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 cipher which human ingenuity cannot resolve."*

This was an accurate assessment of the cryptography that 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.

It's only possible to *prove* things about security by having *formal definitions* of what it means to be "secure." This chapter is about the fundamental skills that 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 How to Write a Security Definition

So far the only form of cryptography we've seen is one-time pad, so our discussion of security has been rather specific to one-time pad. It would be preferable to have a vocabulary to talk about security in a more general sense, so that we can ask whether *any* encryption scheme is secure.

In this section, we'll develop two security definitions for encryption.

What *Doesn't* Go Into a Security Definition

A security definition should give guarantees about what can happen to a system in the presence of an attacker. But not all important properties of a system refer to an attacker. For encryption specifically:

- We don't reference any attacker when we say that the Enc algorithm takes two arguments (a key and a plaintext), or that the KeyGen algorithm takes no arguments. Specifying the types of inputs/outputs (*i.e.*, the "function signature") of the various algorithms is therefore not a statement about security. We call these properties the **syntax** of the scheme.

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

- Even if there is no attacker, it's still important that decryption is an inverse of encryption. This is not a security property of the encryption scheme. Instead, we call it a **correctness** property.

Below are the generic definitions for syntax and correctness of symmetric-key encryption:

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

Definition 2.2 (SKE correctness)

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

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

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

Example

An encryption scheme can have the appropriate syntax but still have degenerate behavior like $\mathsf{Enc}(k, m) = \mathtt{0}^\lambda$ (i.e.*, every plaintext is "encrypted" to* $\mathtt{0}^\lambda$*). Such a scheme would not satisfy the correctness property.*

A different scheme defined by $\mathsf{Enc}(k, m) = m$ *(*i.e.*, the "ciphertext" is always equal to the plaintext itself) and* $\mathsf{Dec}(k, c) = c$ *does satisfy the correctness property, but would not satisfy any reasonable security property.*

"Real-vs-Random" Style of Security Definition

Let's try to make a security definition that formalizes the following intuitive idea:

> *"an encryption scheme is a good one if its ciphertexts look like random junk to an attacker."*

Security definitions always consider **the attacker's view** of the system. What is the "interface" that Alice & Bob expose to the attacker by their use of the cryptography, and does that particular interface benefit the attacker?

In this example, we're considering a scenario where the attacker gets to observe ciphertexts. How *exactly* are these ciphertexts generated? What are the inputs to Enc (key and plaintext), and how are they chosen?

- **Key:** It's hard to imagine any kind of useful security if the attacker knows the key. Hence, we consider that the key is kept secret from the attacker. Of course, the key is generated according to the KeyGen algorithm of the scheme.

  At this point in the course, we consider encryption schemes where the key is used to encrypt only one plaintext. Somehow this restriction must be captured in our security definition. Later, we will consider security definitions that consider a key that is used to encrypt many things.

- **Plaintext:** It turns out to be useful to consider that the attacker actually *chooses* the plaintexts. This a "pessimistic" choice, since it gives much power to the attacker. However, if the encryption scheme is indeed secure when the attacker chooses the plaintexts, then it's also secure in more realistic scenarios where the attacker has some uncertainty about the plaintexts.

These clarifications allow us to fill in more specifics about our informal idea of security:

> *"an encryption scheme is a good one if its ciphertexts look like random junk to an attacker . . . when each key is secret and used to encrypt only one plaintext, even when the attacker chooses the plaintexts."*

A concise way to express all of these details is to consider **the attacker as a calling program** to the following subroutine:

$$\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\mathsf{KeyGen}\\ \quad c := \Sigma.\mathsf{Enc}(k,m)\\ \quad \text{return } c\end{array}}\,.$$

A calling program can choose the argument to the subroutine (in this case, a plaintext), and see *only* the resulting return value (in this case, a ciphertext). The calling program *can't* see values of privately-scoped variables (like k in this case). If the calling program makes many calls to the subroutine, a fresh key k is chosen each time.

The interaction between an attacker (calling program) and this ctxt subroutine appears to capture the relevant scenario. We would like to say that the outputs from the ctxt subroutine are uniformly distributed. A convenient way of expressing this property is to say that this ctxt subroutine should have the same effect on *every* calling program as a ctxt subroutine that (explicitly) samples its output uniformly.

$$\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\mathsf{KeyGen}\\ \quad c := \Sigma.\mathsf{Enc}(k,m)\\ \quad \text{return } c\end{array}} \quad \text{vs.} \quad \boxed{\begin{array}{l}\underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}\\ \quad c \leftarrow \Sigma.\mathcal{C}\\ \quad \text{return } c\end{array}}\,.$$

Intuitively, no calling program should have any way of determining which of these two implementations is answering subroutine calls. As an analogy, one way of saying that "foo is a correct sorting algorithm" is to say that "no calling program would behave differently if foo were replaced by an implementation of mergesort."

In summary, we can define security for encryption in the following way:

> *"an encryption scheme is a good one if, when you plug its* KeyGen *and* Enc *algorithms into the template of the* CTXT *subroutine above, the two implementations of* CTXT *induce identical behavior in every calling program."*

In a few pages, we introduce formal notation and definitions for the concepts introduced here. In particular, both the calling program and subroutine can be randomized algorithms, so we should be careful about what we mean by "identical behavior."

Example *One-time pad is defined with* KeyGen *sampling k uniformly from $\{0,1\}^\lambda$ and* $\textsf{Enc}(k,m) = k \oplus m$. *It satisfies our new security property since, when we plug in this algorithms into the above template, we get the following two subroutine implementations:*

$$\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k \leftarrow \{0,1\}^\lambda \quad \text{// KeyGen } \textit{of OTP}\\ \quad c := k \oplus m \quad \text{// Enc } \textit{of OTP}\\ \quad \text{return } c\end{array}} \quad \textit{vs.} \quad \boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad c \leftarrow \{0,1\}^\lambda \quad \textit{// C of OTP}\\ \quad \text{return } c\end{array}},$$

and these two implementations have the same effect on all calling programs.

