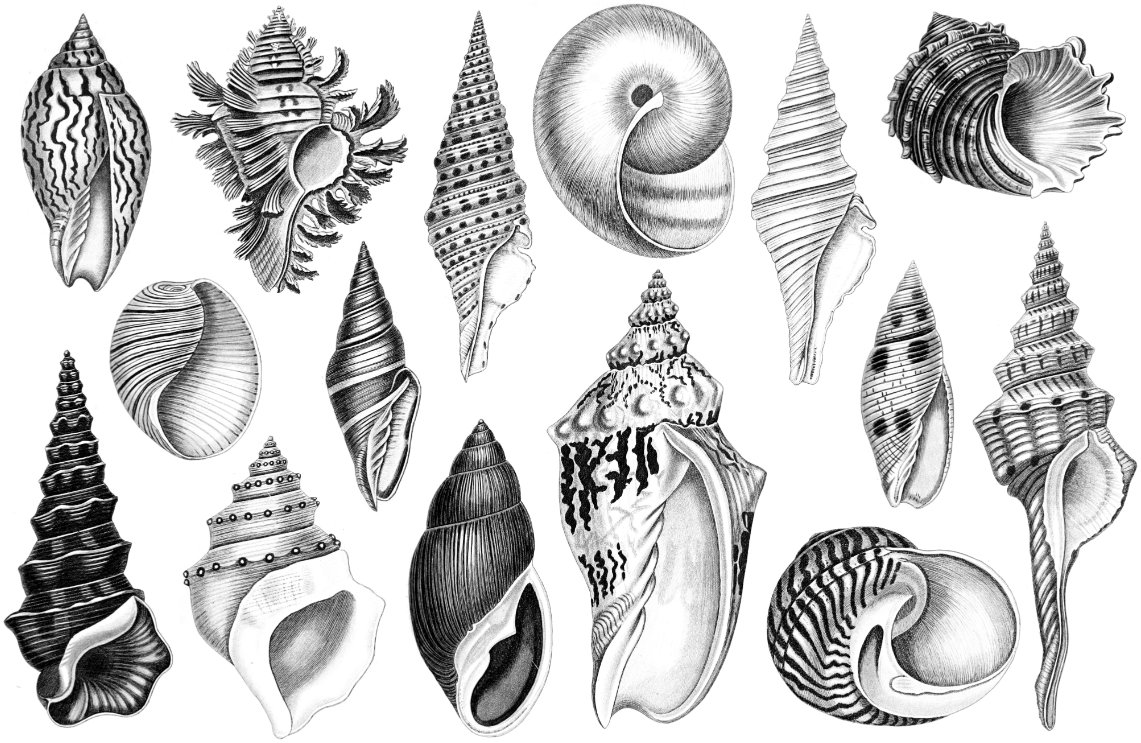


The Joy of Cryptography

Mike Rosulek ⟨rosulekm@eecs.oregonstate.edu⟩
School of Electrical Engineering & Computer Science
Oregon State University, Corvallis, Oregon, USA

Draft of January 22, 2018

Foreword

These are lecture notes for cs427 at Oregon State University, an introductory course in cryptography at the advanced undergraduate level. By reading and studying these notes, you should expect to learn how to:

- State and interpret the standard formal definitions for the most common cryptographic security properties (privacy and authentication).
- Formally prove security properties of sound cryptographic constructions, and break the security of unsound ones.
- Choose the appropriate cryptographic primitive for a task (block ciphers, hash functions, MACs, public-key encryption, etc.) while avoiding common pitfalls.

Along the way, you will also learn how the most common cryptographic constructions work.

Background

You will get the most out of these notes if you have a solid foundation in standard undergraduate computer science material:

- Discrete mathematics [required]: for modular arithmetic, discrete probabilities, simple combinatorics, and especially proof techniques.
- Algorithms & data structures [highly recommended]: for reasoning about computations at an abstract level.
- Theory of computation (automata, formal languages & computability) [recommended]: for even more experience formally reasoning about abstract processes of computation.

Disclaimer

You are reading an early draft of this book. Of course I make every effort to ensure the accuracy of the content, but the content has not yet benefitted from many critical readers. So, *caveat emptor!*

More than that, much interesting and hugely important material is not covered. Some more minor missing material is indicated as such and labeled “to-do”, but conspicuously missing major sections are simply absent. Anyone with a pre-existing opinion about cryptography will likely be appalled at what is missing. In my defense, my university schedule is on the quarter system which necessitates that I generate 10 weeks of course material. Everything beyond that is produced via the much slower *labor-of-love* mechanism.

I welcome feedback from readers, educators, cryptographers — not only on errors and typos but also on the selection, organization, and presentation of the material.

Code-based games

The security definitions and proofs in these notes are presented in a style that is known to the outside world as *code-based games.* I've chosen this style because I think it offers significant pedagogical benefits:

- Every security definition can be expressed in the same style, as the indistinguishability of two games. In my terminology, the games are *libraries* with a common interface/API but different internal implementations. An adversary is any calling program on that interface. These libraries use a concrete pseudocode that reduces ambiguity about an adversary's capabilities. For instance, the adversary controls arguments to subroutines that it calls and sees only the return value. The adversary cannot see any variables that are privately scoped to the library.

- A consistent framework for definitions leads to a consistent process for *proving* and *breaking* security — the two fundamental activities in cryptography.

  In these notes, *breaking* a construction always corresponds to writing a program that expects a particular interface and behaves as differently as possible in the presence of two particular implementations of the interface.

  Proving security nearly always refers to showing a sequence of libraries (called *hybrids*), each of which is indistinguishable from the previous one. Each of these hybrids is written in concrete pseudocode. By identifying what security property we wish to prove, we identify what the endpoints of this sequence must be. The steps that connect adjacent hybrids are stated in terms of syntactic rewriting rules for pseudocode, including down-to-earth steps like factoring out and inlining subroutines, changing the value of unused variables, and so on.

- Cryptography is full of conditional statements of security: *"if A is a secure thingamajig, then B is a secure doohickey."* A conventional proof of such a statement would address the contrapositive: *"given an adversary that attacks the doohickey-security of B, I can construct an attack on the thingamajig-security of A."*

  In my experience, students struggle to find the right way to transform an abstract, hypothetical B-attacking adversary into a successful A-attacking adversary. By defining security in terms of games/libraries, we can avoid this abstract challenge, and indeed avoid the context switch into the contrapositive altogether. In these notes, the thingamajig-security of A gives the student a new *rewriting rule* that can be placed in his/her toolbox and used to bridge hybrids when proving the doohickey-security of B.

Code-based games were first proposed by Shoup[1] and later expanded by Bellare & Rogaway.[2] These notes adopt a simplified and unified style of games, since the goal is not to encompass every possible security definition but only the fundamental ones. The most significant difference in style is that the games in these notes have no explicit Initialize or Finalize step. As a result, all security definitions are expressed as *indistinguishability* of two games/libraries, even security definitions that are fundamentally about unforgeability. Yet, we can still reason about unforgeability properties within this framework. For instance, to say that no adversary can forge a MAC, it suffices to say that no adversary can distinguish a MAC-verification subroutine from a subroutine that always returns false. An index of security definitions has been provided at the end of the book.

One instance where the approach falls short, however, is in defining collision resistance. I have not been able to define it in this framework in a way that is both easy to use and easy to interpret (and perhaps I achieved neither in the end). See Chapter 12 for my best attempt.

Supplementary material

Security proofs in this book follow a standard pattern: We start from one "library" and perform a sequence of small, cumulative modifications. Each modification results in a separate hybrid library that is indistinguishable from the previous one.

I have prepared PDF slide decks to supplement the security proofs contained in the book. They are available from the course website. The slides allow the reader to step forward and backward through the proof's sequence of logical steps, seeing only the current hybrid library at any time (with changes highlighted and annotated).

Acknowledgements

Some financial support for writing this book has been kindly provided by the National Science Foundation (award #1149647) and the Oregon State University Open Textbook Initiative.

About the cover

The cover design consists of assorted shell illustrations from *Bibliothèque conchyliologique*, published in 1846. The images are no longer under copyright, and were obtained from the Biodiversity Heritage Library (http://biodiversitylibrary.org/bibliography/11590). Like a properly deployed cryptographic primitive, a properly deployed shell is the most robust line of defense for a mollusk. To an uniformed observer, a shell is just a shell, and crypto is just crypto. However, there are a wide variety of cryptographic primitives, each of which provides protection against a different kind of attack. Just as for a seasoned *conchologist*, the joy is in appreciating the unique beauty of each form and understanding the subtle differences among them.

[1] Victor Shoup: *Sequences of Games: A Tool for Taming Complexity in Security Proofs.* ia.cr/2004/332

[2] Mihir Bellare & Philip Rogaway: *Code-Based Game-Playing Proofs and the Security of Triple Encryption.* ia.cr/2004/331

Copyright

Contents

0 Review of Concepts & Notation

0.1 Modular Arithmetic

We write the set of integers and the set of natural numbers as:

$$\mathbb{Z} \overset{\text{def}}{=} \{\ldots,-2,-1,0,1,2,\ldots\};$$
$$\mathbb{N} \overset{\text{def}}{=} \{0,1,2,\ldots\}.$$

Theorem 0.1 (Division Theorem) *For all $a,n \in \mathbb{Z}$ with $n \neq 0$, there exist* unique *$q,r \in \mathbb{Z}$ satisfying $a = qn + r$ and $0 \leqslant r < |n|$. Since q and r are unique, we use $\lfloor a/n \rfloor$ to denote q and $a \,\%\, n$ to denote r. Hence:*

$$a = \left\lfloor \frac{a}{n} \right\rfloor n + (a \,\%\, n).$$

The % symbol is often called the **modulo** operator. Beware that some programming languages also have a % operator in which $a \,\%\, n$ *always has the same sign* as a. We will instead use the convention that $a \,\%\, n$ is always always nonnegative.

Definition 0.2 *For $x,n \in \mathbb{Z}$, we say that n* ***divides*** *x (or x* ***is a multiple of*** *n), and write $n \mid x$, if there exists an integer k such that $kn = x$.*

We say that a and b are ***congruent modulo*** *n, and write $a \equiv_n b$, if $n \mid (a-b)$. Equivalently, $a \equiv_n b$ if and only if $a \,\%\, n = b \,\%\, n$.*

We write $\mathbb{Z}_n \overset{\text{def}}{=} \{0,\ldots,n-1\}$ to denote the set of ***integers modulo n.***

In other textbooks you may have seen "$a \equiv_n b$" written as "$a \equiv b \pmod{n}$".

There is a subtle — and often confusing — distinction between the expressions "$a \,\%\, n = b$" and "$a \equiv_n b$." In the first expression, "$a \,\%\, n$" refers to an integer that is always between 0 and $n-1$, and the equals sign denotes equality *over the integers.* In the second expression, the "$\equiv_n$" symbol denotes congruence modulo n, which is a weaker condition than equality over the integers. Note that $a = b$ implies $a \equiv_n b$, but not vice-versa.

Example $99 \equiv_{10} 19$ *because* 10 *divides* $99 - 19$ *according to the definition. But* $99 \neq 19 \,\%\, 10$ *because the right-hand side evaluates to the integer* $19 \,\%\, 10 = 9$, *which is not the same* integer *as the left-hand side* 99.

When adding, subtracting, and multiplying modulo n, it doesn't affect the final result to reduce intermediate steps modulo n. That is, we have the following facts about modular arithmetic:

$$(a+b) \,\%\, n = \Big[(a \,\%\, n) + (b \,\%\, n)\Big] \,\%\, n;$$
$$(a-b) \,\%\, n = \Big[(a \,\%\, n) - (b \,\%\, n)\Big] \,\%\, n;$$
$$ab \,\%\, n = \Big[(a \,\%\, n)(b \,\%\, n)\Big] \,\%\, n.$$

Division is not always possible in $\mathbb{Z}_n$; we will discuss this fact later in the class.

0.2 Strings

We write $\{\texttt{0},\texttt{1}\}^n$ to denote the set of n-bit binary strings, and $\{\texttt{0},\texttt{1}\}^*$ to denote the set of all (finite-length) binary strings. When x is a string of bits, we write $|x|$ to denote the length (in bits) of that string, and we write $\overline{x}$ to denote the result of flipping every bit in x. When it's clear from context that we're talking about strings instead of numbers, we write $\texttt{0}^n$ and $\texttt{1}^n$ to denote strings of n zeroes and n ones, respectively.

When x and y are strings of the same length, we write $x \oplus y$ to denote the bitwise exclusive-or (xor) of the two strings. So, for example, $\texttt{0011} \oplus \texttt{0101} = \texttt{0110}$. The following facts about the xor operation are frequently useful:

$x \oplus x = \texttt{0}^{\lvert x \rvert}$	xor'ing a string with itself results in zeroes.
$x \oplus \texttt{0}^{\lvert x \rvert} = x$	xor'ing with zeroes has no effect.
$x \oplus \texttt{1}^{\lvert x \rvert} = \overline{x}$	xor'ing with ones flips every bit.
$x \oplus y = y \oplus x$	xor is symmetric.
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	xor is associative.

As a corollary:

$$a = b \oplus c \iff b = a \oplus c \iff c = a \oplus b.$$

We use notation $x \| y$ to denote the concatenation of two strings x and y.

0.3 Functions

Let X and Y be finite sets. A function $f : X \to Y$ is:

injective (1-to-1) if it never maps two different inputs to the same output. Formally: $x \neq x' \Rightarrow f(x) \neq f(x')$.

surjective (onto) if every element in Y is a possible output of f. Formally: for all $y \in Y$ there exists an $x \in X$ with $f(x) = y$.

bijective (1-to-1 correspondence) if f is both injective and surjective. If this is the case, we say that f is a *bijection*. Note that bijectivity implies that $|X| = |Y|$.

0.4 Probability

Definition 0.3 *A **(discrete) probability distribution** $\mathcal{D}$ over a set X of **outcomes** is a function $\mathcal{D} : X \to [0,1]$ that satisfies the condition:*

$$\sum_{x \in X} \mathcal{D}(x) = 1.$$

*We say that $\mathcal{D}$ **assigns** probability $\mathcal{D}(x)$ to outcome x. The set X is referred to as the **support** of $\mathcal{D}$.*

*A special distribution is the **uniform** distribution over a finite set X, which assigns probability $1/|X|$ to every element of X.*

Let $\mathcal{D}$ be a probability distribution over X. We write $\Pr_{\mathcal{D}}[A]$ to denote the probability of an event A, where probabilities are according to distribution $\mathcal{D}$. Typically the distribution $\mathcal{D}$ is understood from context, and we omit it from the notation. Formally, an event is a subset of the support X, but it is typical to write Pr[cond] where "cond" is the condition that defines an event $A = \{x \in X \mid x \text{ satisfies condition cond}\}$. Interpreting A strictly as a set, we have $\Pr_{\mathcal{D}}[A] \overset{\text{def}}{=} \sum_{x \in A} \mathcal{D}(x)$.

The **conditional probability** of A given B is defined as $\Pr[A \mid B] \overset{\text{def}}{=} \Pr[A \wedge B]/\Pr[B]$. When $\Pr[B] = 0$, we let $\Pr[A \mid B] = 0$ by convention, to avoid dividing by zero.

Below are some convenient facts about probabilities:

$$\begin{aligned} \Pr[A] &= \Pr[A \mid B]\Pr[B] + \Pr[A \mid \neg B]\Pr[\neg B]; \\ \Pr[A \vee B] &\leqslant \Pr[A] + \Pr[B]. && \text{(union bound)} \end{aligned}$$

Precise Terminology

It is common and tempting to use the word "random" when one really means "*uniformly* at random." We'll try to develop the habit of being more precise about this distinction.

It is also tempting to describe an *outcome* as either random or uniform. For example, one might want to say that "the string x is random." But **an outcome is not random; the *process* that generated the outcome is random.** After all, there are many ways to come up with the same string x, and not all of them are random. So randomness is a property of the *process* and not an inherent property of the *result of the process.*

It's more precise and a better mental habit to say that an outcome is "*sampled* or *chosen* randomly," and it's even better to be precise about what the random process was. For example, "the string x is *chosen uniformly.*"

Notation in Pseudocode

We'll often describe algorithms/processes using pseudocode. In doing so, we will use several different operators whose meanings might be easily confused:

$\leftarrow$ When $\mathcal{D}$ is a probability distribution, we write "$x \leftarrow \mathcal{D}$" to mean "sample x according to the distribution $\mathcal{D}$."

If $\mathcal{A}$ is an algorithm that takes input and also makes some internal random choices, then it is natural to think of its output $\mathcal{A}(y)$ as a distribution — possibly a different distribution for each input y. Then we write "$x \leftarrow \mathcal{A}(y)$" to mean the natural thing: "run $\mathcal{A}$ on input y and assign the output to x."

We overload the "$\leftarrow$" notation slightly, writing "$x \leftarrow X$" when X is a *finite set* to mean that x is sampled from the *uniform distribution* over X.

$:=$ We write $x := y$ for assignments to variables: "take the value of expression y and assign it to variable x."

$\overset{?}{=}$ We write comparisons as $\overset{?}{=}$ (analogous to "==" in your favorite programming language). So $x \overset{?}{=} y$ doesn't modify x (or y), but rather it is an expression which returns `true` if x and y are equal.

You will often see this notation in the conditional part of an if-statement, but also in return statements as well. The following two snippets are equivalent:

```
return $x \overset{?}{=} y$
```

⇔

```
if $x \overset{?}{=} y$:
  return true
else:
  return false
```

In a similar way, we write $x \overset{?}{\in} S$ as an expression that evaluates to true if x is in the set S.

0.5 Asymptotics (Big-O)

Let $f : \mathbb{N} \to \mathbb{N}$ be a function. We characterize the asymptotic growth of f in the following ways:

$$\begin{aligned}
f(n) \text{ is } O(g(n)) &\overset{\text{def}}{\Leftrightarrow} \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
&\Leftrightarrow \exists c > 0 : \text{for all but finitely many } n : f(n) < c \cdot g(n) \\
f(n) \text{ is } \Omega(g(n)) &\overset{\text{def}}{\Leftrightarrow} \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \\
&\Leftrightarrow \exists c > 0 : \text{for all but finitely many } n : f(n) > c \cdot g(n) \\
f(n) \text{ is } \Theta(g(n)) &\overset{\text{def}}{\Leftrightarrow} f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n)) \\
&\Leftrightarrow 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
&\Leftrightarrow \exists c_1, c_2 > 0 : \text{for all but finitely many } n : \\
&\qquad\qquad c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)
\end{aligned}$$

1 One-Time Pad

Secure communication is the act of conveying information from a sender to a receiver, while simultaneously hiding it from everyone else. Secure communication is the oldest application of cryptography, and remains the centerpiece of cryptography to this day. After all, the word *cryptography* means “hidden writing” in Greek. So, secure communication is a natural place to start our study of cryptography.

