled on attempts #4 and #5 at a definition. These can be translated into the new terminology as:

Definition 2.7 (Uniform ctxts) *Let Σ be an encryption scheme. We say that Σ has **one-time uniform ciphertexts** if $\mathcal{L}^{\Sigma}_{\text{ots\$-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$

CTXT($m \in \Sigma.\mathcal{M}$):
  $k \leftarrow \Sigma.\text{KeyGen}$
  $c \leftarrow \Sigma.\text{Enc}(k, m)$
  return c

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$

CTXT($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.C$
  return c

Throughout this course, we will use the "$" symbol to denote something that random (or pseudorandom, as we will see).[4]

Definition 2.8 (One-time secrecy) *Let Σ be an encryption scheme. We say that Σ has **one-time secrecy** if $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$, where:*

[4] It is quite common in CS literature to use the "$" symbol when referring to randomness. This stems from thinking of randomized algorithms as algorithms that "toss coins." Randomized algorithms need to have spare change (i.e., money) sitting around. By convention, that spare change is in US dollars.

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$

EAVESDROP($m_L, m_R \in \Sigma.\mathcal{M}$):
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$c \leftarrow \Sigma.\mathsf{Enc}(k, m_L)$
return c

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

EAVESDROP($m_L, m_R \in \Sigma.\mathcal{M}$):
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$c \leftarrow \Sigma.\mathsf{Enc}(k, m_R)$
return c

Discussion, Pitfalls

It is a common pitfall to imagine the program $\mathcal{A}$ being *simultaneously* linked to both libraries, but this is not what the definition says. The definition of $\mathcal{L}_1 \equiv \mathcal{L}_2$ refers to two different executions: one where $\mathcal{A}$ is linked only to $\mathcal{L}_1$ for its entire lifetime, and one where $\mathcal{A}$ is linked only to $\mathcal{L}_2$ for its entire lifetime. There is never a time where some of $\mathcal{A}$'s subroutine calls are answered by $\mathcal{L}_1$ and others by $\mathcal{L}_2$. This is an especially important distinction when $\mathcal{A}$ makes several subroutine calls in a single execution.

Another common pitfall is confusion about the difference between the algorithms of an encryption scheme (*e.g.*, what is shown in Construction 1.1) and the libraries used in a security definition (*e.g.*, what is shown in Definition 2.8). The big difference is:

- The algorithms of the scheme show a regular user's view of things. A user can encrypt/decrypt anything they want, and do anything they want with the results. The KeyGen, Enc, Dec algorithms show how this is done.
- The libraries capture the attacker's view of things. In particular they specify the *attacker's influence over the victim's use of the algorithms* — this is the attack scenario being considered. For example, one-time secrecy considers an attacker that can compel a victim to encrypt a given plaintext and show the resulting ciphertext. But the attacker can't compel a victim to simply reveal its secret key (the library gives no way to do this). The attacker can't compel a victim to encrypt two plaintexts under the same key.

So, as a user of the cryptographic scheme, **don't** interpret one-time secrecy to mean "I'm not allowed to choose what to encrypt, I have to ask the adversary to choose for me." Instead, think "If I encrypt only one plaintext per key, then I am safe to encrypt things even if the attacker sees the resulting ciphertext and even if she has some influence or partial information on what I'm encrypting, because this is the situation captured in the one-time secrecy library."

Kerckhoffs' Principle

We have previously discussed Kerckhoffs' Principle, which says to assume that the adversary has full knowledge of the algorithms, and only lacks knowledge about the choice of keys. Let's see how Kerckhoffs' Principle is reflected in our style of security definitions.

Most importantly, the definition of interchangeability considers *all* calling programs. In particular, this includes calling programs that "know everything" about (more formally, whose code is allowed to depend arbitrarily on) the two libraries. Or, in other words, the definition considers calling programs that are specially designed to distinguish these two particular libraries.

There is, however, a subtlety that deserves some careful attention. The calling program does *not* know the values of privately scoped variables inside the library. This is an important distinction when these variables are assigned in a randomized way. Take for example the simple library from Claim 2.6:

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

EAVESDROP($m \in \{0,1\}^\lambda$):
  $k \leftarrow \{0,1\}^\lambda$
  return $k \oplus m$

The adversary can know that it might be linked to this library, and it can know that this library includes a statement "$k \leftarrow \{0,1\}^\lambda$." But since k is privately scoped, the adversary has no direct way of knowing the *specific value* of k in a given execution. And indeed, the other library $\mathcal{L}_{\text{otp-rand}}$ doesn't even have a variable named k!