"Left-vs-Right" Style of Security Definition

Here's a different intuitive idea of security:

> *"an encryption scheme is a good one if encryptions of m_L look like encryptions of m_R to an attacker (for all possible m_L, m_R)"*

As above, we are considering a scenario where the attacker sees some ciphertext(s). These ciphertexts are generated with some key; where does that key come from? These ciphertexts encrypt either some m_L or some m_R; where do m_L and m_R come from? We can answer these questions in a similar way as the previous example. Plaintexts m_L and m_R can be chosen by the attacker. The key is chosen according to KeyGen so that it remains secret from the attacker (and is used to generate only one ciphertext).

> *"an encryption scheme is a good one if encryptions of m_L look like encryptions of m_R to an attacker, when each key is secret and used to encrypt only one plaintext, even when the attacker chooses m_L and m_R."*

As before, we formalize this idea by imagining the attacker as a program that calls a particular interface. This time, the attacker will choose **two** plaintexts m_L and m_R, and get a ciphertext in return.[2] Depending on whether m_L or m_R is actually encrypted, those interfaces are implemented as follows:

$$\boxed{\begin{array}{l}\underline{\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\textsf{KeyGen}\\ \quad c := \Sigma.\textsf{Enc}(k, m_L)\\ \quad \text{return } c\end{array}}\ ; \quad \boxed{\begin{array}{l}\underline{\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\textsf{KeyGen}\\ \quad c := \Sigma.\textsf{Enc}(k, m_R)\\ \quad \text{return } c\end{array}}\ .$$

[2] There may be other reasonable ways to formalize this intuitive idea of security. For example, we might choose to give the attacker *two* ciphertexts instead of one, and demand that the attacker can't determine which of them encrypts m_L and which encrypts m_R. See Exercise 2.15.

Now the formal way to say that encryptions of m_L "look like" encryptions of m_R is:

> *"an encryption scheme is a good one if, when you plug its* KeyGen *and* Enc *algorithms into the template of the* EAVESDROP *subroutines above, the two implementations of* EAVESDROP *induce identical behavior in every calling program."*

Example *Does one-time pad satisfy this new security property? To find out, we plug in its algorithms to the above template, and obtain the following implementations:*

EAVESDROP(m_L, m_R):	EAVESDROP(m_L, m_R):
$k \leftarrow \{0,1\}^\lambda$ // KeyGen *of OTP*	$k \leftarrow \{0,1\}^\lambda$ // KeyGen *of OTP*
$c := k \oplus m_L$ // Enc *of OTP*	$c := k \oplus m_R$ // Enc *of OTP*
return c	return c

If these two implementations have the same effect on all calling programs (and indeed they do), then we would say that OTP satisfies this security property.

Is this a better/worse way to define security than the previous way? One security definition considers an attacker whose goal is to distinguish real ciphertexts from random values (real-vs-random paradigm), and the other considers an attacker whose goal is to distinguish real ciphertexts of two different plaintexts (left-vs-right paradigm). Is one "correct" and the other one "incorrect?" We save such discussion until later in the chapter.

2.2 Formalisms for Security Definitions

So far, we've defined security in terms of a single, self-contained subroutine, and imagined the attacker as a program that calls this subroutine. Later in the course we will need to generalize beyond a single subroutine, to a *collection* of subroutines that share common (private) state information. Staying with the software terminology, we call this collection a **library**:

Definition 2.3 (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 (just like a Java interface). 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 is a familiar library:*

$\mathcal{L}$
CTXT(m):
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m$
return c

And here is one possible calling program:

$$\boxed{\begin{array}{l} \mathcal{A}: \\ \hline m \leftarrow \{0,1\}^\lambda \\ c := \textsc{ctxt}(m) \\ \text{return } m \stackrel{?}{=} c \end{array}}$$

You can hopefully convince yourself that

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

If this $\mathcal{A}$ is linked to a different library, its output probability may be different. If a different calling program is linked to this $\mathcal{L}$, the output probability may be different.

Example *A library can contain several subroutines and private 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.*

$$\boxed{\begin{array}{l} \mathcal{L} \\ \hline s \leftarrow \{0,1\}^\lambda \\ \\ \underline{\textsc{reset}():} \\ \quad s \leftarrow \{0,1\}^\lambda \\ \\ \underline{\textsc{guess}(x \in \{0,1\}^\lambda):} \\ \quad \text{return } x \stackrel{?}{=} s \end{array}}$$

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 here) are available in all subroutine scopes (but not to the calling program).

Interchangeability

The idea that "no calling program behaves differently in the presence of these two libraries" still makes sense even for libraries with several subroutines. Since this is such a common concept, we devote new notation to it:

Definition 2.4 (Interchangeable) *Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be two libraries that have the same 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 boolean value,*

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \mathtt{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \mathtt{true}].$$

This definition considers calling programs that give boolean output. Imagine a calling program / attacker whose only goal is to distinguish two particular libraries (indeed, we often refer to the calling program as a **distinguisher**). A boolean output is enough for that task. You can think of the output bit as the calling program's "guess" for which library the calling program thinks it is linked to.

The distinction between "calling program outputs `true`" and "calling program outputs `false`" is not significant. If two libraries don't affect the calling program's probability of outputting `true`, then they also don't affect its probability of outputting `false`:

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

Example *Here are some very simple and straightforward ways that two libraries may be interchangeable. Hopefully it's clear that each pair of libraries has identical behavior, and therefore identical effect on all calling programs.*

*Despite being very simple examples, each of these concepts shows up as a **building block** in a real security proof in this book.*

```
FOO(x):                          FOO(x):
  if x is even:                    if x is even:
    return 0                         return 0
  else if x is odd:        ≡       else if x is odd:
    return 1                         return 1
  else:                            else:
    return -1                        return ∞
```

Their only difference happens in an unreachable block of code.

```
FOO(x):                          FOO(x):
  return BAR(x, x)                 return BAR(x, 0^λ)
                           ≡
BAR(a, b):                       BAR(a, b):
  k ← {0,1}^λ                      k ← {0,1}^λ
  return k ⊕ a                     return k ⊕ a
```

Their only difference is the value they assign to a variable that is never actually used.

```
                                 FOO(x, n):
FOO(x, n):                         for i = 1 to n:
  for i = 1 to λ:          ≡         BAR(x, i)
    BAR(x, i)                      for i = n + 1 to λ:
                                     BAR(x, i)
```

Their only difference is that one library unrolls a loop that occurs in the other library.

```
                                 FOO(x):
FOO(x):                            k ← {0,1}^λ
  k ← {0,1}^λ                      return k ⊕ BAR(x)
  y ← {0,1}^λ              ≡
  return k ⊕ y ⊕ x               BAR(x):
                                   y ← {0,1}^λ
                                   return y ⊕ x
```

Their only difference is that one library inlines a subroutine call that occurs in the other library.