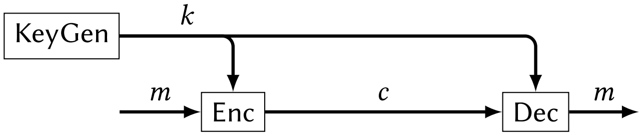
History provides us with roughly 2000 years of attempts to secure sensitive communications in the presence of eavesdroppers. Despite the many brilliant minds that lived during this period of time, almost no useful concepts remain relevant to modern cryptography. In fact, it was not clear how to even *formally define the goal* of secure communication until the 1940s. Today we use security definitions for encryption that were developed only in 1982 and 1990.

The *only* cryptographic method developed before 1900 that has stood the test of time is the **one-time pad**, which appears in some form in essentially every modern encryption scheme. In this chapter, we introduce the one-time pad and discuss its important characteristics. Along the way, we will start to get acclimated to cryptographic security definitions.

1.1 One-Time Pad Definition

For a long time the idea of one-time pad (OTP) was attributed to Gilbert Vernam, a telegraph engineer who patented the scheme in 1919. For that reason, one-time pad is sometimes called “Vernam’s cipher.” However, an earlier description of one-time pad was recently discovered in an 1882 text on telegraph encryption written by banker Frank Miller.[1]

The idea of one-time pad (and indeed, encryption in general) is for a sender and receiver to initially share a **key** k that was chosen according to a key-generation process that we will call KeyGen. When the sender wishes to securely send a message m (called the **plaintext**) to the receiver, she encrypts it using the encryption algorithm Enc and the key k. The result of encryption is a **ciphertext** c, which is sent to the receiver. The receiver can then use the **decryption** algorithm Dec with the same key k to recover the plaintext m.



One-time pad uses keys, plaintexts, and ciphertexts which are all λ-bit strings (*i.e.*, elements of $\{0,1\}^\lambda$). The choice of λ (length of plaintexts, ciphertext, and keys) is a public

[1] Steven M. Bellovin: “Frank Miller: Inventor of the One-Time Pad.” *Cryptologia* 35 (3), 2011.

parameter of the scheme, meaning that it does not need to be kept secret. The specific KeyGen, Enc, and Dec algorithms for one-time pad are given below:

Construction 1.1 (One-time pad)

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

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

Dec($k, c \in \{0,1\}^\lambda$):
    return $k \oplus c$
```

Recall that "$k \leftarrow \{0,1\}^\lambda$" means to sample k uniformly from the set of λ-bit strings. The definition of one-time pad mandates that the key should be chosen in exactly this way.

Example *Encrypting the following 20-bit plaintext m under the 20-bit key k using OTP results in the ciphertext c below:*

$$\begin{array}{rll} & \texttt{11101111101111100011} & (m) \\ \oplus & \texttt{00011001110000111101} & (k) \\ \hline & \texttt{11110110011111011110} & (c = \text{Enc}(k,m)) \end{array}$$

Decrypting the following ciphertext c using the key k results in the plaintext m below:

$$\begin{array}{rll} & \texttt{00001001011110010000} & (c) \\ \oplus & \texttt{10010011101011100010} & (k) \\ \hline & \texttt{10011010110101110010} & (m = \text{Dec}(k,c)) \end{array}$$

Note that Enc and Dec are essentially the same algorithm (return the xor of the two arguments). This results in some small level of convenience and symmetry when implementing one-time pad but you can think of it more as a coincidence than anything fundamental (see Exercises 1.10 & 2.3).

1.2 Properties of One-Time Pad

Correctness

The first property of one-time pad that we would like to verify is that the receiver does indeed recover the intended plaintext when decrypting the ciphertext. Written mathematically:

Claim 1.2 *For all $k, m \in \{0,1\}^\lambda$, it is true that* $\text{Dec}(k, \text{Enc}(k,m)) = m$.

Proof This follows by substituting the definitions of OTP Enc and Dec, along with the properties of xor listed in Chapter 0.2. For all $k, m \in \{0,1\}^\lambda$, we have:

$$\begin{aligned} \text{Dec}(k, \text{Enc}(k,m)) &= \text{Dec}(k, k \oplus m) \\ &= k \oplus (k \oplus m) \\ &= (k \oplus k) \oplus m \\ &= 0^\lambda \oplus m \\ &= m. \end{aligned}$$

Example *Encrypting the following plaintext m under the key k results in ciphertext c, as follows:*

$$\begin{array}{rlr} & \texttt{00110100110110001111} & (m) \\ \oplus & \texttt{11101010011010001101} & (k) \\ \hline & \texttt{11011110101100000010} & (c) \end{array}$$

Decrypting c using the same key k results in the original m:

$$\begin{array}{rlr} & \texttt{11011110101100000010} & (c) \\ \oplus & \texttt{11101010011010001101} & (k) \\ \hline & \texttt{00110100110110001111} & (m) \end{array}$$

Security

Suppose you encrypt a plaintext m and an eavesdropper eventually sees the resulting ciphertext. We want to say something like "the eavesdropper doesn't learn about m." We can be quite precise about what exactly the eavesdropper sees in this situation — in fact, the eavesdropper gets an output of the following algorithm:

```
VIEW($m \in \{0,1\}^\lambda$):
  $k \leftarrow \{0,1\}^\lambda$
  $c := k \oplus m$
  return $c$
```
.

This algorithm describes how the sender computes the values using secret values (choosing a key k in a specific way, and using the one-time-pad encryption procedure). It also describes that the eavesdropper sees *only* the ciphertext (but not the key).

This is a *randomized* algorithm, which you can see from the random choice of k. Even after fixing the input m, the output is not fixed. Instead of thinking of VIEW(m) as a fixed value, we will think of it as a *probability distribution*. So more precisely, we can say that an eavesdropper sees a **sample** from the distribution VIEW(m).

Example *Let's take $\lambda = 3$ and work out by hand the distributions* VIEW(010) *and* VIEW(111). *In each case* VIEW *chooses a value of k uniformly in $\{0,1\}^3$ — each of the possible values with probability 1/8. For each possible choice of k, we can compute what the output c will be:*

VIEW(010):

Pr	k	$c = k \oplus$ 010
⅛	000	010
⅛	001	011
⅛	010	000
⅛	011	001
⅛	100	110
⅛	101	111
⅛	110	100
⅛	111	101

VIEW(111):

Pr	k	$c = k \oplus$ 111
⅛	000	111
⅛	001	110
⅛	010	101
⅛	011	100
⅛	100	011
⅛	101	010
⅛	110	001
⅛	111	000

So the distribution VIEW(010) *assigns probabilty 1/8 to* 010, *probability 1/8 to* 011, *and so on.*

In this example, notice how every string in $\{0,1\}^3$ appears *exactly once* in the c column of VIEW(010). This means that VIEW assigns probability 1/8 to *every* string in $\{0,1\}^3$, which is just another way of saying that the distribution is the *uniform distribution* on $\{0,1\}^3$. The same can be said about the distribution VIEW(111), too. Both distributions are just the uniform distribution in disguise!

This property of one-time pad generalizes beyond these two specific examples. For any λ and any $m, m' \in \{0,1\}^\lambda$, the distributions VIEW(m) and VIEW(m') are identically distributed. Before we prove it, think about this fact from the eavesdropper's point of view. Someone chooses a plaintext m and shows you a sample from the distribution VIEW(m). But this is a distribution that you can sample from yourself, even if you don't know m. Indeed, you could have chosen an arbitrary m' and run VIEW(m') yourself, which would have induced the same distribution as VIEW(m). Truly, the ciphertext that you see (a sample from some distribution) can carry *no information* about m if it is possible to sample from the same ciphertext distribution without even knowing m!

Claim 1.3 *For every $m \in \{0,1\}^\lambda$, the distribution* VIEW(m) *is the* ***uniform distribution*** *on $\{0,1\}^\lambda$. Hence, for all $m, m' \in \{0,1\}^\lambda$, the distributions* VIEW(m) *and* VIEW(m') *are identical.*

Proof Arbitrarily fix $m, c \in \{0,1\}^\lambda$. We will calculate the probability that VIEW(m) produces output c. That event happens only when

$$c = k \oplus m \iff k = m \oplus c.$$

The equivalence follows from the properties of XOR given in Section 0.2. That is,

$$\Pr[\text{VIEW}(m) = c] = \Pr[k = m \oplus c],$$

where the probability is over uniform choice of $k \leftarrow \{0,1\}^\lambda$.

We have fixed specific m and c, so there is *only one* value of k that makes the condition true (encrypts m to c), and that value is exactly $m \oplus c$. Since k is chosen *uniformly* from $\{0,1\}^\lambda$, the probability of choosing the particular value $k = m \oplus c$ is $1/2^\lambda$. ■

Discussion

- **Isn't there a paradox?** Claim 1.2 says that c can always be decrypted to get m, but Claim 1.3 says that c contains no information about m! The answer to this riddle is that Claim 1.2 talks about what can be done with knowledge of the key k. Claim 1.3 is about the output distribution of the VIEW algorithm, which doesn't include k (see Exercise 1.7). In short, if you know k, then you can decrypt c to obtain m; if you don't know k, then c carries no information about m (in fact, it looks uniformly distributed). This is because m, c, k are all *correlated* in a delicate way.[2]

- **Isn't there another paradox?** Claim 1.3 says that the output of VIEW(m) doesn't depend on m, but the VIEW algorithm uses its argument m right there in the last line! The answer to this riddle is perhaps best illustrated by the previous examples of VIEW(010) and VIEW(111). The two tables of values are indeed different (so the

[2] This correlation is explored further in Chapter 3.

choice of $m \in \{\texttt{010}, \texttt{111}\}$ clearly has some effect), but they represent the *same probability distribution* (since order doesn't matter). Claim 1.3 considers only the resulting probability distribution.

Limitations

The keys in one-time pad are as long as the plaintexts they encrypt. This is more or less unavoidable (see Exercise 2.8) and leads to a kind of chicken-and-egg dilemma: If two users want to securely convey a λ-bit message, they first need to securely agree on a λ-bit string. Additionally, one-time pad keys can be used to securely encrypt only one plaintext (hence, "one-time" pad); see Exercise 1.5. Indeed, we can see that the VIEW subroutine in Claim 1.3 provides no way for a caller to guarantee that two plaintexts are encrypted with the same key, so it is not clear how to use Claim 1.3 to argue about what happens in one-time pad when keys are reused in this way.

Despite these limitations, one-time pad illustrates fundamental ideas that appear in most forms of encryption in this course.

Exercises

1.1. The one-time pad encryption of plaintext `mario` (written in ASCII) under key k is:

$$\texttt{1000010000000111010101000001110000011101}.$$

What is the one-time pad encryption of `luigi` under the same key?

1.2. Alice is using one-time pad and notices that when her key is the all-zeroes string $k = \texttt{0}^\lambda$, then $\textsf{Enc}(k, m) = m$ and her message is sent in the clear! To avoid this problem, she decides to modify KeyGen to exclude the all-zeroes key. She modifies KeyGen to choose a key uniformly from $\{\texttt{0},\texttt{1}\}^\lambda \setminus \{\texttt{0}^\lambda\}$, the set of all λ-bit strings except $\texttt{0}^\lambda$. In this way, she guarantees that her plaintext is never sent in the clear.

Is it still true that the eavesdropper's ciphertext distribution is uniform? Prove or disprove.

1.3. When Alice encrypts the key k itself using one-time pad, the ciphertext will always be the all-zeroes string! So if an eavesdropper sees the all-zeroes ciphertext, she learns that Alice encrypted the key itself. Does this contradict Claim 1.3? Why or why not?

1.4. Describe the flaw in this argument:

> Consider the following attack against one-time pad: upon seeing a ciphertext c, the eavesdropper tries every candidate key $k \in \{\texttt{0},\texttt{1}\}^\lambda$ until she has found the one that was used, at which point she outputs the plaintext m. This contradicts the argument in Section 1.2 that the eavesdropper can obtain no information about m by seeing the ciphertext.

1.5. Suppose Alice encrypts two plaintexts m and m' using one-time pad with the same key k. What information about m and m' is leaked to an eavesdropper by doing this (assume the eavesdropper knows that Alice has reused k)? Be as specific as you can!

1.6. A **known-plaintext attack** refers to a situation where an eavesdropper sees a ciphertext $c = \mathsf{Enc}(k, m)$ and also learns/knows what plaintext m was used to generate c.

(a) Show that a known-plaintext attack on OTP results in the attacker learning the key k.

(b) Can OTP be secure if it allows an attacker to recover the encryption key? Is this a contradiction to the security we showed for OTP? Explain.

1.7. Suppose we modify the subroutine discussed in Claim 1.3 so that it also returns k:

```
$\textsc{view}'(m \in \{0,1\}^\lambda)$:
    $k \leftarrow \{0,1\}^\lambda$
    $c := k \oplus m$
    return $(k, c)$
```

.

Is it still true that for every m, the output of $\textsc{view}'(m)$ is distributed uniformly in $(\{0,1\}^\lambda)^2$? Or is the output distribution different for different choice of m?

1.8. In this problem we discuss the security of performing one-time pad encryption twice:

(a) Consider the following subroutine that models the result of applying one-time pad encryption with two *independent* keys:

```
$\textsc{view}'(m \in \{0,1\}^\lambda)$:
    $k_1 \leftarrow \{0,1\}^\lambda$
    $k_2 \leftarrow \{0,1\}^\lambda$
    $c := k_2 \oplus (k_1 \oplus m)$
    return $c$
```

.

Show that the output of this subroutine is uniformly distributed in $\{0,1\}^\lambda$.

(b) What security is provided by performing one-time pad encryption twice with *the same key*?

1.9. We mentioned that one-time pad keys can be used to encrypt only one plaintext, and how this was reflected in the $\textsc{view}$ subroutine of Claim 1.3. Is there a similar restriction about re-using *plaintexts* in OTP (without re-using keys)? If an eavesdropper *knows* that the same plaintext is encrypted twice (but doesn't know what the plaintext is), can she learn anything? Does Claim 1.3 have anything to say about a situation where the same plaintext is encrypted more than once?

1.10. In this problem we consider a variant of one-time pad, in which the keys, plaintexts, and ciphertexts are all elements of $\mathbb{Z}_n$ instead of $\{0,1\}^\lambda$.

(a) What is the decryption algorithm that corresponds to the following encryption algorithm?

```
$\mathsf{Enc}(k, m \in \mathbb{Z}_n)$:
    return $(k + m) \,\%\, n$
```

.

(b) Show that the output of the following subroutine is uniformly distributed in $\mathbb{Z}_n$:

```
VIEW′($m \in \mathbb{Z}_n$):
  $k \leftarrow \mathbb{Z}_n$
  $c := (k + m) \% n$
  return $c$
```

.

2 The Basics of Provable Security

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

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

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

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

2.1 Library-Based Security Reasoning

Motivation

Suppose you write a program $\mathcal{P}$ that calls a SORT subroutine, which is implemented using BubbleSort. Changing the implementation of SORT to use MergeSort might make $\mathcal{P}$ run faster but it will not change anything about what $\mathcal{P}$ actually computes. This is because on every input both BubbleSort and MergeSort will give the same output — they have identical input-output behavior. You could say that the calling program $\mathcal{P}$ can't tell whether it's calling SORT implemented by BubbleSort and QuickSort.[1]

What if the output behavior of a subroutine is *randomized?* In this case, we say that two subroutines have identical input-output behavior if for every input, both generate the *same output distribution.* Consider the following two subroutines:

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

```
ROLL-D8():
  $a \leftarrow \{0,1\}$
  $b \leftarrow \{0,1\}$
  $c \leftarrow \{0,1\}$
  return $4a + 2b + c$
```

[1] Here we are making an assumption that the SORT subroutine can influence the program's behavior *only* through how it computes its output. One example of how to violate this assumption would be if the program measures how much time the subroutine takes. We discuss this assumption later on.

Both produce a return value that is distributed uniformly in $\mathbb{Z}_8$ (think of the first one as tossing an 8-sided die, and the other one as tossing 3 coins). Hence, the subroutines have identical input-output behavior. As above, replacing one by the other will not change the overall behavior of a program. The calling program can't tell which of the two subroutines is being used.

In both cases, the calling program can call a subroutine but can't tell which of two possible implementations is actually used. **This information is hidden** from the calling program! The most fundamental goal in cryptography is to hide information, which suggests to use the terminology of subroutines, input-output behavior, calling programs, etc. to reason about security.

Libraries & Interfaces

The main theme of this chapter is that if two subroutines have identical input-output behavior, then no calling program can tell which one is being used. The choice of subroutine is hidden from the calling program.

We now introduce some terminology and notation for these concepts.

Definition 2.1 (Libraries) *A **library** $\mathcal{L}$ is a collection of subroutines and private/static variables. A library's **interface** consists of the names, argument types, and output type of all of its subroutines. If a program $\mathcal{A}$ includes calls to subroutines in the interface of $\mathcal{L}$, then we write $\mathcal{A} \diamond \mathcal{L}$ to denote the result of **linking** $\mathcal{A}$ to $\mathcal{L}$ in the natural way (answering those subroutine calls using the implementation specified in $\mathcal{L}$). We write $\mathcal{A} \diamond \mathcal{L} \Rightarrow z$ to denote the event that program $\mathcal{A} \diamond \mathcal{L}$ outputs the value z.*

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

```
$\mathcal{A}_1$
$r_1 := \textsc{rand}(6)$
$r_2 := \textsc{rand}(6)$
return $r_1 \overset{?}{=} r_2$
```

```
$\mathcal{A}_2$
$r := \textsc{rand}(6)$
return $r \geqslant 3$
```

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

```
$\mathcal{L}_2$
$\textsc{rand}(n)$:
  return 0
```

We can consider all 4 ways of linking a calling program to a library:

$\mathcal{A}_1 \diamond \mathcal{L}_1$: *rolls two fair 6-sided dice and checks whether they match. This combined program outputs `true` with probability 1/6.*

$\mathcal{A}_1 \diamond \mathcal{L}_2$: *sets r_1 and r_2 to both be 0, so outputs `true` with probability 1.*

$\mathcal{A}_2 \diamond \mathcal{L}_1$: *rolls a 6-sided die (with faces 0 through 5) and checks whether the result is at least 3. This combined program outputs `true` with probability 1/2.*

$\mathcal{A}_2 \diamond \mathcal{L}_2$: *sets r to 0, so outputs `true` with probability 0.*

Example *A library can contain several subroutines and variables that are kept static between subroutine calls. For example, the following library allows the caller to specify a set S and then choose* *randomly from this set* without replacement. *Every time a value is chosen from S, it is removed from S:*