This is like the difference between knowing that you will choose a random card from a deck (*i.e.*, knowing what algorithm you will run to choose a card) versus reading your mind to know exactly what card you chose. Or the difference between knowing that you will choose a λ-bit key versus knowing what your key is.

This subtlety is reflected in Definition 2.5 in the following way. First, we specify two libaries, *then* we consider a particular distinguisher, and *only then* do we link and execute the distinguisher with a library. The algorithm that defines the distinguisher cannot depend on the library's random choices made in a particular execution, since those random choices "happen after" the choice of distinguisher is fixed.

In summary:

Kerckhoffs' Principle, in our terminology:

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

1. *which of the two possible 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).*

2.4 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.

Chaining Several Components

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 $\mathcal{A}$. Depending on the context, it can sometimes be convenient to interpret $\mathcal{A} \diamond \mathcal{L}_1 \diamond \mathcal{L}_2$ as:

- $(\mathcal{A} \diamond \mathcal{L}_1) \diamond \mathcal{L}_2$: a **compound calling program** linked to $\mathcal{L}_2$. After all, $\mathcal{A} \diamond \mathcal{L}_1$ is a program that makes calls to the interface of $\mathcal{L}_2$.
- or: $\mathcal{A} \diamond (\mathcal{L}_1 \diamond \mathcal{L}_2)$: $\mathcal{A}$ linked to a **compound library**. After all, $\mathcal{A}$ is a program that makes calls to the interface of $(\mathcal{L}_1 \diamond \mathcal{L}_2)$.

The placement of the parentheses does not affect the functionality of the overall program, just like how splitting up a real program into different source files doesn't affect its functionality.

In fact, every security proof in this book will have some intermediate steps that involve compound libraries. We will make heavy use of the following helpful result:

Lemma 2.9 (Chaining) *If $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$ then, for any library $\mathcal{L}^*$, we have $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$.*

Proof Note that we are comparing $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}}$ and $\mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$ as compond libraries. Hence we consider a calling program $\mathcal{A}$ that is linked to either $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}}$ or $\mathcal{L} * \diamond \mathcal{L}_{\text{right}}$.

Let $\mathcal{A}$ be such an arbitrary calling program. We must to show that $\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_{\text{left}})$ and $\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_{\text{right}})$ have identical output distribution. As mentioned above, we can interpret $\mathcal{A} \diamond \mathcal{L}^* \diamond \mathcal{L}_{\text{left}}$ as a calling program $\mathcal{A}$ linked to the library $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}}$, but also as a calling program $\mathcal{A} \diamond \mathcal{L}^*$ linked to the library $\mathcal{L}_{\text{left}}$. 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 the compound program $\mathcal{A} \diamond \mathcal{L}^*$. 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. ∎

One-Time Secrecy of One-Time Pad

We now introduced two security definitions to consider:

- One-time uniform ciphertexts (Definition 2.7), which states that ciphertexts should be uniformly distributed in $\Sigma.C$.
- One-time secrecy (Definition 2.8), which states that all m result in the same ciphertext distribution (but that distribution need not be *uniform*).

In the previous chapter, we have actually proved that one-time pad satisfies the first definition (although we didn't use the terminology of interchangeable libraries). Let us use that fact to show that one-time pad also satisfies the one-time secrecy property.

In fact, let's not limit ourselves to one-time pad. Let's instead show something slightly more general than that:

Theorem 2.10 *Let Σ be an encryption scheme. If Σ has one-time uniform ciphertexts (Definition 2.7), then Σ also has one-time secrecy (Definition 2.8). In other words:*

$$\mathcal{L}^{\Sigma}_{\text{ots\$-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ots\$-rand}} \implies \mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}.$$

If you are comfortable with what all the terminology means, then the meaning of the proof is quite simple and unsurprising. If all plaintexts m result in a *uniform* distribution of ciphertexts, then all m result in the *same* distribution of ciphertexts.

It may seem a bit overkill to actually prove this theorem. But proving it slowly, step-by-step gives us a chance to see the structure of security proofs in this course.

Proof We must show that $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$. Instead of directly comparing these two libraries, we will show that:

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

where $\mathcal{L}_{\text{hyb-1}}, \ldots, \mathcal{L}_{\text{hyb-4}}$ are a sequence of what we call **hybrid** libraries that we choose. It is not hard to see that the "$\equiv$" relation is transitive, so this proves that $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$. This proof technique is called the **hybrid technique.**

We are allowed to use the fact that $\mathcal{L}^{\Sigma}_{\text{ots\$-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$. What this means in terms of the proof is that if we ever see an instance of $\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$ show up (*e.g.*, it will appear as part of $\mathcal{L}_{\text{hyb-1}}$), then we can replace it with $\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$. That change will have no effect on the caling program.