Example *Here are more simple examples of interchangeable libraries that involve randomness:*

$$\boxed{\begin{array}{l}\underline{\textsc{foo}():}\\ \quad x \leftarrow \{0,1\}^\lambda\\ \quad y \leftarrow \{0,1\}^\lambda\\ \quad \text{return } x\|y\end{array}} \equiv \boxed{\begin{array}{l}\underline{\textsc{foo}():}\\ \quad z \leftarrow \{0,1\}^{2\lambda}\\ \quad \text{return } z\end{array}}$$

The uniform distribution over strings acts independently on different characters in the string ("‖" refers to concatenation).

$$\boxed{\begin{array}{l}k \leftarrow \{0,1\}^\lambda\\ \\ \underline{\textsc{foo}(x):}\\ \quad \text{return } k \oplus x\end{array}} \equiv \boxed{\begin{array}{l}\underline{\textsc{foo}(x):}\\ \quad \text{if } k \text{ not defined:}\\ \qquad k \leftarrow \{0,1\}^\lambda\\ \quad \text{return } k \oplus x\end{array}}$$

Sampling a value "eagerly" (as soon as possible) vs. sampling a value "lazily" (at the last possible moment before the value is needed). We assume that k is static/global across many calls to FOO*, and initially undefined.*

Formal Restatements of Previous Concepts

We can now re-state our security definitions from the previous section, using this new terminology.

Our "real-vs-random" style of security definition for encryption can be expressed as follows:

Definition 2.5 (Uniform ctxts) *An encryption scheme Σ has **one-time uniform ciphertexts** if:*

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\text{ots\$-real}}\\ \hline \underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\text{KeyGen}\\ \quad c \leftarrow \Sigma.\text{Enc}(k,m)\\ \quad \text{return } c\end{array}} \equiv \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}\\ \hline \underline{\textsc{ctxt}(m \in \Sigma.\mathcal{M}):}\\ \quad c \leftarrow \Sigma.\mathcal{C}\\ \quad \text{return } c\end{array}}$$

In other words, if you fill in the specifics of Σ (*i.e.*, the behavior of its KeyGen and Enc) into these two library "templates," and you get two libraries that are interchangeable (*i.e.*, have the same effect on all calling programs), we will say that Σ has one-time uniform ciphertexts.

Throughout this course, we will use the "\$" symbol to denote randomness (as in real-vs-random).[3]

Our "left-vs-right" style of security definition can be expressed as follows:

Definition 2.6 (One-time secrecy) *An encryption scheme Σ has **one-time secrecy** if:*

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\text{ots-L}}\\ \hline \underline{\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\text{KeyGen}\\ \quad c \leftarrow \Sigma.\text{Enc}(k,m_L)\\ \quad \text{return } c\end{array}} \equiv \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\text{ots-R}}\\ \hline \underline{\textsc{eavesdrop}(m_L, m_R \in \Sigma.\mathcal{M}):}\\ \quad k \leftarrow \Sigma.\text{KeyGen}\\ \quad c \leftarrow \Sigma.\text{Enc}(k,m_R)\\ \quad \text{return } c\end{array}}$$

[3] 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." Hence, randomized algorithms need to have spare change (i.e., money) sitting around. By convention, randomness comes in US dollars.

Previously in Claim 1.3 we argued that one-time-pad ciphertexts follow the uniform distribution. This actually shows that OTP satisfies the uniform ciphertexts definition:

Claim 2.7 (OTP rule) *One-time pad satisfies the one-time uniform ciphertexts property. In other words:*

$\mathcal{L}_{\text{otp-real}}$		$\mathcal{L}_{\text{otp-rand}}$
EAVESDROP$(m \in \{0,1\}^\lambda)$: $k \leftarrow \{0,1\}^\lambda$ // OTP.KeyGen return $k \oplus m$ // OTP.Enc(k,m)	$\equiv$	EAVESDROP$(m \in \{0,1\}^\lambda)$: $c \leftarrow \{0,1\}^\lambda$ // OTP.C return c

Because this property of OTP is quite useful throughout the course, I've given these two libraries special names (apart from $\mathcal{L}^{\text{OTP}}_{\text{ots\$-real}}$ and $\mathcal{L}^{\text{OTP}}_{\text{ots\$-rand}}$).

Discussion, Pitfalls

It is a common pitfall to imagine the calling 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 2.6). The big difference is:

- The algorithms of the scheme show a regular user's view of things. For example, the Enc algorithm takes two inputs: a key and a plaintext. Is there any way of describing an algorithm that takes two arguments other than writing something like Construction 1.1?

- The libraries capture the attacker's view of of a particular scenario, where the users *use the cryptographic algorithms in a very specific way.* For example, when we talk about security of encryption, we don't guarantee security when Alice lets the attacker choose her encryption key! But letting the attacker choose the plaintext is fine; we can guarantee security in that scenario. That's why Definition 2.5 describes a subroutine that calls Enc on a plaintext that is chosen by the calling program, but on a key k chosen by the library.

  A security definition says that some task (*e.g.*, distinguishing ciphertexts from random junk) is impossible, when the attacker is allowed certain influence over the inputs to the algorithms (*e.g.*, full choice of plaintexts, but no influence over the key), and is allowed to see certain outputs from those algorithms (*e.g.*, ciphertexts).

It's **wrong** to summarize one-time secrecy as: "I'm not allowed to choose what to encrypt, I have to ask the attacker to choose for me." The correct interpretation is: "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, Revisited

Kerckhoffs' Principle says to assume that the attacker has complete knowledge of the algorithms being used. Assume that the choice of keys is the *only* thing unknown to the attacker. Let's see how Kerckhoffs' Principle is reflected in our formal security definitions.