```
$\mathcal{L}$
$S := \emptyset$

RESET($S'$):
  $S := S'$

SAMP():
  if $S = \emptyset$ then return err
  $r \leftarrow S$
  $S := S \setminus \{r\}$
  return $r$
```

Discussion

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

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 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 (*e.g.*, the S variable in the previous example). Calling programs cannot directly access internal variables in a library. If we want a calling program to have access to some internal variables, we must explicitly add a subroutine/accessor 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.[2]

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

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

We have already seen several examples of different libraries that have the same input-output behavior. A library that provides a SORT algorithm might implement it using quicksort or mergesort. A library that provides a ROLL-D8 algorithm might implement it using an 8-sided die or 3 coin flips.

We have also argued that if two libraries have identical input-output behavior, then no calling program will behave differently when linked to one or the other. This turns out to be a convenient way to *define* what it means to have identical input-output behavior, and it works even if the libraries use randomness.

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

Discussion

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

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

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

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

- Taking the previous observation even further, the definition applies against calling programs $\mathcal{A}$ that "know everything" about (more formally, whose code is allowed to depend arbitrarily on) the two libraries. This is a reflection of **Kerckhoffs' principle**, which roughly says "assume that the attacker has full knowledge of the system."[3]

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

  This subtlety is reflected in Definition 2.2 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 distinguisher cannot depend on the random choices made by the library, since the choice of randomness "happens after" the distinguisher is fixed.

  Kerckhoff's Principle, adapted to our terminology:

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

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

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

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

Suppose two libraries $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are interchangeable: two libraries that are internally different but appear the same to all calling programs. Think of the internal *differences*

[3] *"Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."* Auguste Kerckhoffs, 1883. Translation: [The method] must not be required to be secret, and it can fall into the enemy's hands without causing inconvenience.

between the two libraries as **information that is perfectly hidden** to the calling program. If the information weren't perfectly hidden, then the calling program could get a whiff of whether it was linked to $\mathcal{L}_{\text{left}}$ or $\mathcal{L}_{\text{right}}$, and use it to act differently in those two cases. But such a calling program would contradict the fact that $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$.

This line of reasoning leads to:

> **The Prime Directive of Security Definitions:**
>
> *If two libraries are interchangeable, then their common interface leaks no information about their internal differences.*

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

An example: One-time pad's uniformity property

In Claim 1.3 we proved that a particular subroutine VIEW generates uniformly distributed output, no matter what input is given to it. We can restate that claim in the new terminology of libraries:

Claim 2.3 (OTP rule) *The following two libraries are interchangeable (*i.e., $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$*):*

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

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

You can tell just by looking that the two libraries $\mathcal{L}_{\text{otp-}\star}$ have the same *interface*. Claim 2.3 says something more specific: the two libraries in fact have the same input-output *behavior.*

2.2 A General-Purpose Security Definition for Encryption

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

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

Syntax of Encryption

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

Definition 2.4 (Encryption syntax)

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

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

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

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

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

Correctness

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

Definition 2.5

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

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

The definition is expressed in terms of a probability, because $\textsf{Enc}$ is allowed to be a randomized algorithm.

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

One-Time Secrecy

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

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

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

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

The two libraries should look something like:

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

and

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

.

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

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

Putting these ideas together, we obtain the following definition:

Definition 2.6 (One-time secrecy) *Let Σ be an encryption scheme. We say that Σ is* **(perfectly) one-time secret** *if $\mathcal{L}^{\Sigma}_{\text{ots-L}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-R}}$, where:*

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

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

This security notion is often called *perfect secrecy* in other sources.[4]

Interpreting/Critiquing a Security Definition

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

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

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

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

- The definition assumes that a key is used to encrypt only a single plaintext: each time QUERY is called, a fresh key is chosen as k. Of course, it is *possible* for different calls to QUERY to use the same value k by chance. But there is no way for the calling program to *ensure* that two plaintexts are encrypted under the same key. This means that one-time secrecy gives no guarantee about what happens when a key is (purposefully, consistently — not just by chance) used to encrypt several plaintexts.

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

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

- The definition does not guarantee that the ciphertext hides *the fact that a plaintext comes from the set $\mathcal{M}$.* In both libraries, the plaintext (m_L or m_R) is an element of $\mathcal{M}$,

[4] Personally, I think that using the term "perfect" leads to an impression that one-time pad should *always* be favored over any other kind of encryption scheme (presumably with only "imperfect" security). But if you want encryption, then you should almost never favor plain old one-time pad.

whereas only the *differences* between the libraries are hidden. More concretely, for one-time pad we have $\mathcal{M} = \{\texttt{0},\texttt{1}\}^\lambda$. The one-time secrecy definition does not hide the fact that the plaintext is λ bits long, and indeed, you can quite clearly deduce the length of the plaintext from the length of the one-time pad ciphertext. If you want to hide the length of a plaintext, then you need to find a security definition that captures that guarantee. It is not a guarantee provided by the one-time secrecy definition.

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

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

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

2.3 How to Prove Security with the Hybrid Technique

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

A Useful Chaining Lemma

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

Lemma 2.7 (Chaining) *If $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$ then $\mathcal{L}^* \diamond \mathcal{L}_{\text{left}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{right}}$ for any library $\mathcal{L}^*$.*

Proof Take an arbitrary calling program $\mathcal{A}$. We would like 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. 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 $\mathcal{L}_{\text{left}}$. After all, $\mathcal{A} \diamond \mathcal{L}^*$ is some program that makes calls to the interface of $\mathcal{L}_{\text{left}}$. Since $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$, swapping $\mathcal{L}_{\text{left}}$ for $\mathcal{L}_{\text{right}}$ has no effect on the output of any calling program. In particular, it has no effect when the calling program happens to be $\mathcal{A} \diamond \mathcal{L}^*$. Hence we have:

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

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

One-Time Secrecy of One-Time Pad

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

Theorem 2.8 *Let* OTP *denote the one-time pad encryption scheme (Construction 1.1). Then* OTP *has one-time secrecy. That is,* $\mathcal{L}^{\text{OTP}}_{\text{ots-L}} \equiv \mathcal{L}^{\text{OTP}}_{\text{ots-R}}$.

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

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

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

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

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

Proof As described above, we will prove that

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

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

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

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$
QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $k \leftarrow \{0,1\}^\lambda$
    $c = k \oplus m_L$
    return $c$
```

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

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

```
QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $c$ = QUERY'($m_L$)
    return $c$
```

$\diamond$

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

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 $\mathcal{L}_{\text{otp-real}}$ library from Claim 2.3 (with the subroutine name changed to avoid naming conflicts). This is no accident!

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

```
QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $c$ = QUERY'($m_L$)
    return $c$
```

$\diamond$

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

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

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

```
QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $c$ = QUERY'($m_R$)
    return $c$
```

$\diamond$

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

The argument to QUERY′ has been changed from m_L to m_R. This has no effect on the library's behavior since QUERY′ *does not actually use its argument* in these hybrids.

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

Having made this crucial change, we can now perform the same sequence of steps, but in reverse.

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

QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $c$ = QUERY′($m_R$)
    return $c$

$\diamond$

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

$\mathcal{L}_{\text{otp-rand}}$ has been replaced with $\mathcal{L}_{\text{otp-real}}$. Again, this has no effect on the library's behavior, due to Claim 2.3. Note that the chaining lemma is being applied with respect to a different common $\mathcal{L}^*$ than before (now m_R instead of m_L is being used).

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

$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$
QUERY($m_L, m_R \in \{0,1\}^\lambda$):
    $k \leftarrow \{0,1\}^\lambda$
    $c = k \oplus m_R$
    return $c$
```

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

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

Summary of the Hybrid Technique

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

- Proving security amounts to showing that two particular libraries, say $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$, are interchangeable.
- To show this, we show a sequence of hybrid libraries, beginning with $\mathcal{L}_{\text{left}}$ and ending with $\mathcal{L}_{\text{right}}$. The hybrid sequence corresponds to a sequence of *allowable modifications* to the library. Each modification is small enough that we can easily justify why it doesn't affect the calling program's output probability.
- Simple things like factoring out & inlining subroutines, changing unused variables, consistently renaming variables, removing & changing unreachable statements, or unrolling loops are always "allowable" modifications in a hybrid proof. As we progress in the course, we will add to our toolbox of allowable modifications. For instance, if we want to prove security of some complicated system that uses a one-time-secret encryption Σ as one of its components, then we are allowed to replace $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ with $\mathcal{L}^{\Sigma}_{\text{ots-R}}$ as one of the steps in the hybrid proof.

2.4 How to Demonstrate Insecurity with Attacks

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

Below is an example of an insecure encryption scheme:

Construction 2.9

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

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

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

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

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

Claim 2.10 *Construction 2.9 does* ***not*** *have one-time secrecy.*

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

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

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

Putting it all together, we define the following distinguisher:

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

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

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

$\diamond$

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

.

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

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

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

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

$\diamond$

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

.

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

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

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

Exercises

2.1. Show that the following libraries are interchangeable:

```
$\mathcal{L}_1$
QUERY$(m \in \{0,1\}^\lambda)$:
  $x \leftarrow \{0,1\}^\lambda$
  $y := x \oplus m$
  return $(x, y)$
```

```
$\mathcal{L}_2$
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.

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

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

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

★ 2.3. 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* $(\mathbb{G}, \otimes)$:

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

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

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

Dec($k, c$):
    ??
```

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

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

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

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

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

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.

2.5. 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.

```
$\mathcal{K} = \{1, \ldots, 9\}$
$\mathcal{M} = \{1, \ldots, 9\}$
$\mathcal{C} = \mathbb{Z}_{10}$

KeyGen:
    $k \leftarrow \{1, \ldots, 9\}$
    return $k$

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

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

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

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 $m = \texttt{1101100001}$.

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

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

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

2.8. 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.

2.9. 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:

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

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

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

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

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

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

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

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

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

2.11. 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.

2.12. 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 $\texttt{0}$ with probability 0.4 and $\texttt{1}$ 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 $\textsc{query}(\texttt{01011},\texttt{10001})$ and receives the result $\texttt{01101}$. 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.

3 Secret Sharing

DNS is the system that maps human-memorable Internet domains like `irs.gov` to machine-readable IP addresses like `166.123.218.220`. If an attacker can masquerade as the DNS system and convince your computer that `irs.gov` actually resides at some other IP address, it might result in a bad day for you.

To protect against these kinds of attacks, a replacement called DNSSEC has been proposed. DNSSEC uses cryptography to make it impossible to falsify a domain-name mapping. The cryptography required to authenticate DNS mappings is certainly interesting, but an even more fundamental question remains: *Who can be trusted with the master cryptographic keys to the system?* The non-profit organization in charge of these kinds of things (ICANN) has chosen the following system. The master key is split into 7 pieces and distributed on smart cards to 7 geographically diverse people, who keep them in safe-deposit boxes.

> *At least five key-holding members of this fellowship would have to meet at a secure data center in the United States to reboot [DNSSEC] in case of a very unlikely system collapse.*
>
> *"If you round up five of these guys, they can decrypt [the root key] should the West Coast fall in the water and the East Coast get hit by a nuclear bomb," [said] Richard Lamb, program manager for DNSSEC at ICANN.*[1]

How is it possible that *any* 5 out of the 7 key-holders can reconstruct the master key, but (presumably) 4 out of the 7 cannot? The solution lies in a cryptographic tool called a **secret-sharing scheme**, the topic of this chapter.

3.1 Definitions

We begin by introducing the syntax of a secret-sharing scheme:

Definition 3.1 (Secret-sharing)

*A t**-out-of-**n **threshold secret-sharing scheme (TSSS)** consists of the following algorithms:*

- Share*: a randomized algorithm that takes a* ***message*** *$m \in \mathcal{M}$ as input, and outputs a sequence $\boldsymbol{s} = (s_1, \ldots, s_n)$ of* ***shares****.*
- Reconstruct*: a deterministic algorithm that takes a collection of t or more shares as input, and outputs a message.*

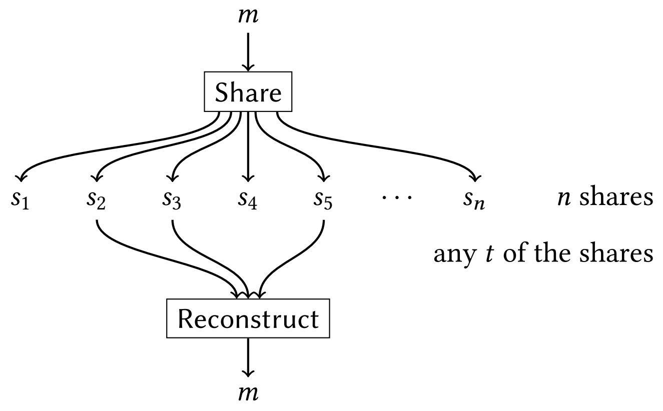
We call $\mathcal{M}$ the ***message space*** *of the scheme, and t its* ***threshold****. As usual, we refer to the parameters/components of a scheme Σ as $\Sigma.t$, $\Sigma.n$, $\Sigma.\mathcal{M}$, $\Sigma.$Share, $\Sigma.$Reconstruct.*

[1] http://www.livescience.com/6791-internet-key-holders-insurance-cyber-attack.html

In secret-sharing, we number the users as $\{1,\ldots,n\}$, with user i receiving share s_i. Let $U \subseteq \{1,\ldots,n\}$ be a subset of users. Then $\{s_i \mid i \in U\}$ refers to the set of shares belonging to users U. If $|U| \geqslant t$, we say that U is **authorized**; otherwise it is **unauthorized**. The goal of secret sharing is for all authorized sets of users/shares to be able to reconstruct the secret, while all unauthorized sets learn nothing.

Definition 3.2 (TSSS correctness)

*A t-out-of-n TSSS satisfies **correctness** if, for all authorized sets $U \subseteq \{1,\ldots,n\}$ (i.e., $|U| \geqslant t$) and for all $\boldsymbol{s} \leftarrow \textsf{Share}(m)$, we have $\textsf{Reconstruct}(\{s_i \mid i \in U\}) = m$.*



Security Definition

We'd like a security guarantee that says something like:

> *if you have an unauthorized set of shares, then you learn no information about the choice of secret message.*

To translate this informal statement into a formal security definition, we follow the The Prime Directive of Security Definitions and define two libraries that allow the calling program to learn a set of shares (for an *unauthorized* set), and that differ only in which secret is shared. If the two libraries are interchangeable, then we conclude that seeing an unauthorized set of shares leaks no information about the choice of secret message. The definition looks like this:

Definition 3.3 (TSSS security)

*Let Σ be a threshold secret-sharing scheme. We say that Σ is **secure** if $\mathcal{L}^{\Sigma}_{\text{tsss-L}} \equiv \mathcal{L}^{\Sigma}_{\text{tsss-R}}$, where:*

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

```
QUERY($m_L, m_R \in \Sigma.\mathcal{M}, U$):
  if $|U| \geqslant \Sigma.t$: return err
  $s \leftarrow \Sigma.\textsf{Share}(m_L)$
  return $\{s_i \mid i \in U\}$
```

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

```
QUERY($m_L, m_R \in \Sigma.\mathcal{M}, U$):
  if $|U| \geqslant \Sigma.t$: return err
  $s \leftarrow \Sigma.\textsf{Share}(m_R)$
  return $\{s_i \mid i \in U\}$
```

In an attempt to keep the notation uncluttered, we have not written the type of the argument U, which is $U \subseteq \{1,\ldots,\Sigma.n\}$.

Discussion

- Similar to the definition of one-time secrecy of encryption, we let the calling program choose the two secret messages that will be shared. As before, this models the fact that an adversary may have partial knowledge or influence over what inputs might be used in the secret-sharing scheme.

- The calling program also chooses the set U of users' shares to obtain. The libraries make it impossible for the calling program to obtain the shares of an *authorized* set (returning `err` in that case).

- Consider a 6-out-of-10 threshold secret-sharing scheme. With the libraries above, the calling program can receive the shares of users $\{1, \ldots, 5\}$ (an unauthorized set) in one call to QUERY, and then receive the shares of users $\{6, \ldots, 10\}$ in another call. It might seem like the calling program can then combine these shares to reconstruct the secret (if the same message was shared in both calls). However, this is *not* the case because these two sets of shares came from two *independent executions* of the Share algorithm. Shares generated by one call to Share should not be expected to function with shares generated by another call, even if both calls to Share used the same secret message.

- Recall that in our style of defining security using libraries, it is only the internal *differences* between the libraries that must be hidden. Anything that is the *same* between the two libraries need not be hidden. One thing that is the same for the two libraries here is the fact that they output the shares belonging to the same set of users U. This security definition does not require shares to hide *which user they belong to.* Indeed, you can modify a secret-sharing scheme so that each user's identity is appended to his/her corresponding share, and the result would still satisfy the security definition above.

- Just like the encryption definition does not address the problem of key distribution, the secret-sharing definition does not address the problem of *who* should run the Share algorithm (if its input m is so secret that it cannot be entrusted to any single person), or *how* the shares should be deliverd to the n different users. Those concerns are considered out of scope by the problem of secret-sharing (although we later discuss clever approaches to the first problem). Rather, the focus is simply on whether it is even possible to encode data in such a way that an unauthorized set of shares gives no information about the secret.

3.2 Insecure Aproach for Secret Sharing

To understand the security requirement for secret sharing, it helps to see an example of an "obvious" approach for secret sharing that is actually *insecure.*

Let's consider a 5-out-of-5 secret-sharing scheme. This means we want to split a secret into 5 pieces so that any 4 of the pieces leak nothing. One way you might think to do this is to *literally chop up the secret* into 5 pieces. For example, if the secret is 500 bits, you might give the first 100 bits to user 1, the second 100 bits to user 2, and so on.

Construction 3.4 (Insecure TSSS)

```
$\mathcal{M} = \{0,1\}^{500}$
$t = 5$
$n = 5$

Share($m$):
  split $m$ into $m = s_1 \| \cdots \| s_5$,
     where each $|s_i| = 100$
  return $(s_1, \ldots, s_5)$

Reconstruct($s_1, \ldots, s_5$):
  return $s_1 \| \cdots \| s_5$
```

It is true that the secret can be constructed by concatenating all 5 shares, and so this construction satisfies the correctness property. (The only authorized set is the set of all 5 users, so we write Reconstruct to expect all 5 shares.)

However, the scheme is **insecure** (as promised). Suppose you have even just 1 share. It is true that you don't know the secret *in its entirety*, but the security definition (for 5-out-of-5 secret sharing) demands that a single share reveals *nothing* about the secret. Of course knowing 100 bits of something is not the same as than knowing *nothing* about it.

We can leverage this observation to make a more formal attack on the scheme, in the form of a distinguisher between the two $\mathcal{L}_{\text{tsss-}\star}$ libraries. As an extreme case, we can distinguish between shares of an all-0 secret and shares of an all-1 secret:

```
$\mathcal{A}$
$s_1 := \text{QUERY}(0^{500}, 1^{500}, \{1\})$
return $s_1 \overset{?}{=} 0^{100}$
```

When this distinguisher is linked to $\mathcal{L}_{\text{tsss-L}}$, it receives a share of 0^{500}, which will itself be a string of all zeroes. In this case, the distinguisher outputs 1 with probability 1. When linked to $\mathcal{L}_{\text{tsss-R}}$, it receives a share of 1^{500} which will be a string of all ones. In this case, the distinguisher outputs 1 with probability 0.

We have constructed a calling program which behaves very differently (indeed, as differently as possible) in the presence of the two libraries. Hence, this secret-sharing scheme is not secure.

Hopefully this example demonstrates one of the main challenges (and amazing things) about secret-sharing schemes. It is easy to reveal information about the secret *gradually* as more shares are obtained, like in this insecure example. However, the security definition of secret sharing is that the shares must leak *absolutely no information* about the secret, until the number of shares passes the threshold value.