We will now show the hybrid libraries that result in:

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

For each library, we highlight the differences from the previous one, and argue why adjacent hybrids are interchangeable.

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

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$

EAVESDROP($m_L, m_R \in \Sigma.\mathcal{M}$):
- $k \leftarrow \Sigma.\text{KeyGen}$
- $c \leftarrow \Sigma.\text{Enc}(k, m_L)$
- return c

As promised, the hybrid sequence begins with $\mathcal{L}^{\Sigma}_{\text{ots-L}}$.

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

EAVESDROP($m_L, m_R \in \Sigma.\mathcal{M}$):
- $c :=$ CTXT(m_L)
- return c

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$

CTXT($m \in \Sigma.\mathcal{M}$):
- $k \leftarrow \Sigma.\text{KeyGen}$
- $c \leftarrow \Sigma.\text{Enc}(k, m)$
- return c

Factoring out a block of statements into a subroutine makes it possible to write the library as a *compound* one, but does not affect its external behavior. Note that the new subroutine is exactly the $\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$ library from Definition 2.7. This was a strategic choice, because of what happens next.

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

EAVESDROP($m_L, m_R \in \Sigma.\mathcal{M}$):
- $c :=$ CTXT(m_L)
- return c

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$

CTXT($m \in \Sigma.\mathcal{M}$):
- $c \leftarrow \Sigma.\mathcal{C}$
- return c

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$ has been replaced with $\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$. The chaining lemma Lemma 2.9 says that this change has no effect on the library's behavior, since the two $\mathcal{L}_{\text{ots\$-}\star}$ libraries are interchangeable.

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

```
EAVESDROP(m_L, m_R ∈ Σ.M):
  c := CTXT( m_R )
  return c
```

◇

```
L^Σ_ots$-rand
CTXT(m ∈ Σ.M):
  c ← Σ.C
  return c
```

The argument to CTXT has been changed from m_L to m_R. This has no effect on the library's behavior since CTXT *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 — $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ and $\mathcal{L}^{\Sigma}_{\text{ots-R}}$ — 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 (*i.e.*, why it has no effect on the calling program). However, the uniform ciphertexts property shows that $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ 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.

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

```
EAVESDROP(m_L, m_R ∈ Σ.M):
  c := CTXT(m_R)
  return c
```

◇

```
L^Σ_ots$-real
CTXT(m ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m)
  return c
```

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$ has been replaced with $\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$. This is another application of the chaining lemma.

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

```
L^Σ_ots-R
EAVESDROP(m_L, m_R ∈ Σ.M):
  k ← Σ.KeyGen
  c ← Σ.Enc(k, m_R)
  return c
```

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

Putting everything together, we showed that $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$. This completes the proof, and we conclude that Σ 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. All security proofs in this book use this technique.

- Proving security means showing that two particular libraries, say $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$, are interchangeable.
- Often $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are significantly different, making them hard to compare directly. To make the comparison more manageable, we can show a sequence of hybrid

libraries, beginning with $\mathcal{L}_{\text{left}}$ and ending with $\mathcal{L}_{\text{right}}$. The idea is to break up the large "gap" between $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ into smaller ones that are easier to justify.

- It is helpful to think of "starting" at $\mathcal{L}_{\text{left}}$, and then making a sequence of small modifications to it, with the goal of eventually reaching $\mathcal{L}_{\text{right}}$ as a result of those modifications. With each modification you should justify why it doesn't affect the calling program (*i.e.*, why the two libraries before/after your modification are interchangeable).
- As discussed in Section 2.3, simple things like inlining/factoring out subroutines, changing unused variables, changing unreachable statements, or unrolling loops are always "allowable" modifications in a hybrid proof since they don't affect the calling program. As we progress in the course, we will see more kinds of useful modifications.
- Most proofs in this course are *conditional*, so they have the form "if A is a secure X, then B is a secure Y." In these proofs, the "X-security of A" gives us another allowable modification we can use in the sequence of hybrids. For example, in the previous proof, we were allowed to use the fact that $\mathcal{L}_{\text{ots\$-real}} \equiv \mathcal{L}_{\text{ots\$-rand}}$, and we did so twice (to show $\mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}}$ and $\mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}}$).

2.5 How to Demonstrate Insecurity with Attacks

We have seen an example of how to prove security properies about encryption schemes. To show that a scheme is *insecure*, we just have to show that the two relevant libraries are *not* interchangeable. To do that, we have to find *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 2.11

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

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 $m_1 \cdots m_\lambda$

This scheme encrypts a plaintext by simply rearranging its bits according to the secret permutation k.