Suppose I write down the source code of two libraries, and your goal is to write an effective distinguisher. So you study the source code of the two libraries and write the best distinguisher that exists. It would be fair to say that your distinguisher "knows" what algorithms are used in the libraries, because it was designed based on the source code of these libraries. The definition of interchangeability considers *literally every* calling program, so it must also consider calling programs like yours that "know" what algorithms are being used.

However, there is an important distinction to make. If you know you might be linked to a library that executes the statement "$k \leftarrow \{0,1\}^\lambda$", that doesn't mean you know the actual *value* of k that was chosen at runtime. Our convention is that all variables within the library are privately scoped, and the calling program can learn about them only indirectly through subroutine outputs. In the library-distinguishing game, you are not allowed to pick a different calling program based on random choices that the library makes! After we settle on a calling program, we measure its effectiveness in terms of probabilities that take into account all possible outcomes of the random choices in the system.

In summary, the calling program "knows" what algorithms are being used (and how they are being used!) because the choice of the calling program is allowed to depend on the 2 specific libraries that we consider. The calling program "doesn't know" things like secret keys because the choice of calling program isn't allowed to depend on the outcome of random sampling done at runtime.

Kerckhoffs' Principle, applied to our formal terminology:

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

1. *which of the two possible libraries it is linked to in any particular execution, and*
2. *the random choices that the library will make during any particular execution (which are usually assigned to privately scoped variables within the library).*

2.3 How to Demonstrate Insecurity with Attacks

We always define security with respect to two libraries — or, if you like, two library *templates* that describe how to insert the algorithms of a cryptographic scheme into two libraries. If the two libraries that you get (after filling in the specifics of a particular scheme) are interchangeable, then we say that the scheme satisfies the security property. If we want to show that some scheme is *insecure*, we have to demonstrate **just one calling program** that behaves differently in the presence of those two libraries.

Let's demonstrate this process with the following encryption scheme, which is like one-time pad but uses bitwise-AND instead of XOR:

Construction 2.8

$\mathcal{K} = \{0,1\}^\lambda$	KeyGen:	Enc(k, m):
$\mathcal{M} = \{0,1\}^\lambda$	$k \leftarrow \{0,1\}^\lambda$	return $k \,\&\, m$ // bitwise-AND
$C = \{0,1\}^\lambda$	return k	

I haven't shown the Dec algorithm, because in fact there is no way to write one that satisfies the correctness requirement. But let's pretend we haven't noticed that yet, and ask whether this encryption scheme satisfies the two security properties defined previously.

Claim 2.9 *Construction 2.8 does **not** have one-time uniform ciphertexts (Definition 2.5).*

Proof To see whether Construction 2.8 satisfies uniform one-time ciphertexts, we have to plug in its algorithms into the two libraries of Definition 2.5 and see whether the resulting libraries are interchangeable. We're considering the following two libraries:

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
CTXT$(m \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$ // Σ.KeyGen
$c := k \,\&\, m$ // Σ.Enc
return c

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$
CTXT$(m \in \{0,1\}^\lambda)$:
$c \leftarrow \{0,1\}^\lambda$ // $\Sigma.C$
return c

To show that these two libraries are **not** interchangeable, we need to write a calling program that behaves differently in their presence. The calling program should make one or more calls to the CTXT subroutine. That means it needs to choose the input m that it passes, and it must make some conclusion (about which of the two libraries it is linked to) based on the return value that it gets. What m should the calling program choose as input to CTXT? What should the calling program look for in the return values?

There are many valid ways to write a good calling program, and maybe you can think of several. One good approach is to observe that bitwise-AND with k can never "turn a 0 into a 1." So perhaps the calling program should choose m to consist of all 0s. When $m = 0^\lambda$, the $\mathcal{L}_{\text{ots\$-real}}$ library will always return all zeroes, but the $\mathcal{L}_{\text{ots\$-rand}}$ library may return strings with both 0s and 1s.

We can formalize this idea with the following calling program:

$\mathcal{A}$:
$c :=$ CTXT(0^λ)
return $c \stackrel{?}{=} 0^\lambda$

.

Next, let's ensure that this calling program behaves differently when linked to each of these two libraries.

$\mathcal{A}$:		$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
$c := \textsc{ctxt}(\mathtt{0}^\lambda)$ return $c \stackrel{?}{=} \mathtt{0}^\lambda$	$\diamond$	$\textsc{ctxt}(m)$: $k \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$ $c := k \mathbin{\&} m$ return c

When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{ots\$-real}}$, c is computed as $k \mathbin{\&} \mathtt{0}^\lambda$. No matter what k is, the result is always all-zeroes. Therefore, $\mathcal{A}$ will always return `true`.

In other words, $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-real}} \Rightarrow \mathtt{true}] = 1$.

$\mathcal{A}$:		$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$
$c := \textsc{ctxt}(\mathtt{0}^\lambda)$ return $c \stackrel{?}{=} \mathtt{0}^\lambda$	$\diamond$	$\textsc{ctxt}(m)$: $c \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$ return c

When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{ots\$-rand}}$, c is chosen uniformly from $\{\mathtt{0},\mathtt{1}\}^\lambda$. The probability that c then happens to be all-zeroes is $1/2^\lambda$.

In other words, $\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots\$-rand}} \Rightarrow \mathtt{true}] = 1/2^\lambda$.

Since these two probabilities are different, this shows that $\mathcal{L}^{\Sigma}_{\text{ots\$-real}} \not\equiv \mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$. In other words, the scheme does not satisfy this uniform ciphertexts property. ■

So far we have two security definitions. Does this encryption scheme satisfy one but not the other?