3.3 A Simple 2-out-of-2 Scheme

Believe it or not, we have already seen a simple secret-sharing scheme! In fact, it might even be best to think of one-time pad as the simplest secret-sharing scheme, since by itself it is not so useful for encryption.

Construction 3.5 (2-out-of-2 TSSS)

```
$\mathcal{M} = \{0,1\}^{\ell}$
$t = 2$
$n = 2$

Share($m$):
  $s_1 \leftarrow \{0,1\}^{\ell}$
  $s_2 := s_1 \oplus m$
  return $(s_1, s_2)$

Reconstruct($s_1, s_2$):
  return $s_1 \oplus s_2$
```

Since it's a 2-out-of-2 scheme, the only authorized set of users is $\{1,2\}$, so have written Reconstruct to expect both shares s_1 and s_2 as its inputs. Correctness follows easily from what we've already learned about the properties of XOR.

Example *If we want to share the string* $m = \texttt{1101010001}$ *then the* Share *algorithm might choose*

$$\begin{aligned} s_1 &:= \texttt{0110000011} \\ s_2 &:= s_1 \oplus m \\ &= \texttt{0110000011} \oplus \texttt{1101010001} = \texttt{1011010010}. \end{aligned}$$

Then the two shares can be recombined by XOR*ing them together, since:*

$$s_1 \oplus s_2 = \texttt{0110000011} \oplus \texttt{1011010010} = \texttt{1101010001} = m.$$

Theorem 3.6 *Construction 3.5 is a secure 2-out-of-2 threshold secret-sharing scheme.*

Proof Let Σ denote Construction 3.5. We will show that $\mathcal{L}^{\Sigma}_{\text{tsss-L}} \equiv \mathcal{L}^{\Sigma}_{\text{tsss-R}}$ using a hybrid proof.

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

```
$\mathcal{L}^{\Sigma}_{\text{tsss-L}}$
QUERY$(m_L, m_R, U)$:
  if $|U| \geqslant 2$: return err
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_L$
  return $\{s_i \mid i \in U\}$
```

As usual, the starting point is $\mathcal{L}^{\Sigma}_{\text{tsss-L}}$, shown here with the details of the secret-sharing scheme filled in (and the types of the subroutine arguments omitted to reduce clutter).

```
QUERY$(m_L, m_R, U)$:
  if $|U| \geqslant 2$: return err
  if $U = \{1\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_L$
    return $\{s_1\}$
  elsif $U = \{2\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_L$
    return $\{s_2\}$
  else return $\emptyset$
```

It has no effect on the library's behavior if we duplicate the main body of the library into 3 branches of a new if-statement. The reason for doing so is that the scheme generates s_1 and s_2 differently. This means that our proof will eventually handle the 3 different unauthorized sets ($\{1\}$, $\{2\}$, and $\emptyset$) in fundamentally different ways.

```
QUERY$(m_L, m_R, U)$:
  if $|U| \geqslant 2$: return err
  if $U = \{1\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_R$
    return $\{s_1\}$
  elsif $U = \{2\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_L$
    return $\{s_2\}$
  else return $\emptyset$
```

The definition of s_2 has been changed in the first if-branch. This has no effect on the library's behavior since s_2 is never actually used in this branch.

```
QUERY$(m_L, m_R, U)$:
  if $|U| \geqslant 2$: return err
  if $U = \{1\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_R$
    return $\{s_1\}$
  elsif $U = \{2\}$:
    $s_2 \leftarrow$ QUERY'$(m_L, m_R)$
    return $\{s_2\}$
  else return $\emptyset$
```

$\diamond$

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$
QUERY'$(m_L, m_R)$:
  $k \leftarrow \{0,1\}^\ell$
  $c := k \oplus m_L$
  return $c$
```

Recognizing the second branch of the if-statement as a one-time pad encryption (of m_L under key s_1), we factor out the generation of s_2 in terms of the library $\mathcal{L}^{OTP}_{\text{ots-L}}$ from the one-time secrecy definition. This has no effect on the library's behavior. Importantly, the subroutine in $\mathcal{L}^{OTP}_{\text{ots-L}}$ expects *two arguments*, so that is what we must pass. We choose to pass m_L and m_R for reasons that should become clear very soon.

```
QUERY$(m_L, m_R, U)$:
  if $|U| \geqslant 2$: return err
  if $U = \{1\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_R$
    return $\{s_1\}$
  elsif $U = \{2\}$:
    $s_2 \leftarrow$ QUERY'$(m_L, m_R)$
    return $\{s_2\}$
  else return $\emptyset$
```

$\diamond$

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$
QUERY'$(m_L, m_R)$:
  $k \leftarrow \{0,1\}^\ell$
  $c := k \oplus m_R$
  return $c$
```

We have replaced $\mathcal{L}^{OTP}_{\text{ots-L}}$ with $\mathcal{L}^{OTP}_{\text{ots-R}}$. From the one-time secrecy of one-time pad (and the composition lemma), this change has no effect on the library's behavior.

```
QUERY($m_L, m_R, U$):
  if $|U| \geqslant 2$: return err
  if $U = \{1\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_R$
    return $\{s_1\}$
  elsif $U = \{2\}$:
    $s_1 \leftarrow \{0,1\}^\ell$
    $s_2 := s_1 \oplus m_R$
    return $\{s_2\}$
  else return $\emptyset$
```

A subroutine has been inlined; no effect on the library's behavior.

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

```
$\mathcal{L}^{\Sigma}_{\text{tsss-R}}$
QUERY($m_L, m_R, U$):
  if $|U| \geqslant 2$: return err
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_R$
  return $\{s_i \mid i \in U\}$
```

The code has been simplified. Specifically, the branches of the if-statement can all be unified, with no effect on the library's behavior. The result is $\mathcal{L}^{\Sigma}_{\text{tsss-R}}$.

We showed that $\mathcal{L}^{\Sigma}_{\text{tsss-L}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-5}} \equiv \mathcal{L}^{\Sigma}_{\text{tsss-R}}$, and so the secret-sharing scheme is secure. ■

We in fact proved a slightly more general statement. The only property of one-time pad we used was its one-time secrecy. Substituting one-time pad for any other one-time secret encryption scheme would still allow the same proof to go through. So we actually proved the following:

Theorem 3.7 *If Σ is an encryption scheme with one-time secrecy, then the following 2-out-of-2 threshold secret-sharing scheme $\mathcal{S}$ is secure:*

```
$\mathcal{M} = \Sigma.\mathcal{M}$    Share($m$):                              Reconstruct($s_1, s_2$):
      $t = 2$                  $s_1 \leftarrow \Sigma$.KeyGen                 return $\Sigma$.Dec($s_1, s_2$)
      $n = 2$                  $s_2 \leftarrow \Sigma$.Enc($s_1, m$)
                               return ($s_1, s_2$)
```

3.4 Polynomial Interpolation

You are probably familiar with the fact that two points determine a line (in Euclidean geometry). It is also true that 3 points determine a parabola, and so on. The next secret-sharing scheme we discuss is based on the following principle:

> $d + 1$ points determine a *unique* degree-d polynomial, and this is true even working modulo a prime.

A note on terminology: If f is a polynomial that can be written as $f(x) = \sum_{i=0}^{d} f_i x^i$, then we say that f is a **degree-**d polynomial. It would be more technically correct to say that the degree of f is *at most d* since we allow the leading coefficient f_d to be zero. For convenience, we'll stick to saying "degree-d" to mean "degree at most d."

Theorem 3.8 (Poly Interpolation) *Let p be a prime, and let $\{(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})\} \subseteq (\mathbb{Z}_p)^2$ be a set of points whose x_i values are all distinct. Then there is a* ***unique*** *degree-d polynomial f with coefficients in $\mathbb{Z}_p$ that satisfies $y_i \equiv_p f(x_i)$ for all i.*

Proof Let $f(x) = \sum_{i=0}^{d} f_i x^i$ be a degree-d polynomial. Consider the problem of evaluating f on a set of points $x_1, \ldots, x_{d+1}$. We can express outputs of f as a linear function of the coefficients of f in the following way:

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{d+1}) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^d \\ & & \vdots & & \ddots & \vdots \\ 1 & x_{d+1} & x_{d+1}^2 & x_{d+1}^3 & \cdots & x_{d+1}^d \end{bmatrix}}_{V} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_d \end{bmatrix}.$$

What's notable about this expression is that V is a special kind of matrix called a **Vandermonde** matrix. A Vandermonde matrix is determined by the values $x_1, \ldots, x_{d+1}$, where the (i, j) entry of the matrix is x_i^{j-1}. One important property of Vandermonde matrices (that we won't prove here) is that the determinant of a square Vandermonde matrix is:

$$\det(V) = \prod_{i<j} (x_j - x_i).$$

The Vandermonde matrix V in our expression is square, having dimensions $(d+1) \times (d+1)$. Also, since all of the x_i values are distinct, the expression for the determinant must be non-zero. That means V is **invertible!**

So, knowing $\{(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})\}$, we can solve for the coefficients of f in the following equation:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{d+1} \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^d \\ & & \vdots & & \ddots & \vdots \\ 1 & x_{d+1} & x_{d+1}^2 & x_{d+1}^3 & \cdots & x_{d+1}^d \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_d \end{bmatrix}$$

$$\implies \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_d \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^d \\ & & \vdots & & \ddots & \vdots \\ 1 & x_{d+1} & x_{d+1}^2 & x_{d+1}^3 & \cdots & x_{d+1}^d \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{d+1} \end{bmatrix}.$$

Note the matrix inverse in the second equation. There is a unique solution for the vector $\boldsymbol{f} = (f_0, \ldots, f_d)$, hence a unique degree-$d$ polynomial f fitting the points.

The proof was written as if the linear algebra was over the real numbers (or complex numbers if you prefer). Indeed, the statement is true for real and complex numbers. However, all of the logic still holds when the linear equations are over $\mathbb{Z}_p$, when p is a prime. Formally, this is because $\mathbb{Z}_p$ is a *field* (working modulo p, you have addition, multiplication, and *inverses* of nonzero elements). Most of what you know about linear algebra "just works" when matrix operations are in a field. Replacing the "=" sign (integer equality) with "$\equiv_p$" (congruence modulo p), and all the steps of the proof still hold. ■

3.5 Shamir Secret Sharing

Part of the challenge in designing a secret-sharing scheme is making sure that *any* authorized set of users can reconstruct the secret. We have just seen that *any* $d + 1$ points on a degree-d polynomial are enough to reconstruct the polynomial. So a natural approach for secret sharing is to let each user's share be a point on a polynomial.

That's exactly what **Shamir secret sharing** does. To share a secret $m \in \mathbb{Z}_p$ with threshold t, first choose a degree-$(t-1)$ polynomial f that satisfies $f(0) \equiv_p m$, with all other coefficients chosen uniformly in $\mathbb{Z}_p$. The ith user receives the point $(i, f(i) \% p)$ on the polynomial. The interpolation theorem shows that any t shares can uniquely determine the polynomial f, and hence recover the secret $f(0) \equiv_p m$.

Construction 3.9 (Shamir SSS)

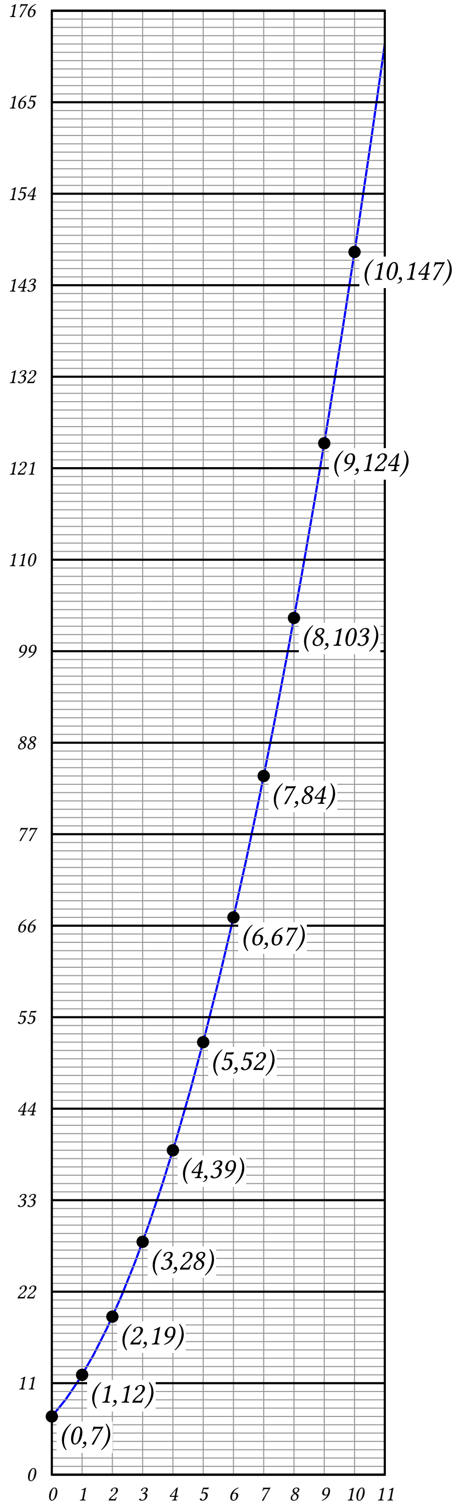
```
$\mathcal{M} = \mathbb{Z}_p$
$p$ : prime
$n < p$
$t \leqslant n$

Share($m$):
    $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
    $f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
    for $i = 1$ to $n$:
        $s_i := (i, f(i) \% p)$
    return $\boldsymbol{s} = (s_1, \ldots, s_n)$

Reconstruct($\{s_i \mid i \in U\}$):
    $f(x)$ := unique degree-$(t-1)$
            polynomial mod $p$ passing
            through points $\{s_i \mid i \in U\}$
    return $f(0)$
```

Correctness follows from the interpolation theorem.

Example

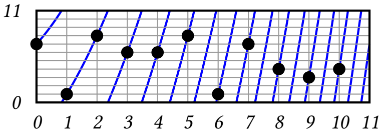


Here is an example of 3-out-of-5 secret sharing over $\mathbb{Z}_{11}$ (so $p = 11$). Suppose the secret being shared is $m = 7 \in \mathbb{Z}_{11}$. The Share *algorithm chooses a random degree-2 polynomial with constant coefficient 7. Let's say that the remaining two coefficients are chosen as $f_2 = 1$ and $f_1 = 4$, resulting in the following polynomial:*

$$f(x) = \boxed{1}\,x^2 + \boxed{4}\,x + 7$$

It is easy to visualize this polynomial over the real numbers; it is simply a parabola. But Shamir secret sharing asks us to consider this polynomial mod 11. What does that look like?

On the left is a plot of $f(x)$ over the real numbers. Working mod 11 means to "wrap around" every time the polynomial crosses over a multiple of 11 along the y-axis. This results in the blue plot below:



This is a picture of a mod-11 parabola. In fact, since we care only about $\mathbb{Z}_{11}$ inputs to f, you could rightfully say that just the 11 highlighted points alone (not the blue curve) are a picture of a mod-11 parabola.

For each user $i \in \{1, \ldots, 5\}$, we distribute the share $(i, f(i) \% 11)$. These shares correspond to the highlighted points in the mod-11 picture above.

user (i)	*f(i)*	*share (i, f(i) % 11)*
1	$f(1) = 12$	$(1,1)$
2	$f(2) = 19$	$(2,8)$
3	$f(3) = 28$	$(3,6)$
4	$f(4) = 39$	$(4,6)$
5	$f(5) = 52$	$(5,8)$

To show the scheme's security, we first show a convenient lemma about the distribution of shares in an unauthorized set:

Lemma 3.10 *Let p be a prime and define $\mathbb{Z}_p^+ \overset{\text{def}}{=} \mathbb{Z}_p \setminus \{0\}$. The following two libraries are interchangeable:*

```
$\mathcal{L}_{\text{ssss-real}}$
QUERY($m, t, U \subseteq \mathbb{Z}_p^+$):
  if $|U| \geqslant t$: return err
  $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
  $f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
  for $i \in U$:
    $s_i := (i, f(i) \% p)$
  return $\{s_i \mid i \in U\}$
```

```
$\mathcal{L}_{\text{ssss-rand}}$
QUERY($m, t, U \subseteq \mathbb{Z}_p^+$):
  if $|U| \geqslant t$: return err
  for $i \in U$:
    $y_i \leftarrow \mathbb{Z}_p$
    $s_i := (i, y_i)$
  return $\{s_i \mid i \in U\}$
```

In other words, if we evaluate a uniformly chosen degree $t-1$ polynomial on fewer than t points, the results are (jointly) uniformly distributed.

Proof We will prove the lemma here for the special case where the calling program always provides a set U with $|U| = t - 1$. Exercise 3.5 deals with the more general case.

Fix a message $m \in \mathbb{Z}_p$, fix set U of users with $|U| = t - 1$, and for each $i \in U$ fix a value $y_i \in \mathbb{Z}_p$. We wish to consider the probability that a call to QUERY(m, t, U) outputs $\{(i, y_i) \mid i \in U\}$, in each of the two libraries.

In library $\mathcal{L}_{\text{ssss-rand}}$, this event happens with probability $1/p^{t-1}$ since QUERY chooses the $t - 1$ different y_i values uniformly in $\mathbb{Z}_p$.

In library $\mathcal{L}_{\text{ssss-real}}$, the event happens if and only if the degree-$(t-1)$ polynomial $f(x)$ chosen by QUERY happens to pass through the set of points $\mathcal{P} = \{(i, y_i) \mid i \in U\} \cup \{(0, m)\}$. These are t points with distinct x-coordinates, so by Theorem 3.8 there is a *unique* degree-$(t-1)$ polynomial f with coefficients in $\mathbb{Z}_p$ passing through these points.

The QUERY subroutine picks f uniformly from the set of degree-$(t-1)$ polynomials satisfying $f(0) \equiv_p m$, of which there are p^{t-1}. Exactly one such polynomial causes the event in question, so the probability of the event is $1/p^{t-1}$.

Since the two libraries assign the same probability to all outcomes, we have $\mathcal{L}_{\text{ssss-real}} \equiv \mathcal{L}_{\text{ssss-rand}}$. ■

Theorem 3.11 *Shamir's secret-sharing scheme (Construction 3.9) is secure according to Definition 3.3.*

Proof Let $\mathcal{S}$ denote the Shamir secret-sharing scheme. We prove that $\mathcal{L}^{\mathcal{S}}_{\text{tsss-L}} \equiv \mathcal{L}^{\mathcal{S}}_{\text{tsss-R}}$ via a hybrid argument.