Claim 2.12 *Construction 2.11 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 2.11. There are probably many "reasons" why

this construction is insecure, each of which leads to a different distinguisher $\mathcal{A}$. We need only to 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, the ciphertext preserves (leaks) the number of `0`s and `1`s in the plaintext. By counting the number of `0`s and `1`s in the ciphertext, we know exactly how many `0`s and `1`s 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 EAVESDROP 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 `0`s and `1`s. An extreme example (and why not be extreme?) would be to choose $m_L = \texttt{0}^\lambda$ and $m_R = \texttt{1}^\lambda$. 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:

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

.

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 2.11 in $\mathcal{L}^{\Sigma}_{\text{ots-L}}$):

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

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$
EAVESDROP(m_L, m_R):
$k \leftarrow \{\text{permutations of } \{1, \ldots, \lambda\}\}$
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 $\texttt{0}^\lambda$, so each bit of m_L is `0`, and each bit of c is `0`. Hence, the final output of $\mathcal{A}$ is always 1 (true):

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

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

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

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
EAVESDROP(m_L, m_R):
$k \leftarrow \{\text{permutations of } \{1, \ldots, \lambda\}\}$
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 always 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 2.11 does **not** satisfy the definition of one-time secrecy. ■

Exercises

2.1. Below are two calling programs $\mathcal{A}_1, \mathcal{A}_2$ and two libraries $\mathcal{L}_1, \mathcal{L}_2$ with a common interface:

$\mathcal{A}_1$
$r_1 \leftarrow \text{RAND}(6)$
$r_2 \leftarrow \text{RAND}(6)$
return $r_1 \stackrel{?}{=} r_2$

$\mathcal{A}_2$
$r \leftarrow \text{RAND}(6)$
return $r \stackrel{?}{\geqslant} 3$

$\mathcal{L}_1$
RAND(n):
$r \leftarrow \mathbb{Z}_n$
return r

$\mathcal{L}_2$
RAND(n):
return 0

(a) What is $\Pr[\mathcal{A}_1 \diamond \mathcal{L}_1 \Rightarrow 1]$?

(b) What is $\Pr[\mathcal{A}_1 \diamond \mathcal{L}_2 \Rightarrow 1]$?

(c) What is $\Pr[\mathcal{A}_2 \diamond \mathcal{L}_1 \Rightarrow 1]$?

(d) What is $\Pr[\mathcal{A}_2 \diamond \mathcal{L}_2 \Rightarrow 1]$?

2.2. In each problem, a pair of libraries are described. State whether or not $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$. If so, show how they assign identical probabilities to all outcomes. If not, then describe a successful *distinguisher*.

Assume that both libraries use the same value of n. Does your answer ever depend on the choice of n?

In part (a), $\overline{x}$ denotes the bitwise-complement of x. In part (d), $x \,\&\, y$ denotes the bitwise-AND of the two strings:

(a)

$\mathcal{L}_{\text{left}}$
QUERY() :
$x \leftarrow \{0,1\}^n$
return x

$\mathcal{L}_{\text{right}}$
QUERY() :
$x \leftarrow \{0,1\}^n$
$y := \overline{x}$
return y

(b)

$\mathcal{L}_{\text{left}}$
QUERY() :
$x \leftarrow \mathbb{Z}_n$
return x

$\mathcal{L}_{\text{right}}$
QUERY() :
$x \leftarrow \mathbb{Z}_n$
$y := 2x \,\%\, n$
return y

(c)

$\mathcal{L}_{\text{left}}$
QUERY($c \in \mathbb{Z}_n$) :
if $c = 0$
return null
$x \leftarrow \mathbb{Z}_n$
return x

$\mathcal{L}_{\text{right}}$
QUERY($c \in \mathbb{Z}_n$) :
if $c = 0$
return null
$x \leftarrow \mathbb{Z}_n$
$y := cx \,\%\, n$
return y

(d)

$\mathcal{L}_{\text{left}}$
QUERY() :
$x \leftarrow \{0,1\}^n$
$y \leftarrow \{0,1\}^n$
return $x \,\&\, y$

$\mathcal{L}_{\text{right}}$
QUERY() :
$z \leftarrow \{0,1\}^n$
return z

2.3. Show that the following libraries are interchangeable:

$\mathcal{L}_{\text{left}}$
QUERY$(m \in \{0,1\}^\lambda)$:
$x \leftarrow \{0,1\}^\lambda$
$y := x \oplus m$
return (x, y)

$\mathcal{L}_{\text{right}}$
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.