Claim 2.10 *Construction 2.8 does **not** satisfy one-time secrecy (Definition 2.6).*

Proof This claim refers to a different security definition, which involves two different libraries. When we plug in the details of Construction 2.8 into the libraries of Definition 2.6, we get the following:

$\mathcal{L}^{\Sigma}_{\text{ots-L}}$	$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
$\textsc{eavesdrop}(m_L, m_R)$: $k \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$ // Σ.KeyGen $c := k \mathbin{\&} m_L$ // Σ.Enc(k, m_L) return c	$\textsc{eavesdrop}(m_L, m_R)$: $k \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$ // Σ.KeyGen $c := k \mathbin{\&} m_R$ // Σ.Enc(k, m_R) return c

Now we need to write a calling program that behaves differently in the presence of these two libraries. We can use the same overall idea as last time, but not the same actual calling program, since these libraries provide a different interface. In this example, the calling program needs to call the EAVESDROP subroutine which takes *two* arguments m_L and m_R. How should the calling program choose m_L and m_R? Which two plaintexts have different looking ciphertexts?

A good approach is to choose m_L to be all zeroes and m_R to be all ones. We know from before that an all-zeroes plaintext always encrypts to an all-zeroes ciphertext, so the calling program can check for that condition. More formally, we can define the calling program:

$\mathcal{A}$:
$c := \textsc{eavesdrop}(\mathtt{0}^\lambda, \mathtt{1}^\lambda)$ return $c \stackrel{?}{=} \mathtt{0}^\lambda$

Next, we need to compute its output probabilities in the presence of the two libraries.

$\mathcal{A}$:		$\mathcal{L}^{\Sigma}_{\text{ots-L}}$
$c := \textsc{eavesdrop}(\boxed{0^\lambda}, 1^\lambda)$ return $c \overset{?}{=} 0^\lambda$	$\diamond$	$\textsc{eavesdrop}(\boxed{m_L}, m_R)$: $k \leftarrow \{0,1\}^\lambda$ $c := k \mathbin{\&} \boxed{m_L}$ return c

When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{ots-L}}$, c is computed as an encryption of $m_L = 0^\lambda$. No matter what k is, the result is always all-zeroes. So,

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

$\mathcal{A}$:		$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
$c := \textsc{eavesdrop}(0^\lambda, \boxed{1^\lambda})$ return $c \overset{?}{=} 0^\lambda$	$\diamond$	$\textsc{eavesdrop}(m_L, \boxed{m_R})$: $k \leftarrow \{0,1\}^\lambda$ $c := k \mathbin{\&} \boxed{m_R}$ return c

When $\mathcal{A}$ is linked to $\mathcal{L}_{\text{ots-R}}$, c is computed as an encryption of $m_R = 1^\lambda$. In other words, $c := k \mathbin{\&} 1^\lambda$. But the bitwise-AND of any string k with all 1s is just k itself. So c is just equal to k, which was chosen uniformly at random. The probability that a uniformly random c happens to be all-zeroes is

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{ots-R}} \Rightarrow \texttt{true}] = 1/2^\lambda.$$

Since these two probabilities are different, $\mathcal{L}^{\Sigma}_{\text{ots-L}} \not\equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$ and the scheme does not have one-time secrecy. ∎

2.4 How to Prove Security with The Hybrid Technique

We proved that one-time pad satisfies the uniform ciphertexts property (Claim 1.3) by carefully calculating certain probabilities. This will not be a sustainable strategy as things get more complicated later in the course. In this section we will introduce a technique for proving security properties, which usually avoids tedious probability calculations.

Chaining Several Components

Before getting to a security proof, we introduce a convenient lemma. Consider a compound program 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, $\mathcal{L}_1$ can make calls to subroutines in $\mathcal{L}_2$, but not vice-versa. 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 2.11 (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 compound 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 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 distributions. 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:

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

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

An Example Hybrid Proof

In this section we will prove something about the following scheme, which encrypts twice with OTP, using independent keys:

Construction 2.12 ("Double OTP")

$\mathcal{K} = (\{\texttt{0},\texttt{1}\}^\lambda)^2$ $\mathcal{M} = \{\texttt{0},\texttt{1}\}^\lambda$ $\mathcal{C} = \{\texttt{0},\texttt{1}\}^\lambda$	KeyGen: $k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$ $k_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$ return (k_1, k_2)	$\text{Enc}\big((k_1,k_2), m\big)$: $c_1 := k_1 \oplus m$ $c_2 := k_2 \oplus c_1$ return c_2	$\text{Dec}\big((k_1,k_2), c_2\big)$: $c_1 := k_2 \oplus c_2$ $m := k_1 \oplus c_1$ return m

It would not be too hard to directly show that ciphertexts in this scheme are uniformly distributed, as we did for plain OTP. However, the new hybrid technique will allow us to leverage what we already know about OTP in an elegant way, and avoid any probability calculations.

Claim 2.13 *Construction 2.12 has one-time uniform ciphertexts (Definition 2.6).*

Proof In terms of libraries, we must show that:

$$\boxed{\begin{array}{l} \mathcal{L}^{\Sigma}_{\text{ots\$-real}} \\ \hline \underline{\textsc{ctxt}(m):} \\ \quad \left.\begin{array}{l} k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda \\ k_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda \end{array}\right\} \text{KeyGen} \\ \quad \left.\begin{array}{l} c_1 := k_1 \oplus m \\ c_2 := k_2 \oplus c_1 \\ \text{return } c_2 \end{array}\right\} \text{Enc} \end{array}} \equiv \boxed{\begin{array}{l} \mathcal{L}^{\Sigma}_{\text{ots\$-rand}} \\ \hline \underline{\textsc{ctxt}(m):} \\ \quad c \leftarrow \{\texttt{0},\texttt{1}\}^\lambda \\ \quad \text{return } c \end{array}}$$

Instead of directly comparing these two libraries, we will introduce some additional libraries $\mathcal{L}_{\textsf{hyb-1}}$, $\mathcal{L}_{\textsf{hyb-2}}$, $\mathcal{L}_{\textsf{hyb-3}}$, and show that:

$$\mathcal{L}^{\Sigma}_{\textsf{ots\$-real}} \equiv \mathcal{L}_{\textsf{hyb-1}} \equiv \mathcal{L}_{\textsf{hyb-2}} \equiv \mathcal{L}_{\textsf{hyb-3}} \equiv \mathcal{L}^{\Sigma}_{\textsf{ots\$-rand}}$$

Since the $\equiv$ symbol is transitive, this will achieve our goal.