$\mathcal{L}^{\mathcal{S}}_{\text{tsss-L}}$:

```
$\mathcal{L}^{\mathcal{S}}_{\text{tsss-L}}$
QUERY($m_L, m_R, U$):
  if $|U| \geqslant t$: return err
  $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
  $f(x) := m_L + \sum_{j=1}^{t-1} f_j x^j$
  for $i \in U$:
    $s_i := (i, f(i) \% p)$
  return $\{s_i \mid i \in U\}$
```

Our starting point is $\mathcal{L}^{\mathcal{S}}_{\text{tsss-L}}$, shown here with the details of Shamir secret-sharing filled in.

```
QUERY($m_L, m_R, U$):
  return QUERY'($m_L, t, U$)
```

$\diamond$

```
$\mathcal{L}_{\text{ssss-real}}$
QUERY'($m, t, U$):
  if $|U| \geqslant t$: return err
  $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
  $f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
  for $i \in U$:
    $s_i := (i, f(i) \% p)$
  return $\{s_i \mid i \in U\}$
```

Almost the entire body of the QUERY subroutine has been factored out in terms of the $\mathcal{L}_{\text{ssss-real}}$ library defined above. The only thing remaining is the “choice” of whether to share m_L or m_R. Restructuring the code in this way has no effect on the library’s behavior.

```
QUERY($m_L, m_R, U$):
  return QUERY'($m_L, t, U$)
```

$\diamond$

```
$\mathcal{L}_{\text{ssss-rand}}$
QUERY'($m, t, U$):
  if $|U| \geqslant t$: return err
  for $i \in U$:
    $y_i \leftarrow \mathbb{Z}_p$
    $s_i := (i, y_i)$
  return $\{s_i \mid i \in U\}$
```

By Lemma 3.10, we can replace $\mathcal{L}_{\text{ssss-real}}$ with $\mathcal{L}_{\text{ssss-rand}}$, having no effect on the library’s behavior.

```
QUERY($m_L, m_R, U$):
  return QUERY'($m_R, t, U$)
```

$\diamond$

```
$\mathcal{L}_{\text{ssss-rand}}$
QUERY'($m, t, U$):
  if $|U| \geqslant t$: return err
  for $i \in U$:
    $y_i \leftarrow \mathbb{Z}_p$
    $s_i := (i, y_i)$
  return $\{s_i \mid i \in U\}$
```

The argument to QUERY′ has been changed from m_L to m_R. This has no effect on the library’s behavior, since QUERY′ is actually ignoring its argument in these hybrids.

```
QUERY($m_L, m_R, U$):
  return QUERY'($m_R, t, U$)
```

$\diamond$

```
$\mathcal{L}_{\text{ssss-real}}$
QUERY'($m, t, U$):
  if $|U| \geqslant t$: return err
  $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
  $f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
  for $i \in U$:
    $s_i := (i, f(i) \% p)$
  return $\{s_i \mid i \in U\}$
```

Applying the same steps in reverse, we can replace $\mathcal{L}_{\text{ssss-rand}}$ with $\mathcal{L}_{\text{ssss-real}}$, having no effect on the library’s behavior.

$\mathcal{L}^{\mathcal{S}}_{\text{tsss-R}}$:

```
$\mathcal{L}^{\mathcal{S}}_{\text{tsss-R}}$

QUERY($m_L, m_R, U$):
  if $|U| \geqslant t$: return err
  $f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
  $f(x) := m_R + \sum_{j=1}^{t-1} f_j x^j$
  for $i \in U$:
    $s_i := (i, f(i) \% p)$
  return $\{s_i \mid i \in U\}$
```

A subroutine has been inlined, which has no effect on the library's behavior. The resulting library is $\mathcal{L}^{\mathcal{S}}_{\text{tsss-R}}$.

We showed that $\mathcal{L}^{\mathcal{S}}_{\text{tsss-L}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}^{\mathcal{S}}_{\text{tsss-R}}$, so Shamir's secret sharing scheme is secure. ■

★ 3.6 Visual Secret Sharing

Here is a fun variant of 2-out-of-2 secret-sharing called **visual secret sharing.** In this variant, both the secret and the shares are black-and-white images. We require the same security property as traditional secret-sharing — that is, a single share (image) by itself reveals no information about the secret (image). What makes visual secret sharing different is that we require the *reconstruction* procedure to be done visually.

More specifically, each share should be printed on transparent sheets. When the two shares are stacked on top of each other, the secret image is revealed visually. We will discuss a simple visual secret sharing scheme that is inspired by the following observations:

when ▚ is stacked on top of ▚, the result is ▚
when ▞ is stacked on top of ▞, the result is ▞
when ▚ is stacked on top of ▞, the result is █
when ▞ is stacked on top of ▚, the result is █

Importantly, when stacking shares on top of each other in the first two cases, the result is a 2×2 block that is half-black, half-white (let's call it "gray"); while in the other cases the result is completely black.

The idea is to process each pixel of the source image independently, and to encode each pixel as a 2×2 block of pixels in each of the shares. A white pixel should be shared in a way that the two shares stack to form a "gray" 2×2 block, while a black pixel is shared in a way that results in a black 2×2 block.

More formally:

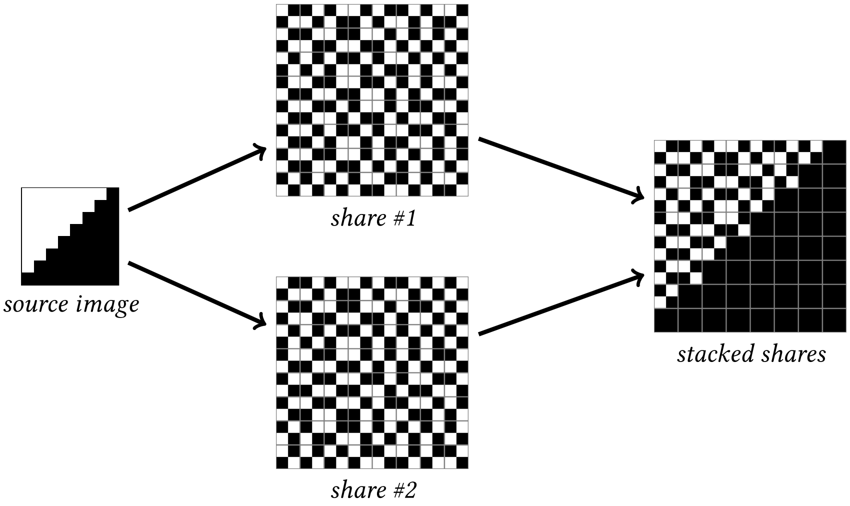
Construction 3.12

```
Share($m$):
  initialize empty images $s_1, s_2$, with dimensions twice that of $m$
  for each position $(i,j)$ in $m$:
    randomly choose $b_1 \leftarrow$ {▚,▞}
    if $m[i,j]$ is a white pixel: set $b_2 := b_1$
    if $m[i,j]$ is a black pixel: set $b_2$ to the "opposite" of $b_1$ (i.e., {▚,▞} $\setminus \{b_1\}$)
    add $2 \times 2$ block $b_1$ to image $s_1$ at position $(2i, 2j)$
    add $2 \times 2$ block $b_2$ to image $s_2$ at position $(2i, 2j)$
  return $(s_1, s_2)$
```

It is not hard to see that share s_1 leaks no information about the secret image m, because it consists of uniformly chosen 2×2 blocks. In the exercises you are asked to prove that s_2 also individually leaks nothing about the secret image.

Note that whenever the source pixel is white, the two shares have identical 2×2 blocks (so that when stacked, they make a "gray" block). Whenever a source pixel is black, the two shares have opposite blocks, so stack to make a black block.

Example



Exercises

3.1. Generalize Construction 3.5 to be an n-out-of-n secret-sharing scheme, and prove that your scheme is correct and secure.

3.2. Prove Theorem 3.7.

3.3. Fill in the details of the following alternative proof of Theorem 3.6: Starting with $\mathcal{L}_{\textsf{tsss-L}}$, apply the first step of the proof as before, to duplicate the main body into 3 branches of a new if-statement. Then apply Exercise 2.1 to the second branch of the if-statement. Argue that m_L can be replaced with m_R and complete the proof.

3.4. Suppose T is a fixed (publicly known) invertible $n \times n$ matrix over $\mathbb{Z}_p$, where p is a prime.

(a) Show that the following two libraries are interchangeable:

```
$\mathcal{L}_{\text{left}}$
QUERY():
    $\boldsymbol{r} \leftarrow (\mathbb{Z}_p)^n$
    return $\boldsymbol{r}$
```
;
```
$\mathcal{L}_{\text{right}}$
QUERY():
    $\boldsymbol{r} \leftarrow (\mathbb{Z}_p)^n$
    return $T \times \boldsymbol{r}$
```
.

(b) Show that the following two libraries are interchangeable:

```
$\mathcal{L}_{\text{left}}$
QUERY($\boldsymbol{v} \in (\mathbb{Z}_p)^n$):
    $\boldsymbol{r} \leftarrow (\mathbb{Z}_p)^n$
    $\boldsymbol{z} := \boldsymbol{v} + T\boldsymbol{r}$
    return $\boldsymbol{z}$
```
;
```
$\mathcal{L}_{\text{right}}$
QUERY($\boldsymbol{v} \in (\mathbb{Z}_p)^n$):
    $\boldsymbol{z} \leftarrow (\mathbb{Z}_p)^n$
    return $\boldsymbol{z}$
```
.

3.5. The text gives a proof of Lemma 3.10 for the special case where the calling program always calls QUERY with $|U| = t - 1$. This exercise shows one way to complete the proof. Define the following "wrapper" library:

```
$\mathcal{L}_{\text{wrap}}$
QUERY($m, t, U \subseteq \mathbb{Z}_p^+$):
    $U' := U$
    while $|U'| < t - 1$:
        add an arbitrary element of $\mathbb{Z}_p^+ \setminus U'$ to $U'$
    $s \leftarrow$ QUERY$'(m, t, U')$
    return $\{s_i \mid i \in U\}$
```
.

(a) Argue that $\mathcal{L}_{\text{sss-real}} \equiv \mathcal{L}_{\text{wrap}} \diamond \mathcal{L}'_{\text{sss-real}}$, where on the right-hand side $\mathcal{L}'_{\text{sss-real}}$ refers to $\mathcal{L}_{\text{sss-real}}$ but with its QUERY subroutine renamed to QUERY$'$.

(b) Argue that $\mathcal{L}_{\text{sss-rand}} \equiv \mathcal{L}_{\text{wrap}} \diamond \mathcal{L}'_{\text{sss-rand}}$, with the same interpretation as above.

(c) Argue that for any calling program $\mathcal{A}$, the combined program $\mathcal{A} \diamond \mathcal{L}_{\text{wrap}}$ only calls QUERY$'$ with $|U| = t - 1$. Hence, the proof presented in the text applies when the calling program has the form $\mathcal{A} \diamond \mathcal{L}_{\text{wrap}}$.

(d) Combining the previous parts, show that $\mathcal{L}_{\text{sss-real}} \equiv \mathcal{L}_{\text{sss-rand}}$ (i.e., the two libraries are interchangeable with respect to *arbitrary* calling programs).

3.6. Let $\mathcal{S}$ be a t-out-of-n threshold secret sharing scheme with $\mathcal{S}.\mathcal{M} = \{\texttt{0}, \texttt{1}\}^\ell$, and where each user's share is also a string of bits. Prove that if $\mathcal{S}$ satisfies security then every user's share must be at least ℓ bits long.

Hint: Prove the contrapositive. Suppose the first user's share is less than ℓ bits (and that this fact is known to everyone). Show how users 2 through t can violate security by enumerating all possibilities for the first user's share.

3.7. n users have shared two secrets using Shamir secret sharing. User i has a share $s_i = (i, y_i)$ of the secret m, and a share $s'_i = (i, y'_i)$ of the secret m'. Both sets of shares use the same prime modulus p.

Suppose each user i locally computes $z_i = (y_i + y'_i) \,\%\, p$.

(a) Prove that if the shares of m and shares of m' had the same threshold, then the resulting $\{(i, z_i) \mid i \leqslant n\}$ are a valid secret-sharing of the secret $m + m'$.

(b) Describe what the users get when the shares of m and m' had different thresholds (say, t and t', respectively).

3.8. Suppose there are 5 people on a committee: Alice (president), Bob, Charlie, David, Eve. Suggest how they can securely share a secret so that it can only be opened by:

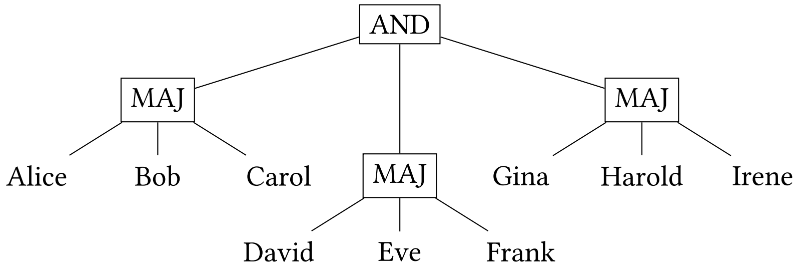
- ► Alice and any one other person
- ► Any three people

Describe in detail how the sharing algorithm works and how the reconstruction works (for all authorized sets of users).

3.9. Suppose there are 9 people on an important committee: Alice, Bob, Carol, David, Eve, Frank, Gina, Harold, & Irene. Alice, Bob & Carol form a subcommittee; David, Eve & Frank form another subcommittee; and Gina, Harold & Irene form another subcommittee.

Suggest how a dealer can share a secret so that it can only be opened when a majority of each subcommittee is present. Describe why a 6-out-of-9 threshold secret-sharing scheme does **not** suffice.

Hint:



⋆ 3.10. Generalize the previous exercise. A **monotone formula** is a boolean function $\phi : \{0,1\}^n \to \{0,1\}$ that when written as a formula uses only AND and OR operations (no NOTs). For a set $A \subseteq \{1, \ldots, n\}$, let χ_A be the bitstring where whose ith bit is 1 if and only if $i \in A$.

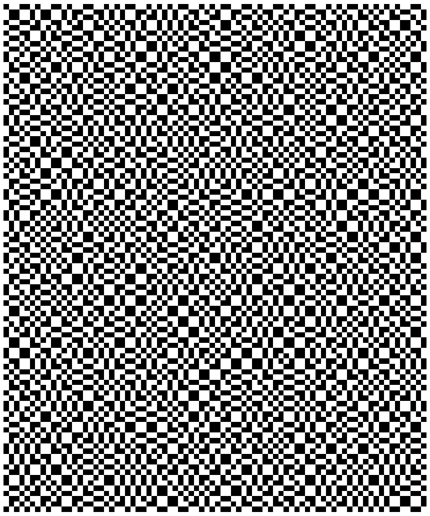
For every monotone formula $\phi : \{0,1\}^n \to \{0,1\}$, construct a secret-sharing scheme whose authorized sets are $\{A \subseteq \{1, \ldots, n\} \mid \phi(\chi_A) = 1\}$. Prove that your scheme is secure.

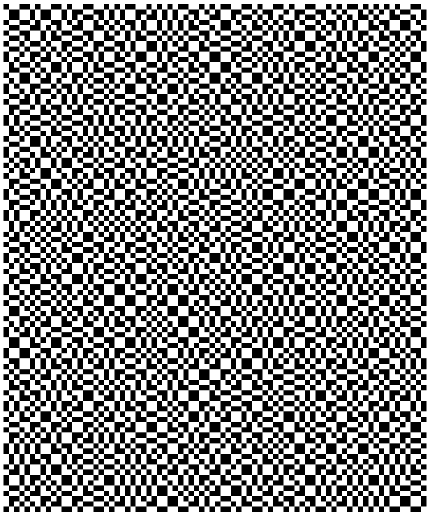
Hint: express the formula as a tree of AND and OR gates.

3.11. Prove that share s_2 in Construction 3.12 is distributed independently of the secret m.

⋆ 3.12. Construct a 3-out-of-3 visual secret sharing scheme. Any two shares should together reveal nothing about the source image, but any three reveal the source image when stacked together.

3.13. Using actual transparencies or with an image editing program, reconstruct the secret shared in these two images:





4 Basing Cryptography on Limits of Computation

John Nash was a mathematician who earned the 1994 Nobel Prize in Economics for his work in game theory. His life story was made into a successful movie, *A Beautiful Mind.*

In 1955, Nash was in correspondence with the United States National Security Agency (NSA),[1] discussing new methods of encryption that he had devised. In these letters, he also proposes some general principles of cryptography (bold highlighting not in the original):

> *We see immediately that in principle the enemy needs very little information to begin to break down the process. Essentially, as soon as r bits*[2] *of enciphered message have been transmitted the key is about determined. This is no security, for a practical key should not be too long.* ***But this does not consider how easy or difficult it is for the enemy to make the computation determining the key. If this computation, although possible in principle, were sufficiently long at best then the process could still be secure in a practical sense.***
>
> *The most direct computation procedure would be for the enemy to try all 2^r possible keys, one by one. Obviously this is easily made impractical for the enemy by simply choosing r large enough.*
>
> *In many cruder types of enciphering, particulary those which are not autocoding, such as substitution ciphers [letter for letter, letter pair for letter pair, triple for triple..] shorter means for computing the key are feasible, essentially because the key can be determined piece meal, one substitution at a time.*
>
> *So* ***a logical way to classify enciphering processes is by the way in which the computation length for the computation of the key increases with increasing length of the key. This is at best exponential*** *and at worst probably a relatively small power of r, ar^2 or ar^3, as in substitution ciphers.*
>
> *Now my general conjecture is as follows: For almost all sufficiently complex types of enciphering, especially where the instructions given by different portions of the key interact complexly with each other in the determination of their ultimate effects on the enciphering, the mean key computation length increases exponentially with the length of the key, or in other words, with the information content of the key.*
>
> *The significance of this general conjecture, assuming its truth, is easy to see. It means that* ***it is quite feasible to design ciphers that are effectively un-***

[1] The original letters, handwritten by Nash, are available at: http://www.nsa.gov/public_info/press_room/2012/nash_exhibit.shtml.

[2] Nash is using r to denote the length of the key, in bits. We would use λ.

> ***breakable.*** *As ciphers become more sophisticated the game of cipher breaking by skilled teams, etc. should become a thing of the past.*

Nash's letters were declassified only in 2012, so they did not directly influence the development of modern cryptography. Still, his letters illustrate the most important idea of "modern" cryptography: that **security can be based on the *computational difficulty* of an attack rather than on the *impossibility* of an attack**. In other words, we are willing to accept that breaking a cryptographic scheme may be possible *in principle*, as long as breaking it would require too much computational effort to be feasible.

We have already discussed one-time-secret encryption and secret sharing. For both of these tasks we were able to achieve a level of security guaranteeing that attacks are *impossible in principle*. But that's essentially the limit of what can be accomplished with such ideal security. Everything else we will see in this class, and every well-known product of cryptography (public-key encryption, hash functions, etc.) has a "modern"-style level of security, which guarantees that attacks are merely *computationally infeasible*, not *impossible*.

4.1 Polynomial-Time Computation

Nash's letters also spell out the importance of distinguishing between computations that take a polynomial amount of time and those that take an exponential amount of time.[3] Throughout computer science, polynomial-time is used as a formal definition of "efficient," and exponential-time (and above) is synonymous with "intractible."