2.4. Show that the following libraries are **not** interchangeable. Describe an explicit distinguishing calling program, and compute its output probabilities when linked to both libraries:

$\mathcal{L}_{\text{left}}$
EAVESDROP$(m_L, m_R \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m_L$
return (k, c)

$\mathcal{L}_{\text{right}}$
EAVESDROP$(m_L, m_R \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m_R$
return (k, c)

★ 2.5. 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 $e \otimes a = a \otimes e = a$ 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* $(\mathbb{G}, \otimes)$:

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

(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.

2.6. Suppose we modify one-time pad to add a few 0 bits to the end of every ciphertext:

$\mathcal{K} = \{0,1\}^\lambda$ $\mathcal{M} = \{0,1\}^\lambda$ $C = \{0,1\}^{\lambda+2}$	KeyGen: $k \leftarrow \{0,1\}^\lambda$ return k	Enc(k, m): $c := k \oplus m$ return $c \| 00$	Dec(k, c): remove last 2 bits of c $m := k \oplus c$ return m

(In Enc, "||" refers to concatenation of strings.) Show that the resulting scheme still satisfies one-time secrecy. Your proof can use the fact that one-time pad has one-time secrecy.

2.7. The text showed that the uniform ciphertext property implies one-time secrecy, *i.e.*:

$$\mathcal{L}^\Sigma_{\text{ots\$-real}} \equiv \mathcal{L}^\Sigma_{\text{ots\$-rand}} \implies \mathcal{L}^\Sigma_{\text{ots-L}} \equiv \mathcal{L}^\Sigma_{\text{ots-R}}.$$

Show that the converse is **not** true. That is,

$$\mathcal{L}^\Sigma_{\text{ots-L}} \equiv \mathcal{L}^\Sigma_{\text{ots-R}}. \not\implies \mathcal{L}^\Sigma_{\text{ots\$-real}} \equiv \mathcal{L}^\Sigma_{\text{ots\$-rand}}$$

Give an example of an encryption scheme Σ which has one-time secrecy but *not* uniform ciphertexts.

Hint: Such a scheme has already been discussed.

2.8. 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.

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

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

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

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 $m = 1101100001$.

$$\begin{aligned} k &= 1\_\_0\_\_11010\_1\_0\_0\_\_\_ \\ m &= \ \ 11\ 01\ \ \ \ \ 1\ 0\ 0\ 001 \\ \implies \text{Enc}(k, m) &= 11100111010110000001 \end{aligned}$$

2.10. Suppose we modify the scheme from the previous problem to first permute the bits of m (as in Construction 2.11) 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 2λ characters long with λ "_" characters.

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

2.11. Prove that if an encryption scheme Σ 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 (*i.e.*, a distinguisher against the appropriate libraries).

In one-time pad, Enc is a deterministic function but **for full credit**, you should prove the statement even if Enc is randomized. However, you may assume that Dec is deterministic.

Hint: The definition of interchangeabilitiy doesn't care about the running time of the distinguisher/calling program. So even an exhaustive brute-force attack would be valid.

2.12. Let Σ denote an encryption scheme where $\Sigma.C \subseteq \Sigma.\mathcal{M}$ (so that it is possible to use the scheme to encrypt its own ciphertexts). Define Σ^2 to be the following **nested-encryption** 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)

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

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

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

2.13. 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)

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)

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.

2.14. Prove that an encryption scheme Σ satisfies one-time secrecy **if and only if** the following two libraries are interchangeable:

$\mathcal{L}^{\Sigma}_{\text{left}}$

CTXT$(m \in \Sigma.\mathcal{M})$:
  $k \leftarrow \Sigma.\mathsf{KeyGen}$
  $c \leftarrow \Sigma.\mathsf{Enc}(k, m)$
  return c

$\mathcal{L}^{\Sigma}_{\text{right}}$

CTXT$(m \in \Sigma.\mathcal{M})$:
  $k \leftarrow \Sigma.\mathsf{KeyGen}$
  $m' \leftarrow \Sigma.\mathcal{M}$
  $c \leftarrow \Sigma.\mathsf{Enc}(k, m')$
  return c

Note: you have to prove both directions!

2.15. 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.

2.16. In this problem we consider modifying one-time pad so that the key is not chosen uniformly. Let $\mathcal{D}_\lambda$ denote the probability distribution over $\{\texttt{0},\texttt{1}\}^\lambda$ where we choose each bit of the result to be `0` with probability 0.4 and `1` with probability 0.6.

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

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

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