The intermediate libraries are called **hybrids**, since they will contain a mix of characteristics from the two "endpoints" of this sequence. These hybrids are chosen so that it is very easy to show that consecutive libraries in this sequence are interchangeable. The particular hybrids we have in mind here are:

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\textsf{ots\$-real}}\\ \hline \underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad k_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := k_2 \oplus c_1\\ \quad \text{return } c_2\end{array}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := \textsc{ctxt}'(c_1)\\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\textsf{otp-real}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad k \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } k \oplus m'\end{array}}}_{\mathcal{L}_{\textsf{hyb-1}}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := \textsc{ctxt}'(c_1)\\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\textsf{otp-rand}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad c \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } c\end{array}}}_{\mathcal{L}_{\textsf{hyb-2}}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } c_2\end{array}}}_{\mathcal{L}_{\textsf{hyb-3}}} \equiv \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\textsf{ots\$-rand}}\\ \hline \underline{\textsc{ctxt}(m):}\\ \quad c_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } c_2\end{array}}$$

Next, we provide a justification for each "$\equiv$" in the expression above. For each pair of adjacent libraries, we highlight their differences below:

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\textsf{ots\$-real}}\\ \hline \underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad k_2 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := k_2 \oplus c_1\\ \quad \text{return } c_2\end{array}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := \textsc{ctxt}'(c_1)\\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\textsf{otp-real}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad k \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } k \oplus m'\end{array}}}_{\mathcal{L}_{\textsf{hyb-1}}}$$

The only difference between these two libraries is that the highlighted expressions have been factored out into a separate subroutine, and some variables have been renamed. In both libraries, c_2 is chosen as the XOR of c_1 and a uniformly chosen string. These differences make no effect on the calling program. Importantly, the subroutine that we have factored out is exactly the one in the $\mathcal{L}_{\textsf{otp-real}}$ library (apart from renaming the subroutine).

$$\underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := \textsc{ctxt}'(c_1)\\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\textsf{otp-real}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad k \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } k \oplus m'\end{array}}}_{\mathcal{L}_{\textsf{hyb-1}}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad c_1 := k_1 \oplus m\\ \quad c_2 := \textsc{ctxt}'(c_1)\\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\textsf{otp-rand}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad c \leftarrow \{\texttt{0},\texttt{1}\}^\lambda\\ \quad \text{return } c\end{array}}}_{\mathcal{L}_{\textsf{hyb-2}}}$$

Claim 2.7 says that $\mathcal{L}_{\textsf{otp-real}} \equiv \mathcal{L}_{\textsf{otp-rand}}$, so Lemma 2.11 says that we can replace an instance of $\mathcal{L}_{\textsf{otp-real}}$ in a compound library with $\mathcal{L}_{\textsf{otp-rand}}$, as we have done here. This change will have no effect on the calling program.

$$\underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda \\ \quad c_1 := k_1 \oplus m \\ \quad c_2 := \textsc{ctxt}'(c_1) \\ \quad \text{return } c_2\end{array}} \diamond \boxed{\begin{array}{l}\mathcal{L}_{\text{otp-rand}}\\ \hline \underline{\textsc{ctxt}'(m'):}\\ \quad c \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda\\ \quad \text{return } c\end{array}}}_{\mathcal{L}_{\text{hyb-2}}} \equiv \underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda \\ \quad c_1 := k_1 \oplus m \\ \quad c_2 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda \\ \quad \text{return } c_2\end{array}}}_{\mathcal{L}_{\text{hyb-3}}}$$

The only difference between these two libraries is that a subroutine call has been inlined. This difference has no effect on the calling program.

$$\underbrace{\boxed{\begin{array}{l}\underline{\textsc{ctxt}(m):}\\ \quad k_1 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda \\ \quad c_1 := k_1 \oplus m \\ \quad c_2 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda \\ \quad \text{return } c_2\end{array}}}_{\mathcal{L}_{\text{hyb-3}}} \equiv \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}\\ \hline \underline{\textsc{ctxt}(m):}\\ \quad c_2 \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda\\ \quad \text{return } c_2\end{array}}$$

The only difference between these two libraries is that the two highlighted lines have been removed. But it should be clear that these lines have no effect: k_1 is used only to compute c_1, which is never used again. Hence, this difference has no effect on the calling program.

The final hybrid is exactly $\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$ (although with a variable name changed). We have shown that $\mathcal{L}^{\Sigma}_{\text{ots\$-rand}} \equiv \mathcal{L}^{\Sigma}_{\text{ots\$-real}}$, meaning that this encryption scheme has one-time uniform ciphertexts. ■

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 $\mathcal{L}_{\text{right}}$. You must justify why each modification doesn't affect the calling program (*i.e.*, why the two libraries before/after your modification are interchangeable).
- As discussed in Section 2.2, 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 have no effect on

the calling program. As we progress in the course, we will see more kinds of useful modifications.

A Contrasting Example

Usually the boundary between secure and insecure is razor thin. Let's make a small change to the previous encryption scheme to illustrate this point. Instead of applying OTP to the plaintext twice, with independent keys, what would happen if we use the *same key?*

Construction 2.14 ("dOuBℓ∃ OTP")

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

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

Enc(k, m):
  $c_1 := k \oplus m$
  $c_2 := k \oplus c_1$
  return c_2

Dec(k, c_2):
  $c_1 := k \oplus c_2$
  $m := k \oplus c_1$
  return m

You probably noticed that the ciphertext c_2 is computed as $c_2 := k \oplus (k \oplus m)$, which is just a fancy way of saying $c_2 := m$. There is certainly no way this kind of "double-OTP" is secure.