In cryptography, it makes a lot of sense to not worry about guaranteeing security in the presence of attackers with unlimited computational resources. Not even the most powerful nation-states can invest 2^{256} CPU cycles towards a cryptographic attack.[4] So the modern approach to cryptography (more or less) follows Nash's suggestion, demanding that breaking a scheme requires exponential time.

Definition 4.1 *A program runs in **polynomial time** if there exists a constant $c > 0$ such that for all sufficiently long input strings x, the program stops after no more than $O(|x|^c)$ steps.*

Polynomial time is not a perfect match to what we mean by "efficient." Polynomial time includes algorithms with running time $\Theta(n^{1000})$, while excluding those with running time $\Theta(n^{\log\log\log n})$. Despite that, it's extremely useful because of the following *closure property*: repeating a polynomial-time process a polynomial number of times results in a polynomial-time process overall.

[3] Nash was surprisingly ahead of his time here. Polynomial-time wasn't formally proposed as a natural definition for "efficiency" in computation until Alan Cobham, 10 years after Nash's letter was written (Alan Cobham, *The intrinsic computational difficulty of functions*, in *Proc. Logic, Methodology, and Philosophy of Science II*, 1965). Until Nash's letters were declassified, the earliest well-known argument hinting at the importance of polynomial-time was in a letter from Kurt Gödel to John von Neumann. But that letter is not nearly as explicit as Nash's, and was anyway written a year later.

[4] Consider a *Hitchhiker's Guide to the Universe* scenario, in which the entire planet Earth is a supercomputer. Suppose you have a 10GHz computer (2^{33} cycles/sec) for every *atom* on earth (2^{166}). Running them all for the age of the earth (2^{57} seconds) gives you a single 2^{256} computation.

Security Parameter

The definition of polynomial-time is *asymptotic*, since it considers the behavior of a computer program *as the size of the inputs grows to infinity.* Cryptographic algorithms often take multiple different inputs to serve various purposes, so to be absolutely clear about our "measuring stick" for polynomial time, we measure the efficiency of cryptographic algorithms (and adversaries!) against something called the **security parameter**, which is the number of bits needed to represent secret keys and/or randomly chosen values used in the scheme. We will typically use λ to refer to the security parameter of a scheme.

It's helpful to think of the security parameter as a tuning knob for the cryptographic system. When we dial up this knob, we increase the size of the keys in the system, the size of all associated things like ciphertexts, and the required computation time of the associated algorithms. Most importantly, the amount of effort required by the honest users grows reasonably (as a polynomial function of the security parameter) while the effort required to violate security increases faster than any (polynomial-time) adversary can keep up.

Potential Pitfall: Numerical Algorithms

The public-key cryptographic algorithms that we will see are based on problems from abstract algebra and number theory. These schemes require users to perform operations on very large numbers. We must remember that representing the number N on a computer requires only $\lceil \log_2(N+1) \rceil$ bits. This means that $\lceil \log_2(N+1) \rceil$, rather than N, is our security parameter! We will therefore be interested in whether certain operations on the number N run in polynomial-time as a function of $\lceil \log_2(N+1) \rceil$, rather than in N. Keep in mind that the difference between running time $O(\log N)$ and $O(N)$ is the difference between writing down a number and counting to the number.

For reference, here are some numerical operations that we will be using later in the class, and their known efficiencies:

Efficient algorithm known:	**No known efficient algorithm:**
Computing GCDs	Factoring integers
Arithmetic mod N	Computing $\phi(N)$ given N
Inverses mod N	Discrete logarithm
Exponentiation mod N	Square roots mod composite N

By "efficient," we mean polynomial-time. However, all of the problems in the right-hand column *do* have known polynomial-time algorithms on quantum computers.

4.2 Negligible Probabilities

In all of the cryptography that we'll see, an adversary can *always* violate security simply by guessing some value that was chosen at random, like a secret key. However, imagine a system that has 1024-bit secret keys chosen uniformly at random. The probability of correctly guessing the key is 2^{-1024}, which is so low that we can safely ignore it.

We don't worry about "attacks" that have such a ridiculously small success probability. But we would worry about an attack that succeeds with, say, probability 1/2. Somewhere between 2^{-1024} and 2^{-1} we need to find a sensible place to draw the line. In the same way that polynomial time formalizes "efficient" running times, we will use an *asymptotic* definition of what is a negligibly small probability.

Consider a scheme with keys that are λ bits long. Then a blind-guessing attack may succeed with probability $1/2^\lambda$. Now what about an adversary who makes 2 blind guesses, or λ guesses, or λ^{42} guesses? Such an adversary would still run in polynomial time, and might succeed in its attack with probability $2/2^\lambda$, $\lambda/2^\lambda$, or $\lambda^{42}/2^\lambda$. The important thing is that, no matter what polynomial you put on top, the probability still goes to zero. Indeed, $1/2^\lambda$ **goes to zero so fast that no polynomial can "rescue" it.** This suggests our formal definition:

Definition 4.2 (Negligible) *A function f is **negligible** if, for every polynomial p, we have $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.*

In other words, a negligible function approaches zero so fast that you can never catch up when multiplying by a polynomial. This is exactly the property we want from a security guarantee that is supposed to hold against all polynomial-time adversaries. If a polynomial-time adversary succeeds with probability f, then running the same attack p times would still be an overall polynomial-time attack (if p is a polynomial), and potentially have success probability $p \cdot f$.

When you want to check whether a function is negligible, you only have to consider polynomials p of the form $p(\lambda) = \lambda^c$ for some constant c:

Claim 4.3 *If for every integer c, $\lim_{\lambda\to\infty} \lambda^c f(\lambda) = 0$, then f is negligible.*

Proof Suppose f has this property, and take an arbitrary polynomial p. We want to show that $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.

If d is the degree of p, then $\lim_{\lambda\to\infty} \frac{p(\lambda)}{\lambda^{d+1}} = 0$. Therefore,

$$\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = \lim_{\lambda\to\infty}\left[\frac{p(\lambda)}{\lambda^{d+1}}\left(\lambda^{d+1}\cdot f(\lambda)\right)\right] = \left(\lim_{\lambda\to\infty}\frac{p(\lambda)}{\lambda^{d+1}}\right)\left(\lim_{\lambda\to\infty}\lambda^{d+1}\cdot f(\lambda)\right) = 0\cdot 0.$$

The second equality is a valid law for limits since the two limits on the right exist and are not an indeterminate expression like $0 \cdot \infty$. The final equality follows from the hypothesis on f. ■

Example *The function $f(\lambda) = 1/2^\lambda$ is negligible, since for any integer c, we have:*

$$\lim_{\lambda\to\infty} \lambda^c/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)}/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)-\lambda} = 0,$$

since $c\log(\lambda) - \lambda$ approaches $-\infty$ in the limit, for any constant c. Using similar reasoning, one can show that the following functions are also negligible:

$$\frac{1}{2^{\lambda/2}}, \qquad \frac{1}{2^{\sqrt{\lambda}}}, \qquad \frac{1}{2^{\log^2\lambda}}, \qquad \frac{1}{\lambda^{\log\lambda}}.$$

Functions like $1/\lambda^5$ approach zero but not fast enough to be negligible. To see why, we can take polynomial $p(\lambda) = \lambda^6$ and see that the resulting limit does not satisfy the requirement from Definition 4.2:

$$\lim_{\lambda\to\infty} p(\lambda)\frac{1}{\lambda^5} = \lim_{\lambda\to\infty} \lambda = \infty \neq 0$$

In this class, when we see a negligible function, it will typically always be one that is easy to recognize as negligible (just as in an undergraduate algorithms course, you won't really encounter algorithms where it's hard to tell whether the running time is polynomial).

Definition 4.4 ($f \approx g$) *If $f, g : \mathbb{N} \to \mathbb{R}$ are two functions, we write $f \approx g$ to mean that $\big|f(\lambda) - g(\lambda)\big|$ is a negligible function.*

We use the terminology of negligible functions exclusively when discussing probabilities, so the following are common:

$$\begin{aligned} \Pr[X] \approx 0 &\iff \text{"event } X \text{ almost never happens"} \\ \Pr[Y] \approx 1 &\iff \text{"event } Y \text{ almost always happens"} \\ \Pr[A] \approx \Pr[B] &\iff \text{"events } A \text{ and } B \text{ happen with} \\ &\qquad \text{essentially the same probability"}^{5} \end{aligned}$$

Additionally, the $\approx$ symbol is *transitive*:[6] if $\Pr[X] \approx \Pr[Y]$ and $\Pr[Y] \approx \Pr[Z]$, then $\Pr[X] \approx \Pr[Z]$ (perhaps with a slightly larger, but still negligible, difference).

4.3 Indistinguishability

So far we have been writing formal security definitions in terms of interchangeable libraries, which requires that two libraries have *exactly the same* effect on *every* calling program. Going forward, our security definitions will not be quite as demanding. First, we only consider polynomial-time calling programs; second, we don't require the libraries to have exactly the same effect on the calling program, only that the difference in effects is negligible.

Definition 4.5 (Indistinguishable) *Let $\mathcal{L}_{\textsf{left}}$ and $\mathcal{L}_{\textsf{right}}$ be two libraries with a common interface. We say that $\mathcal{L}_{\textsf{left}}$ and $\mathcal{L}_{\textsf{right}}$ are **indistinguishable**, and write $\mathcal{L}_{\textsf{left}} \approxeq \mathcal{L}_{\textsf{right}}$, if for all polynomial-time programs $\mathcal{A}$ that output a single bit, $\Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{left}} \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{right}} \Rightarrow 1]$.*

*We call the quantity $\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{right}} \Rightarrow 1]\Big|$ the **advantage** or **bias** of $\mathcal{A}$ in distinguishing $\mathcal{L}_{\textsf{left}}$ from $\mathcal{L}_{\textsf{right}}$. Two libraries are therefore indistinguishable if all polynomial-time calling programs have negligible advantage in distinguishing them.*

From the properties of the "$\approx$" symbol, we can see that indistinguishability of libraries is also transitive, which allows us to carry out hybrid proofs of security in the same way as before.

Analogous to Lemma 2.7, we also have the following library chaining lemma, which you are asked to prove as an exercise:

[5] $\Pr[A] \approx \Pr[B]$ doesn't mean that events A and B almost always happen **together** (when A and B are defined over a common probability space) — imagine A being the event "the coin came up heads" and B being the event "the coin came up tails." These events have the same probability but never happen together. To say that "A and B almost always happen together," you'd have to say something like $\Pr[A \oplus B] \approx 0$, where $A \oplus B$ denotes the event that *exactly one* of A and B happens.

[6] It's only transitive when applied a polynomial number of times. So you can't define a whole series of events X_i, show that $\Pr[X_i] \approx \Pr[X_{i+1}]$, and conclude that $\Pr[X_1] \approx \Pr[X_{2^n}]$. It's rare that we'll encounter this subtlety in this course.

Lemma 4.6 (Chaining) *If $\mathcal{L}_{\textsf{left}} \approxeq \mathcal{L}_{\textsf{right}}$ then $\mathcal{L}^* \diamond \mathcal{L}_{\textsf{left}} \approxeq \mathcal{L}^* \diamond \mathcal{L}_{\textsf{right}}$ for any polynomial-time library $\mathcal{L}^*$.*

Bad-Event Lemma

A common situation is when two libraries carry out exactly the same steps until some exceptional condition happens. In that case, we can bound an adversary's distinguishing advantage by the probability of the exceptional condition.

More formally, we can state a lemma of Bellare & Rogaway.[7] We present it without proof, since it involves a *syntactic* requirement. Proving it formally would require a formal definition of the syntax/semantics of the pseudocode that we use for these libraries.

Lemma 4.7 (Bad events) *Let $\mathcal{L}_{\textsf{left}}$ and $\mathcal{L}_{\textsf{right}}$ be libraries that each define a variable named 'bad' that is initialized to 0. If $\mathcal{L}_{\textsf{left}}$ and $\mathcal{L}_{\textsf{right}}$ have identical code, except for code blocks reachable only when bad $= 1$ (*e.g.*, guarded by an "if bad $= 1$" statement), then*

$$\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{left}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{right}} \Rightarrow 1]\Big| \leqslant \Pr[\mathcal{A} \diamond \mathcal{L}_{\textsf{left}} \textit{ sets bad} = 1].$$

4.4 Birthday Probabilities & Sampling With[out] Replacement

In many cryptographic schemes, the users repeatedly choose random strings (*e.g.*, each time they encrypt a message), and security breaks down if the same string is ever chosen twice. Hence, it is important that the probability of a repeated sample is *negligible*. In this section we compute the probability of such events and express our findings in a modular way, as a statement about the indistinguishability of two libraries.

Birthday Probabilities

If q people are in a room (and we assume that each person's birthday is uniformly chosen from among the possible days in a year), what is the probability that there are two people in the room with the same birthday? This question is known as the **birthday problem**, and it is famous because the answer is highly unintuitive to most people.[8]

Let's make the question more general. Imagine taking q independent, uniform samples from a set of N items. What is the probability that the same value gets chosen more than once? In other words, what is the probability that the following program outputs 1?

```
$\mathcal{B}$
for $i := 1$ to $q$:
  $s_i \leftarrow \{1, \ldots, N\}$
  for $j := 1$ to $i - 1$:
     if $s_i = s_j$ then return 1
return 0
```

[7] Mihir Bellare & Phillip Rogaway: "Code-Based Game-Playing Proofs and the Security of Triple Encryption," in Eurocrypt 2006. ia.cr/2004/331

[8] It is sometimes called the "birthday paradox," even though it is not really a paradox. The *actual* birthday paradox is that the "birthday paradox" is not a paradox.

Let's give a name to this probability:

$$\text{BirthdayProb}(q,N) \overset{\text{def}}{=} \Pr[\mathcal{B} \text{ outputs } 1].$$

It is possible to write an exact formula for this probability:

Lemma 4.8 $\textit{BirthdayProb}(q,N) = 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right).$

Proof Let us instead compute the probability that $\mathcal{B}$ outputs 0, which will allow us to then solve for the probability that it outputs 1. In order for $\mathcal{B}$ to output 0, it must avoid the early termination conditions in each iteration of the main loop. Therefore:

$$\begin{aligned}\Pr[\mathcal{B} \text{ outputs } 0] &= \Pr[\mathcal{B} \text{ doesn't terminate early in iteration } i = 1]\\ &\quad\cdot \Pr[\mathcal{B} \text{ doesn't terminate early in iteration } i = 2]\\ &\quad\vdots\\ &\quad\cdot \Pr[\mathcal{B} \text{ doesn't terminate early in iteration } i = q]\end{aligned}$$

In iteration i of the main loop, there are $i - 1$ previously chosen values $s_1, \ldots, s_{i-1}$. The program terminates early if any of these are chosen again as s_i, otherwise it continues to the next iteration. Since there are N choices for s_i, each with equal probability:

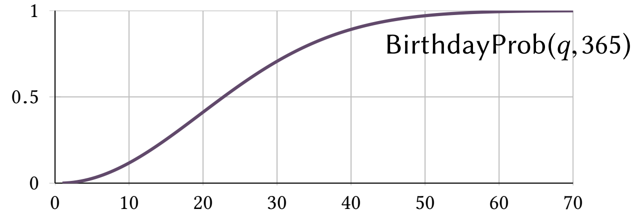
$$\Pr[\mathcal{B} \text{ doesn't terminate early in iteration } i] = 1 - \frac{i-1}{N}.$$

Putting everything together:

$$\begin{aligned}\text{BirthdayProb}(q,N) &= \Pr[\mathcal{B} \text{ outputs } 1]\\ &= 1 - \Pr[\mathcal{B} \text{ outputs } 0]\\ &= 1 - \left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{q-1}{N}\right)\\ &= 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right)\end{aligned}$$

This completes the proof. ■

This formula for BirthdayProb(q,N) is not easy to understand at a glance. We can get a better sense of its behavior as a function of q by plotting it. Below is a plot with $N = 365$, corresponding to the classic birthday problem:



With only $q = 23$ people the probability of a shared birthday already exceeds 50%. The graph could be extended to the right (all the way to $q = 365$), but even at $q = 70$ the probability exceeds 99.9%.

Asymptotic Bounds on the Birthday Probability

It will be helpful to have an *asymptotic* formula for how $\text{BirthdayProb}(q,N)$ grows as a function of q and N. We are most interested in the case where q is relatively small compared to N (*e.g.*, when q is a polynomial function of λ but N is exponential).

Lemma 4.9 (Birthday Bound) *If $q \leqslant \sqrt{2N}$, then*

$$0.632\frac{q(q-1)}{2N} \leqslant \mathit{BirthdayProb}(q,N) \leqslant \frac{q(q-1)}{2N}.$$

Since the upper and lower bounds differ by only a constant factor, it makes sense to write $\mathit{BirthdayProb}(q,N) = \Theta(q^2/N)$.

Proof We split the proof into two parts.

- To prove the upper bound, we use the fact that when x and y are positive,

$$\begin{aligned}(1-x)(1-y) &= 1-(x+y)+xy\\ &\geqslant 1-(x+y).\end{aligned}$$

More generally, when all terms x_i are positive, $\prod_i(1-x_i) \geqslant 1-\sum_i x_i$. Hence,

$$1-\prod_i(1-x_i) \leqslant 1-(1-\textstyle\sum_i x_i) = \sum_i x_i.$$

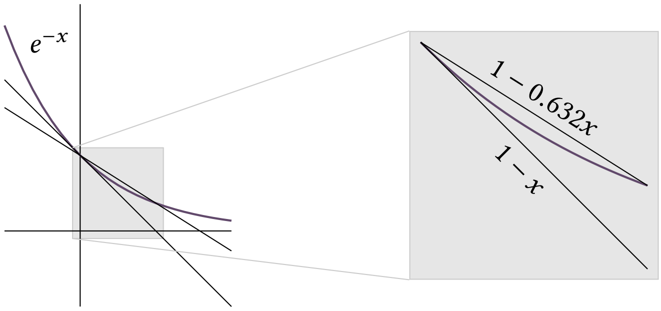
Applying that fact,

$$\text{BirthdayProb}(q,N) \overset{\text{def}}{=} 1-\prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right) \leqslant \sum_{i=1}^{q-1}\frac{i}{N} = \frac{\sum_{i=1}^{q-1} i}{N} = \frac{q(q-1)}{2N}.$$

- To prove the lower bound, we use the fact that when $0 \leqslant x \leqslant 1$,

$$1-x \leqslant e^{-x} \leqslant 1-0.632x.$$

This fact is illustrated below. The significance of 0.632 is that $1-\frac{1}{e} = 0.63212\ldots$



We can use both of these upper and lower bounds on e^{-x} to show the following:

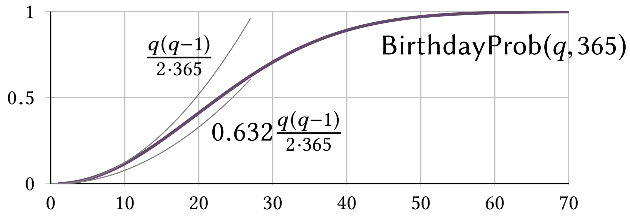
$$\prod_{i=1}^{q-1}\left(1-\frac{i}{N}\right) \leqslant \prod_{i=1}^{q-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{q-1}\frac{i}{N}} = e^{-\frac{q(q-1)}{2N}} \leqslant 1-0.632\frac{q(q-1)}{2N}.$$

With the last inequality we used the fact that $q \leqslant \sqrt{2N}$, and therefore $\frac{q(q-1)}{2N} \leqslant 1$ (this is necessary to apply the inequality $e^{-x} \leqslant 1 - 0.632x$). Hence:

$$\begin{aligned} \text{BirthdayProb}(q,N) &\overset{\text{def}}{=} 1 - \prod_{i=1}^{q-1} \left(1 - \frac{i}{N}\right) \\ &\geqslant 1 - \left(1 - 0.632\frac{q(q-1)}{2N}\right) = 0.632\frac{q(q-1)}{2N}. \end{aligned}$$

This completes the proof. ■

Below is a plot of these bounds compared to the actual value of BirthdayProb(q,N) (for $N = 365$):



As mentioned previously, BirthdayProb(q,N) grows roughly like q^2/N within the range of values we care about (q small relative to N).