For educational purposes, let's try to repeat the steps of our previous security proof on this (insecure) scheme and **see where things break down.** If we wanted to show that Construction 2.14 has uniform ciphertexts, we would have to show that the following two libraries are interchangeable:

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
ctxt(m):
  $k \leftarrow \{0,1\}^\lambda$ // KeyGen
  $c_1 := k \oplus m$
  $c_2 := k \oplus c_1$
  return c_2
} Enc

$\overset{?}{\equiv}$

$\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$
ctxt(m):
  $c \leftarrow \{0,1\}^\lambda$
  return c

In the previous hybrid proof, the first step was to factor out the statements "$k_2 \leftarrow \{0,1\}^\lambda$; $c_2 := k_2 \oplus c_1$" into a separate subroutine, so we could argue that the result of c_2 was uniformly distributed. If we do something analogous with this example, we get:

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
ctxt(m):
  $k \leftarrow \{0,1\}^\lambda$
  $c_1 := k \oplus m$
  $c_2 := k \oplus c_1$
  return c_2

$\overset{?}{\equiv}$

ctxt(m):
  $c_1 := k \oplus m$ // ??
  $c_2 :=$ ctxt$'(c_1)$
  return c_2

$\diamond$

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

$\mathcal{L}_{\text{hyb}}$

Do you see the problem? In "$\mathcal{L}_{\text{hyb}}$", we have tried to move the variable k into $\mathcal{L}_{\text{otp-real}}$. Since this scope is private, every operation we want to do with k has to be provided by its container library $\mathcal{L}_{\text{otp-real}}$. But there is a mismatch: $\mathcal{L}_{\text{otp-real}}$ only gives us a way to use k in one xor expression, whereas we need to use the same k in two xor expressions to match the behavior of $\mathcal{L}_{\text{ots\$-real}}$. The compound library $\mathcal{L}_{\text{hyb}}$ has an unresolved reference to k in the line "$c_1 := k \oplus m$," and therefore doesn't have the same behavior as $\mathcal{L}_{\text{ots\$-real}}$.[4] This is the step of the security proof that breaks down.

Here's a more conceptual way to understand what went wrong here. The important property of OTP is that its ciphertexts look uniform *when the key is used to encrypt only one plaintext*. This "double OTP" variant uses OTP in a way that doesn't fulfill that condition, and therefore provides no security guarantee. The previous (successful) proof was able to factor out some XOR's in terms of $\mathcal{L}_{\text{otp-real}}$ without breaking anything, and that's how we know the scheme was using OTP in a way that is consistent with its security guarantee.

As you can hopefully see, the process of a security proof provides a way to catch these kinds of problems. It is very common in a hybrid proof to factor out some statements in terms of a library from some other security definition. This step can only be done successfully if the underlying cryptography is being used in an appropriate way.

2.5 How to Compare/Contrast Security Definitions

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. In this chapter we introduced two security definitions: one in the "real-vs-random" style and one in the "left-vs-right" style. In this section we treat the *security definitions themselves* as objects worth studying. Are both of these security definitions "the same," in some sense? Do they both capture all of our intuitions about security?

One Security Definition Implies Another

One way to compare/contrast two security definitions is to prove that one implies the other. In other words, if an encryption scheme satisfies definition #1, then it also satisfies definition #2.

Theorem 2.15 *If an encryption scheme Σ has one-time uniform ciphertexts (Definition 2.5), then Σ also has one-time secrecy (Definition 2.6). 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 this statement is quite simple and unsurprising. "If all plaintexts m induce a *uniform* distribution of ciphertexts, then all m induce the *same* distribution of ciphertexts."

This fairly straight-forward statement can be proven formally, giving us another example of the hybrid proof technique:

Proof We are proving an if-then statement. We want to show that the "then"-part of the statement is true; that is, $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$. We are allowed to use the fact that the "if"-part is true; that is, $\mathcal{L}^{\Sigma}_{\text{ots\$-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$.

[4] I would say that the library "doesn't compile" due to a scope/reference error.

The proof uses the hybrid technique. We will start with the library $\mathcal{L}_{\text{ots-L}}$, and make a small sequence of justifiable changes to it, until finally reaching $\mathcal{L}_{\text{ots-R}}$. Along the way, we can use the fact that $\mathcal{L}_{\text{ots\$-real}} \equiv \mathcal{L}_{\text{ots\$-rand}}$. This suggests some "strategy" for the proof: if we can somehow get $\mathcal{L}_{\text{ots\$-real}}$ to appear as a component in one of the hybrid libraries, then we can replace it with $\mathcal{L}_{\text{ots\$-rand}}$ (or vice-versa), in a way that hopefully makes progress towards our goal of transforming $\mathcal{L}_{\text{ots-L}}$ to $\mathcal{L}_{\text{ots-R}}$.

Below we list the sequence of hybrid libraries, and justify why each one is interchangeable with the previous library.

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

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

The starting point of our hybrid sequence is $\mathcal{L}^{\Sigma}_{\text{ots-L}}$.

```
EAVESDROP(m_L, m_R):
  c := CTXT(m_L)
  return c
```

$\diamond$

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

```
CTXT(m):
  k ← Σ.KeyGen
  c ← Σ.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 2.5. This was a strategic choice, because of what happens next.

```
EAVESDROP(m_L, m_R):
  c := CTXT(m_L)
  return c
```

$\diamond$

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

```
CTXT(m):
  c ← Σ.C
  return c
```

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$ has been replaced with $\mathcal{L}^{\Sigma}_{\text{ots\$-rand}}$. The chaining lemma Lemma 2.11 says that this change has no effect on the library's behavior, since the two $\mathcal{L}_{\text{ots\$-}\star}$ libraries are interchangeable.

```
EAVESDROP(m_L, m_R):
  c := CTXT(m_R)
  return c
```

$\diamond$

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

```
CTXT(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. 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.

$\text{EAVESDROP}(m_L, m_R)$:
$c := \text{CTXT}(m_R)$
return c

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{ots\$-real}}$
$\text{CTXT}(m)$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m)$
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.

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$
$\text{EAVESDROP}(m_L, m_R)$:
$k \leftarrow \Sigma.\text{KeyGen}$
$c \leftarrow \Sigma.\text{Enc}(k, m_R)$
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. ■

One Security Definition Doesn't Imply Another

Another way we might compare security definitions is to identify any schemes that satisfy one definition without satisfying the other. This helps us understand the boundaries and "edge cases" of the definition.

A word of warning: If we have two security definitions that both capture our intuitions rather well, then any scheme which satisfies one definition and not the other is bound to appear **unnatural and contrived.** The point is to gain more understanding of the *security definitions themselves*, and unnatural/contrived schemes are just a means to do that.

Theorem 2.16 *There is an encryption scheme that satisfies one-time secrecy (Definition 2.6) but not one-time uniform ciphertexts (Definition 2.5). In other words, one-time secrecy* ***does not*** *necessarily imply one-time uniform ciphertexts.*

Proof One such encryption scheme is given below:

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

This scheme is just OTP with the bits 00 added to every ciphertext. The following facts about the scheme should be believable (and the exercises encourage you to prove them formally if you would like more practice at that sort of thing):

- This scheme satisfies one-time one-time secrecy, meaning that encryptions of m_L are distributed identically to encryptions of m_R, for any m_L and m_R of the attacker's choice. We can characterize the ciphertext distribution in both cases as "λ uniform bits followed by `00`.” Think about how you might use the hybrid proof technique to formally prove that this scheme satisfies one-time secrecy!

- This scheme does not satisfy the one-time uniform ciphertexts property. Its ciphertexts always end with `00`, whereas uniform strings end with `00` with probability 1/4. Think about how you might formallize this observation as a calling program / distinguisher for the relevant two libraries! ■

You might be thinking, surely this can be fixed by redefining the ciphertext space as $\mathcal{C}$ as the set of $\lambda + 2$-bit strings whose last two bits are `00`. This is a clever idea, and indeed it would work. If we change the definition of the ciphertext space $\mathcal{C}$ following this suggestion, then the scheme would satisfy the uniform ciphertexts property (this is because the $\mathcal{L}_{\text{ots\$-rand}}$ library samples uniformly from whatever $\mathcal{C}$ is specified as part of the encryption scheme).

But this observation raises an interesting point. Isn’t it weird that security hinges on how narrowly you define the set $\mathcal{C}$ of ciphertexts, when $\mathcal{C}$ really has no effect on the *functionality* of encryption? Again, no one really cares about this contrived “OTP + `00`” encryption scheme. The point is to illuminate interesting edge cases in the *security definition itself!*

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:

$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{L}_1$	$\mathcal{L}_2$
$r_1 := \textsc{rand}(6)$ $r_2 := \textsc{rand}(6)$ return $r_1 \stackrel{?}{=} r_2$	$r := \textsc{rand}(6)$ return $r \stackrel{?}{\geqslant} 3$	$\underline{\textsc{rand}(n):}$ $r \leftarrow \mathbb{Z}_n$ return r	$\underline{\textsc{rand}(n):}$ return 0

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

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

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

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

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)

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

(b)

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

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

(d)

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

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

(c)

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

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

2.3. Show that the following libraries are interchangeable:

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

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

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

$\mathcal{L}_{\text{right}}$
$\underline{\text{EAVESDROP}(m_L, m_R \in \{0,1\}^\lambda):}$
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m_R$
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. In the proof of Claim 2.9 we considered an attacker / calling program that calls CTXT(0^λ).

(a) How does this attacker's effectiveness change if it calls CTXT(1^λ) instead?

(b) How does its effectiveness change if it calls CTXT(m) for a uniformly chosen m?

2.7. The following scheme encrypts a plaintext by simply reordering its bits, according to the secret permutation k.

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

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 $m_1 \cdots m_\lambda$

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.

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.

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

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

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

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`.

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

2.10. Suppose we modify the scheme from the previous problem to first permute the bits of m (as in Exercise 2.7) 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).

The Enc algorithm of one-time pad is deterministic, but our definitions of encryption allow Enc to be randomized (*i.e.*, it may give different outputs when called twice with the same k and m). **For full credit**, you should prove the statement even for the case of Enc is randomized. However, you may assume that Dec is deterministic.

Hint: The definition of interchangeability does not place any restriction on the running time of the distinguisher/calling program. Even an exhaustive brute-force attack would be valid.

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

$$
\boxed{
\begin{array}{lll}
\mathcal{K} = (\Sigma.\mathcal{K})^2 & & \\
\mathcal{M} = \Sigma.\mathcal{M} & & \\
\mathcal{C} = \Sigma.\mathcal{C} & \underline{\mathsf{Enc}((k_1, k_2), m)\text{:}} & \underline{\mathsf{Dec}((k_1, k_2), c_2)\text{:}} \\
 & \quad c_1 := \Sigma.\mathsf{Enc}(k_1, m) & \quad c_1 := \Sigma.\mathsf{Dec}(k_2, c_2) \\
\underline{\mathsf{KeyGen}\text{:}} & \quad c_2 := \Sigma.\mathsf{Enc}(k_2, c_1) & \quad m := \Sigma.\mathsf{Dec}(k_1, c_1) \\
\quad k_1 \leftarrow \Sigma.\mathcal{K} & \quad \text{return } c_2 & \quad \text{return } m \\
\quad k_2 \leftarrow \Sigma.\mathcal{K} & & \\
\quad \text{return } (k_1, k_2) & &
\end{array}
}
$$

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

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

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

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

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

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

Note: you must prove both directions of the if-and-only-if with a hybrid proof.

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

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

FOO($m_L, m_R \in \Sigma.\mathcal{M}$):
  $k_1 \leftarrow \Sigma.\text{KeyGen}$
  $c_1 := \Sigma.\text{Enc}(k_1, m_L)$
  $k_2 \leftarrow \Sigma.\text{KeyGen}$
  $c_2 := \Sigma.\text{Enc}(k_2, m_R)$
  return (c_1, c_2)

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

FOO($m_L, m_R \in \Sigma.\mathcal{M}$):
  $k_1 \leftarrow \Sigma.\text{KeyGen}$
  $c_1 := \Sigma.\text{Enc}(k_1, m_R)$
  $k_2 \leftarrow \Sigma.\text{KeyGen}$
  $c_2 := \Sigma.\text{Enc}(k_2, m_L)$
  return (c_1, c_2)

Note: you must prove both directions of the if-and-only-if with a hybrid proof.

2.16. 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.

2.17. 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 the uniform distribution on $\{\texttt{0},\texttt{1}\}^\lambda$.

(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.

2.18. Complete the proof of Theorem 2.16.

(a) Formally prove (using the hybrid technique) that the scheme in that theorem satisfies one-time secrecy.

(b) Give a distinguishing calling program to show that the scheme doesn't satisfy one-time uniform ciphertexts.