The Birthday Problem in Terms of Indistinguishable Libraries

Below are two libraries which will also be useful for future topics.

```
$\mathcal{L}_{\text{samp-L}}$

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$
```

```
$\mathcal{L}_{\text{samp-R}}$

$R := \emptyset$

SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

Both libraries provide a SAMP subroutine that samples a random element of $\{0,1\}^\lambda$. The implementation in $\mathcal{L}_{\text{samp-L}}$ samples uniformly and independently from $\{0,1\}^\lambda$ each time. It samples **with replacement,** so it is possible (although maybe unlikely) for multiple calls to SAMP to return the same value in $\mathcal{L}_{\text{samp-L}}$.

On the other hand, $\mathcal{L}_{\text{samp-R}}$ samples λ-bit strings **without replacement**. It keeps track of a set R, containing all the values it has previously sampled, and avoids choosing them again ("$\{0,1\}^\lambda \setminus R$" is the set of λ-bit strings excluding the ones in R).[9] In this library, SAMP will never output the same value twice.

[9] In our convention, when a variable like R is initialized outside of any subroutine, it means that the variable is *static* (its value is maintained across different subroutine calls) and *private* (its value is not directly accessible to the calling program).

The "obvious" distinguishing strategy. A natural way (but maybe not the *only* way) to distinguish these two libraries, therefore, would be to call SAMP many times. If you ever see a repeated output, then you must certainly be linked to $\mathcal{L}_{\text{samp-L}}$. After some number of calls to SAMP, if you still don't see any repeated outputs, you might eventually stop and guess that you are linked to $\mathcal{L}_{\text{samp-R}}$.

Let $\mathcal{A}_q$ denote this "obvious" calling program that makes q calls to SAMP and returns 1 if it sees a repeated value. Clearly, the program can never return 1 when it is linked to $\mathcal{L}_{\text{samp-R}}$. On the other hand, when it is linked to $\mathcal{L}_{\text{samp-L}}$, it returns 1 with probability exactly $\text{BirthdayProb}(q, 2^\lambda)$. Therefore, the *advantage* of $\mathcal{A}_q$ is exactly $\text{BirthdayProb}(q, 2^\lambda)$.

This program behaves differently in the presence of these two libraries, therefore they are not *interchangeable.* But are the libraries *indistinguishable?* We have demonstrated a calling program with advantage $\text{BirthdayProb}(q, 2^\lambda)$. We have not specified q exactly, but if $\mathcal{A}_q$ is mean to run in polynomial time (as a function of λ), then q must be a polynomial function of λ. Then the advantage of $\mathcal{A}_q$ is $\text{BirthdayProb}(q, 2^\lambda) = \Theta(q^2/2^\lambda)$, which is *negligible!*

To show that the librares are indistinguishable, we have to show that *all* calling programs have negligible advantage. It is not enough just to show that this *particular* calling program has negligible advantage. Perhaps surprisingly, the "obvious" calling program that we considered is the *best possible* distinguisher!

Lemma 4.10 (Repl. Sampling) *Let $\mathcal{L}_{\text{samp-L}}$ and $\mathcal{L}_{\text{samp-R}}$ be defined as above. Then for all calling programs $\mathcal{A}$ that make q queries to the* SAMP *subroutine, the advantage of $\mathcal{A}$ in distinguishing the libraries is* ***at most*** *$BirthdayProb(q, 2^\lambda)$.*

In particular, when $\mathcal{A}$ is polynomial-time (in λ), then q grows as a polynomial in the security parameter. Hence, $\mathcal{A}$ has negligible advantage. Since this is true for all polynomial-time $\mathcal{A}$, we have $\mathcal{L}_{\text{samp-L}} \approx \mathcal{L}_{\text{samp-R}}$.

Proof Consider the following hybrid libraries:

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

$R := \emptyset$
bad := 0

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  if $r \in R$ then:
    bad := 1

  $R := R \cup \{r\}$
  return $r$
```

```
$\mathcal{L}_{\text{hyb-R}}$

$R := \emptyset$
bad := 0

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  if $r \in R$ then:
    bad := 1
    $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

First, let us prove some simple observations about these libraries:

$\mathcal{L}_{\text{hyb-L}} \equiv \mathcal{L}_{\text{samp-L}}$: Note that $\mathcal{L}_{\text{hyb-L}}$ simply samples uniformly from $\{0,1\}^\lambda$. The extra R and bad variables in $\mathcal{L}_{\text{hyb-L}}$ don't actually have an effect on its external behavior (they are used only for convenience later in the proof).

$\mathcal{L}_{\text{hyb-R}} \equiv \mathcal{L}_{\text{samp-R}}$: Whereas $\mathcal{L}_{\text{samp-R}}$ avoids repeats by simply sampling from $\{0,1\}^\lambda \setminus R$, this library $\mathcal{L}_{\text{hyb-R}}$ samples r uniformly from $\{0,1\}^\lambda$ and retries if the result happens to be in R. This method is called *rejection sampling*, and it has the same effect[10] as sampling r directly from $\{0,1\}^\lambda \setminus R$.

Conveniently, $\mathcal{L}_{\text{hyb-L}}$ and $\mathcal{L}_{\text{hyb-R}}$ differ only in code that is reachable when bad = 1 (highlighted). So, using Lemma 4.7, we can bound the advantage of the calling program:

$$\begin{aligned}
&\Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{samp-R}} \Rightarrow 1]\Big| \\
&\qquad = \Big|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-R}} \Rightarrow 1]\Big| \\
&\qquad \leqslant \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}} \text{ sets bad} := 1].
\end{aligned}$$

Finally, we can observe that $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-L}}$ sets bad := 1 only in the event that it sees a repeated sample from $\{0,1\}^\lambda$. This happens with probability $\text{BirthdayProb}(q, 2^\lambda)$. ■

Discussion

- Cryptographic schemes are often designed so that users sample repeatedly from $\{0,1\}^\lambda$. Security often breaks down when the same value is sampled twice, and this happens in the range of $\sqrt{2^{\lambda+1}} \sim 2^{\lambda/2}$ steps. The birthday bounds can inform the choice of parameters used in real systems.

- Stating the birthday problem in terms of indistinguishable libraries makes it a useful tool in future security proofs. For example, when proving the security of a construction we can replace a uniform sampling step with a sampling-without-replacement step. This change has only a negligible effect, but now the rest of the proof can take advantage of the fact that samples are never repeated.

  Another way to say this is that, when you are thinking about a cryptographic construction, it is "safe to assume" that randomly sampled long strings do not repeat, and behave accordingly.

Exercises

4.1. Which of the following are negligible functions in λ? Justify your answers.

$$\frac{1}{2^{\lambda/2}} \quad \frac{1}{2^{\log(\lambda^2)}} \quad \frac{1}{\lambda^{\log(\lambda)}} \quad \frac{1}{\lambda^2} \quad \frac{1}{2^{(\log \lambda)^2}} \quad \frac{1}{(\log \lambda)^2} \quad \frac{1}{\lambda^{1/\lambda}} \quad \frac{1}{\sqrt{\lambda}} \quad \frac{1}{2^{\sqrt{\lambda}}}$$

4.2. Suppose f and g are negligible.

(a) Show that $f + g$ is negligible.

(b) Show that $f \cdot g$ is negligible.

(c) Give an example f and g which are both negligible, but where $f(\lambda)/g(\lambda)$ is not negligible.

[10] The two approaches for sampling from $\{0,1\}^\lambda \setminus R$ may have different running times, but our model considers only the input-output behavior of the library.

4.3. Show that when f is negligible, then for every polynomial p, the function $p(\lambda)f(\lambda)$ not only approaches 0, but it is also negligible itself.

Hint: use the contrapositive. Suppose that $p(\lambda)f(\lambda)$ is non-negligible, where p is a polynomial. Conclude that f must also be non-negligible.

4.4. Prove that the $\approx$ relation is transitive. Let $f,g,h : \mathbb{N} \to \mathbb{R}$ be functions. Using the definition of the $\approx$ relation, prove that if $f \approx g$ and $g \approx h$ then $f \approx h$. You may find it useful to invoke the *triangle inequality*: $|a - c| \leqslant |a - b| + |b - c|$.

4.5. Prove Lemma 4.6.

★ 4.6. A *deterministic* program is one that uses no random choices. Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two *deterministic* libraries with a common interface. Show that either $\mathcal{L}_1 \equiv \mathcal{L}_2$, or else $\mathcal{L}_1$ & $\mathcal{L}_2$ can be distinguished with advantage 1.

4.7. For this problem, consider the following two libraries:

```
$\mathcal{L}_{\text{left}}$

AVOID($v \in \{0,1\}^\lambda$):
    return null

SAMP():
    $r \leftarrow \{0,1\}^\lambda$
    return $r$
```

```
$\mathcal{L}_{\text{right}}$
$\mathcal{V} := \emptyset$

AVOID($v \in \{0,1\}^\lambda$):
    $\mathcal{V} := \mathcal{V} \cup \{v\}$
    return null

SAMP():
    $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{V}$
    return $r$
```

(a) Prove that the two libraries are indistinguishable. More precisely, show that if an adversary makes q_1 number of calls to AVOID and q_2 calls to SAMP, then its distinguishing advantage is at most $q_1 q_2 / 2^\lambda$. For a polynomial-time adversary, both q_1 and q_2 (and hence their product) are polynomial functions of the security parameter, so the advantage is negligible.

(b) Suppose an adversary makes a total of n_i calls to AVOID (with distinct arguments) before making its ith call to SAMP, and furthermore the adversary makes q calls to SAMP overall (so that $n_1 \leqslant n_2 \leqslant \cdots \leqslant n_q$). Show that the two libraries *can* be distinguished with advantage at least:

$$0.632 \cdot \frac{\sum_{i=1}^{q} n_i}{2^\lambda}$$

4.8. Prove the following generalization of the results in this chapter:

> Fix a value $x \in \{0,1\}^\lambda$. Then when taking q uniform samples from $\{0,1\}^\lambda$, the probability that there exist two distinct samples **whose** XOR **is** x is BirthdayProb$(q, 2^\lambda)$.

Hint: One way to prove this involves applying Exercise 4.7. Another way involves applying Claim 2.3 to the program $\mathcal{B}$ in Section 4.4.

4.9. Suppose you want to enforce password rules so that at least 2^{128} passwords satisfy the rules. How many characters long must the passwords be, in each of these cases?

(a) Passwords consist of lowercase `a` through `z` only.

(b) Passwords consist of lowercase and uppercase letters `a`–`z` and `A`–`Z`.

(c) Passwords consist of lower/uppercase letters and digits `0`–`9`.

(d) Passwords consist of lower/uppercase letters, digits, and any symbol characters that appear on a standard US keyboard (including the space character).

5 Pseudorandom Generators

We have already seen that randomness is essential for cryptographic security. Following Kerckhoff's principle, we assume that an adversary knows *everything* about our cryptographic algorithms except for the outcome of the internal random choices made when running the algorithms. Private randomness truly is the *only* resource honest users can leverage for security in the presence of an adversary.

One-time secrecy for encryption is a very demanding requirement, when we view randomness as a resource. One-time secrecy essentially requires one to use independent, uniformly random bits to mask every bit of plaintext that is sent. As a result, textbook one-time pad is usually grossly impractical for practical use.

In this chapter, we ask whether it might be beneficial to settle for a distribution of bits which is not uniform but merely "looks uniform enough." This is exactly the idea behind **pseudorandomness.** A pseudorandom distribution is not uniform in a strict mathematical sense, but is *indistinguishable* from the uniform distribution by polynomial-time algorithms (in the sense defined in the previous chapter).

Perhaps surprisingly, a relatively small amount of *truly uniform* bits can be used to generate a huge amount of *pseudorandom* bits. Pseudorandomness therefore leads to an entire new world of cryptographic possibilities, an exploration that we begin in this chapter.

5.1 Definition

As mentioned above, a pseudorandom distribution "looks uniform" to all polynomial-time computations. We already know of a distribution that "looks uniform" — namely, the uniform distribution itself! A more interesting case is when λ uniform bits are used to *deterministically* (*i.e.*, without further use of randomness) produce $\lambda + \ell$ pseudorandom bits, for $\ell > 0$. The process that "extends" λ bits into $\lambda + \ell$ bits is called a pseudorandom generator. More formally:

Definition 5.1 (PRG security)

*Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a deterministic function with $\ell > 0$. We say that G is a **secure pseudorandom generator (PRG)** if $\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}^G_{\text{prg-rand}}$, where:*

```
$\mathcal{L}^G_{\text{prg-real}}$

QUERY():
  $s \leftarrow \{0,1\}^\lambda$
  return $G(s)$
```

```
$\mathcal{L}^G_{\text{prg-rand}}$

QUERY():
  $z \leftarrow \{0,1\}^{\lambda+\ell}$
  return $z$
```

*The value ℓ is called the **stretch** of the PRG. The input to the PRG is typically called a **seed**.*

Discussion

- Is `0010110110` a random string? Is it pseudorandom? What about `0000000001`? Do these questions make any sense?

  Randomness and pseudorandomness are not properties of individual strings, they are properties of the *process* used to generate the string. We will try to be precise about how we talk about these things (and you should too). When we have a value $z = G(s)$ where G is a PRG and s is chosen uniformly, we can say that z was "chosen pseudorandomly", but not that z "is pseudorandom". On the other hand, a *distribution* can be described as "pseudorandom." The same goes for describing a value as "chosen uniformly" and describing a distribution as "uniform."

- Pseudorandomness can happen only in the computational setting, where we restrict focus to polynomial-time adversaries. The exercises ask you to prove that for all functions G (with positive stretch), $\mathcal{L}^G_{\text{prg-real}} \not\equiv \mathcal{L}^G_{\text{prg-rand}}$ (note the use of $\equiv$ rather than $\approx$). That is, the output distribution of the PRG can never actually *be* uniform in a mathematical sense. Because it has postive stretch, the best it can hope to be is pseudorandom.

- It's sometimes convenient to think in terms of statistical tests. When given access to some data claiming to be uniformly generated, your first instinct is probably to perform a set of basic *statistical tests*: Are there roughly an equal number of `0`s as `1`s? Does the substring `01010` occur with roughly the frequency I would expect? If I interpret the string as a series of points in the unit square $[0,1)^2$, is it true that roughly $\pi/4$ of them are within Euclidean distance 1 of the origin?[1]

  The definition of pseudorandomness is kind of a "master" definition that encompasses all of these statistical tests and more. After all, what is a statistical test, but a polynomial-time procedure that obtains samples from a distribution and outputs a yes-or-no decision? Pseudorandomness implies that *every* statistical test will "accept" when given pseudorandomly generated inputs with essentially the same probability as when given uniformly sampled inputs.

- Consider the case of a length-doubling PRG (so $\ell = \lambda$; the PRG has input length λ and output length 2λ). The PRG only has 2^λ possible inputs, and so there are at most only 2^λ possible outputs. Among all of $\{0,1\}^{2\lambda}$, this is a miniscule fraction indeed. Almost all strings of length 2λ are *impossible outputs* of G. So how can outputs of G possibly "look uniform?"

  The answer is that it's not clear how to take advantage of this observation. While it is true that most strings (in a *relative* sense as a fraction of $2^{2\lambda}$) are impossible outputs of G, it is also true that 2^λ of them are possible, which is certainly a lot from the perspective of a program who runs in polynomial time in λ. Recall that the problem at hand is designing a distinguisher to behave as differently as possible in the presence of pseudorandom and uniform distributions. It is not enough to behave differently on just a few strings here and there — an individual string can contribute

[1] For a list of statistical tests of randomness that are actually used in practice, see http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf.

at most $1/2^\lambda$ to the final distinguishing advantage. A successful distinguisher must be able to recognize a huge number of outputs of G so it can behave differently on them, but there are exponentially many G-outputs, and they may not follow any easy-to-describe pattern.

Below is an example that explores these ideas in more detail.

Example *Let G be a length-doubling PRG as above, let t be an arbitrary string $t \in \{\texttt{0},\texttt{1}\}^{2\lambda}$, and consider the following distinguisher $\mathcal{A}_t$ that has the value t hard-coded:*

```
$\mathcal{A}_t$ :
  $z \leftarrow$ QUERY()
  return $z \overset{?}{=} t$
```
.

What is the distinguishing advantage of $\mathcal{A}_t$?

We can see easily what happens when $\mathcal{A}_t$ is linked to $\mathcal{L}^G_{\textsf{prg-rand}}$. We get:

$$\Pr[\mathcal{A}_t \diamond \mathcal{L}^G_{\textsf{prg-rand}} \Rightarrow 1] = 1/2^{2\lambda}.$$

What happens when linked to $\mathcal{L}^G_{\textsf{prg-real}}$ depends on whether t is a possible output of G, which is a simple property of G. We always allow distinguishers to depend arbitrarily on G. In particular, it is fine for a distinguisher to "know" whether its hard-coded value t is a possible output of G. What the distinguisher "doesn't know" is which library it is linked to, and the value of s that was chosen in the case that it is linked to $\mathcal{L}^G_{\textsf{prg-real}}$.

Suppose for simplicity that G is injective (the exercises explore what happens when G is not). If t is a possible output of G, then there is exactly one choice of s such that $G(s) = t$, so we have:

$$\Pr[\mathcal{A}_t \diamond \mathcal{L}^G_{\textsf{prg-real}} \Rightarrow 1] = 1/2^{\lambda}.$$

Hence, the distinguishing advantage of $\mathcal{A}_t$ is $\left|1/2^{2\lambda} - 1/2^\lambda\right| \leqslant 1/2^\lambda$. If t is not a possible output of G, then we have:

$$\Pr[\mathcal{A}_t \diamond \mathcal{L}^G_{\textsf{prg-real}} \Rightarrow 1] = 0.$$

Hence, the distinguishing advantage of $\mathcal{A}_t$ is $\left|1/2^{2\lambda} - 0\right| = 1/2^{2\lambda}$.

In either case, $\mathcal{A}_t$ has negligible ditsinguishing advantage. This merely shows that $\mathcal{A}_t$ (for any hard-coded t) is not a particularly helpful distinguisher for any PRG. Of course, any candidate PRG might be insecure because of other distinguishers, but this example should serve to illustrate that PRG security is at least compatible with the fact that some strings are impossible outputs of the PRG.

Related Concept: Random Number Generation

The definition of a PRG includes a *uniformly sampled* seed. In practice, this PRG seed has to come from somewhere. Generally a source of "randomness" is provided by the hardware or operating system, and the process that generates these random bits is (confusingly) called a random *number* generator (RNG).

In this course we won't cover low-level random *number* generation, but merely point out what makes it different than the PRGs that we study:

- The job of a PRG is to take a small amount of "ideal" (in other words, uniform) randomness and extend it.
- By contrast, an RNG usually takes many inputs over time and maintains an internal state. These inputs are often from physical/hardware sources. While these inputs are "noisy" in some sense, it is hard to imagine that they would be statistically *uniform*. So the job of the RNG is to "refine" (sometimes many) sources of noisy data into uniform outputs.

Perspective on this Chapter

PRGs are a fundamental cryptographic building block that can be used to construct more interesting things. But you are unlikely to ever find yourself designing your own PRG or building something from a PRG that couldn't be built from some higher-level primitive instead. For that reason, we will not discuss specific PRGs or how they are designed in practice. Rather, the purpose of this chapter is to build your understanding of the concepts (like pseudorandomness, indistinguishability, proof techniques) that will be necessary in the rest of the class.

5.2 Application: Shorter Keys in One-Time-Secret Encryption

PRGs essentially convert a smaller amount of uniform randomness into a larger amount of *good-enough-for-polynomial-time* randomness. So a natural first application of PRGs is to obtain one-time-secure encryption but with keys that are shorter than the plaintext. This is the first of many acts of crypto-magic which are impossible except when restricting our focus to polynomial-time adversaries.

The main idea is to hold a short key k, expand it to a longer string using a PRG G, and use the result as a one-time pad on the (longer) plaintext. More precisely, let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a PRG, and define the following encryption scheme:

Construction 5.2 (Pseudo-OTP)

```
$\mathcal{K} = \{0,1\}^\lambda$                KeyGen:                   Enc($k,m$):                 Dec($k,c$):
$\mathcal{M} = \{0,1\}^{\lambda+\ell}$           $k \leftarrow \mathcal{K}$                 return $G(k) \oplus m$         return $G(k) \oplus c$
$\mathcal{C} = \{0,1\}^{\lambda+\ell}$           return $k$
```

The resulting scheme will not have (perfect) one-time secrecy. That is, encryptions of m_L and m_R will not be identically distributed in general. However, the distributions will be *indistinguishable* if G is a secure PRG. The precise flavor of security obtained by this construction is the following.

Definition 5.3 *Let Σ be an encryption scheme, and let $\mathcal{L}^{\Sigma}_{\text{ots-L}}$ and $\mathcal{L}^{\Sigma}_{\text{ots-R}}$ be defined as in Definition 2.6 (and repeated below for convenience). Then Σ has **(computational) one-time secrecy** if $\mathcal{L}^{\Sigma}_{\text{ots-L}} \approx$*

$\mathcal{L}^{\Sigma}_{\text{ots-R}}$. *That is, if for all polynomial-time distinguishers* $\mathcal{A}$, *we have* $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-L}} = 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ots-R}} = 1]$.

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

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

Claim 5.4 *Let* pOTP$[G]$ *denote Construction 5.2 instantiated with PRG* G. *If* G *is a secure PRG then* pOTP$[G]$ *has computational one-time secrecy.*

Proof We must show that $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}} \approx \mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$. As usual, we will proceed using a sequence of hybrids that begins at $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}}$ and ends at $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$. For each hybrid library, we will demonstrate that it is indistinguishable from the previous one. Note that we are allowed to use the fact that G is a secure PRG. In practical terms, this means that if we can express some hybrid library in terms of $\mathcal{L}^{G}_{\text{prg-real}}$ (one of the libraries in the PRG security definition), we can replace it with its counterpart $\mathcal{L}^{G}_{\text{prg-rand}}$ (or vice-versa). The PRG security of G says that such a change will be indistinguishable.

$\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}}$:

```
$\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}}$
QUERY($m_L, m_R \in \{0,1\}^{\lambda+\ell}$):
    $k \leftarrow \{0,1\}^{\lambda}$
    $c := G(k) \oplus m_L$
    return $c$
```

The starting point is $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}}$, shown here with the details of pOTP$[G]$ filled in.

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

```
QUERY($m_L, m_R$):
    $z \leftarrow$ QUERY'()
    $c := z \oplus m_L$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-real}}$
QUERY'():
    $s \leftarrow \{0,1\}^{\lambda}$
    return $G(s)$
```

The first hybrid step is to factor out the computations involving G, in terms of the $\mathcal{L}^{G}_{\text{prg-real}}$ library.

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

```
QUERY($m_L, m_R$):
    $z \leftarrow$ QUERY'()
    $c := z \oplus m_L$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-rand}}$
QUERY'():
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    return $z$
```

From the PRG security of G, we may replace the instance of $\mathcal{L}^{G}_{\text{prg-real}}$ with $\mathcal{L}^{G}_{\text{prg-rand}}$. The resulting hybrid library $\mathcal{L}_{\text{hyb-2}}$ is indistinguishable from the previous one.

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

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$
QUERY($m_L, m_R$):
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    $c := z \oplus m_L$
    return $c$
```

A subroutine has been inlined. Note that the resulting library is precisely $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$! Here, OTP is instantiated on plaintexts of size $\lambda + \ell$ (and the variable k has been renamed to z).

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

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$
QUERY($m_L, m_R$):
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    $c := z \oplus m_R$
    return $c$
```

The (perfect) one-time secrecy of OTP allows us to replace $\mathcal{L}^{\text{OTP}}_{\text{ots-L}}$ with $\mathcal{L}^{\text{OTP}}_{\text{ots-R}}$; they are interchangeable.

The rest of the proof is essentially a “mirror image” of the previous steps, in which we perform the same steps but in reverse (and with m_R hanging around instead of m_L).

$\mathcal{L}_{\text{hyb-5}}$:

```
QUERY($m_L, m_R$):
    $z \leftarrow$ QUERY′()
    $c := z \oplus m_R$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-rand}}$
QUERY′():
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    return $z$
```

A statement has been factored out into a subroutine, which happens to exactly match $\mathcal{L}^{G}_{\text{prg-rand}}$.

$\mathcal{L}_{\text{hyb-6}}$:

```
QUERY($m_L, m_R$):
    $z \leftarrow$ QUERY′()
    $c := z \oplus m_R$
    return $c$
```

$\diamond$

```
$\mathcal{L}^{G}_{\text{prg-real}}$
QUERY′():
    $s \leftarrow \{0,1\}^{\lambda}$
    return $G(s)$
```

From the PRG security of G, we can replace $\mathcal{L}^{G}_{\text{prg-rand}}$ with $\mathcal{L}^{G}_{\text{prg-real}}$. The resulting library is indistinguishable from the previous one.

$\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$:

```
$\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$
QUERY($m_L, m_R$):
    $k \leftarrow \{0,1\}^{\lambda}$
    $c := G(k) \oplus m_R$
    return $c$
```

A subroutine has been inlined. The result is $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$.

Summarizing, we showed a sequence of hybrid libraries satisfying the following:

$$\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}.$$

Hence, $\mathcal{L}^{\text{pOTP}[G]}_{\text{ots-L}} \approx \mathcal{L}^{\text{pOTP}[G]}_{\text{ots-R}}$, and pOTP has (computational) one-time secrecy. ■

★ 5.3 Taking the Contrapositive Point-of-View

We just proved the statement "if G is a secure PRG, then pOTP[G] has one-time secrecy," but let's also think about the contrapositive of that statement:

> If the pOTP[G] scheme is **not** one-time secret, then G is **not** a secure PRG.

If the pOTP scheme is not secure, then there is some distinguisher $\mathcal{A}$ that can distinguish the two $\mathcal{L}_{\textsf{ots-}\star}$ libraries with better than negligible advantage. Knowing that such an $\mathcal{A}$ exists, can we indeed break the security of G?

Imagine going through the sequence of hybrid libraries with this hypothetical $\mathcal{A}$ as the calling program. We know that one of the steps of the proof must break down since $\mathcal{A}$ successfully distinguishes between the endpoints of the hybrid sequence. Some of the steps of the proof were unconditional; for example, factoring out and inlining subroutines *never* has an effect on the calling program. These steps of the proof always hold; they are the steps where we write $\mathcal{L}_{\textsf{hyb-}i} \equiv \mathcal{L}_{\textsf{hyb-}(i+1)}$.

The steps where we write $\mathcal{L}_{\textsf{hyb-}i} \approx \mathcal{L}_{\textsf{hyb-}(i+1)}$ are *conditional.* In our proof, the steps $\mathcal{L}_{\textsf{hyb-1}} \approx \mathcal{L}_{\textsf{hyb-2}}$ and $\mathcal{L}^{\textsf{OTP}}_{\textsf{ots-R}} \approx \mathcal{L}_{\textsf{hyb-4}}$ relied on G being a secure PRG. So if a hypothetical $\mathcal{A}$ was able to break the security of pOTP, then that same $\mathcal{A}$ *must* also successfully distinguish between $\mathcal{L}_{\textsf{hyb-1}}$ and $\mathcal{L}_{\textsf{hyb-2}}$, or between $\mathcal{L}_{\textsf{hyb-5}}$ and $\mathcal{L}_{\textsf{hyb-6}}$ — the only conditional steps of the proof.

Let's examine the two cases:

- ► Suppose the hypothetical $\mathcal{A}$ successfully distinguishes between $\mathcal{L}_{\textsf{hyb-1}}$ and $\mathcal{L}_{\textsf{hyb-2}}$. Let's recall what these libraries actually look like:

$$\mathcal{L}_{\textsf{hyb-1}} = \boxed{\begin{array}{l} \underline{\textsc{query}(m_L, m_R):} \\ \quad z \leftarrow \textsc{query}'() \\ \quad c = z \oplus m_L \\ \quad \text{return } c \end{array}} \diamond \mathcal{L}^G_{\textsf{prg-real}};$$

$$\mathcal{L}_{\textsf{hyb-2}} = \boxed{\begin{array}{l} \underline{\textsc{query}(m_L, m_R):} \\ \quad z \leftarrow \textsc{query}'() \\ \quad c = z \oplus m_L \\ \quad \text{return } c \end{array}} \diamond \mathcal{L}^G_{\textsf{prg-rand}}.$$

  Interestingly, these two libraries share a common component that is linked to either $\mathcal{L}_{\textsf{prg-real}}$ or $\mathcal{L}_{\textsf{prg-rand}}$. (This is no coincidence!) Let's call that common library $\mathcal{L}^*$ and write

$$\mathcal{L}_{\textsf{hyb-1}} = \mathcal{L}^* \diamond \mathcal{L}^G_{\textsf{prg-real}}; \qquad \mathcal{L}_{\textsf{hyb-2}} = \mathcal{L}^* \diamond \mathcal{L}^G_{\textsf{prg-rand}}.$$

  Since $\mathcal{A}$ successfully distinguishes between $\mathcal{L}_{\textsf{hyb-1}}$ and $\mathcal{L}_{\textsf{hyb-2}}$, the following advantage is non-negligible:

$$\Big| \Pr[\mathcal{A} \diamond \mathcal{L}^* \diamond \mathcal{L}^G_{\textsf{prg-real}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}^* \diamond \mathcal{L}^G_{\textsf{prg-rand}} \Rightarrow 1] \Big|.$$

  But with a change of perspective, this means that $\mathcal{A} \diamond \mathcal{L}^*$ is a calling program that successfully distinguishes $\mathcal{L}^G_{\textsf{prg-real}}$ from $\mathcal{L}^G_{\textsf{prg-rand}}$. In other words, $A \diamond \mathcal{L}^*$ **breaks the PRG security of G!**

- Suppose the hypothetical $\mathcal{A}$ only distinguishes $\mathcal{L}_{\text{hyb-5}}$ from $\mathcal{L}_{\text{hyb-6}}$. Going through the reasoning above, we will reach a similar conclusion but with a different $\mathcal{L}^*$ than before (in fact, it will be mostly the same library but with m_L replaced with m_R).

So you can think of our security proof as a very roundabout way of saying the following:

> *If you give me an adversary/distinguisher $\mathcal{A}$ that breaks the one-time secrecy of* pOTP$[G]$, *then I can use it to build an adversary that breaks the PRG security of G. More specifically, G is guaranteed to be broken by (at least) one of the two distinguishers:*[2]

$$\mathcal{A} \diamond \boxed{\begin{array}{l} \underline{\textsc{query}(m_L, m_R):} \\ \quad z \leftarrow \textsc{query}'() \\ \quad c = z \oplus m_L \\ \quad \text{return } c \end{array}} \qquad \textit{or} \qquad \mathcal{A} \diamond \boxed{\begin{array}{l} \underline{\textsc{query}(m_L, m_R):} \\ \quad z \leftarrow \textsc{query}'() \\ \quad c = z \oplus m_R \\ \quad \text{return } c \end{array}}\,.$$

In fact, this would be the "traditional" method for proving security that you might see in many other cryptography texts. You would suppose you are given an adversary $\mathcal{A}$ breaking pOTP, then you would demonstrate why at least one of the adversaries given above breaks the PRG. Unfortunately, it is quite difficult to explain to students how one is supposed to *come up with* these PRG-adversaries in terms of the one-time-secrecy adversaries. For that reason, we will reason about proofs in the "if this is secure then that is secure" realm, rather than the contrapositive "if that is insecure then this is insecure."

5.4 Extending the Stretch of a PRG

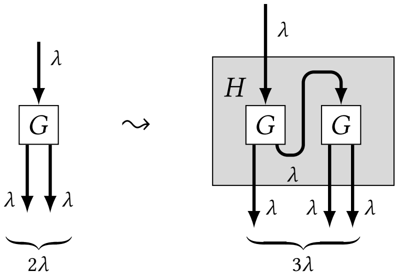
Recall that the *stretch* of a PRG is the amount by which the PRG's output length exceeds its input length. A PRG with very long stretch seems much more useful than one with small stretch. Is there a limit to the stretch of a PRG? Using only λ bits of true uniform randomness, can we generate 100λ, or even λ^3 pseudorandom bits?

In this section we will see that once you can extend a PRG a little bit, you can also extend it a lot. This is another magical feat that is possible with pseudorandomness but not with truly uniform distributions. We will demonstrate the concept by extending a PRG with stretch λ into one with stretch 2λ, but the idea can be used to increase the stretch of any PRG indefinitely (see the exercises).

Construction 5.5 (PRG feedback) *Let $G : \{\mathtt{0},\mathtt{1}\}^\lambda \to \{\mathtt{0},\mathtt{1}\}^{2\lambda}$ be a length-doubling PRG* (i.e., *a PRG with stretch λ). When $x \in \{\mathtt{0},\mathtt{1}\}^{2\lambda}$, we write x_{left} to denote the leftmost λ bits of x and x_{right} to denote the rightmost λ bits. Define the length-tripling function $H : \{\mathtt{0},\mathtt{1}\}^\lambda \to \{\mathtt{0},\mathtt{1}\}^{3\lambda}$ as follows:*

[2] The statement is somewhat non-constructive, since we don't know for sure which of the two distinguishers will be the one that actually works. But a way around this is to consider a single distinguisher that flips a coin and acts like the first one with probability 1/2 and acts like the second one with probability 1/2.

```
$H(s)$:
    $x := G(s)$
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```



Claim 5.6 *If G is a secure length-doubling PRG, then H (defined above) is a secure length-tripling PRG.*

Proof We want to show that $\mathcal{L}^H_{\text{prg-real}} \approx \mathcal{L}^H_{\text{prg-rand}}$. As usual, we do so with a hybrid sequence. Since we assume that G is a secure PRG, we are allowed to use the fact that $\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}^G_{\text{prg-rand}}$. In this proof, we will use the fact twice: once for each occurrence of G in the code of H.

$\mathcal{L}^H_{\text{prg-real}}$:

```
$\mathcal{L}^H_{\text{prg-real}}$
QUERY():
    $s \leftarrow \{0,1\}^\lambda$
    $x := G(s)$
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```

The starting point is $\mathcal{L}^H_{\text{prg-real}}$, shown here with the details of H filled in.

```
QUERY():
    $x :=$ QUERY'()
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```

$\diamond$

```
$\mathcal{L}^G_{\text{prg-real}}$
QUERY'():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

The first invocation of G has been factored out into a subroutine. The resulting hybrid library includes an instance of $\mathcal{L}^G_{\text{prg-real}}$.

```
QUERY():
    $x :=$ QUERY'()
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```

$\diamond$

```
$\mathcal{L}^G_{\text{prg-rand}}$
QUERY'():
    $z \leftarrow \{0,1\}^{2\lambda}$
    return $z$
```

From the PRG security of G, we can replace the instance of $\mathcal{L}^G_{\text{prg-real}}$ with $\mathcal{L}^G_{\text{prg-rand}}$. The resulting hybrid library is indistinguishable.

```
QUERY():
    $x \leftarrow \{0,1\}^{2\lambda}$
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```

A subroutine has been inlined.

```
QUERY():
    $x_{\text{left}} \leftarrow \{0,1\}^\lambda$
    $x_{\text{right}} \leftarrow \{0,1\}^\lambda$
    $y := G(x_{\text{right}})$
    return $x_{\text{left}} \| y$
```

Choosing 2λ uniformly random bits and then splitting them into two halves has exactly the same effect as choosing λ uniformly random bits and independently choosing λ more.

```
QUERY():                          $\mathcal{L}^G_{\text{prg-real}}$
    $x_{\text{left}} \leftarrow \{0,1\}^\lambda$      $\diamond$     QUERY$'$():
    $y :=$ QUERY$'$()                   $s \leftarrow \{0,1\}^\lambda$
    return $x_{\text{left}} \| y$                  return $G(s)$
```

The remaining appearance of G has been factored out into a subroutine. Now $\mathcal{L}^G_{\text{prg-real}}$ makes its second appearance.

```
QUERY():                          $\mathcal{L}^G_{\text{prg-rand}}$
    $x_{\text{left}} \leftarrow \{0,1\}^\lambda$      $\diamond$     QUERY$'$():
    $y :=$ QUERY$'$()                   $z \leftarrow \{0,1\}^{2\lambda}$
    return $x_{\text{left}} \| y$                  return $z$
```

Again, the PRG security of G lets us replace $\mathcal{L}^G_{\text{prg-real}}$ with $\mathcal{L}^G_{\text{prg-rand}}$. The resulting hybrid library is indistinguishable.

```
QUERY():
    $x_{\text{left}} \leftarrow \{0,1\}^\lambda$
    $y \leftarrow \{0,1\}^{2\lambda}$
    return $x_{\text{left}} \| y$
```

A subroutine has been inlined.

```
$\mathcal{L}^H_{\text{prg-rand}}$:
    $\mathcal{L}^H_{\text{prg-rand}}$
    QUERY():
        $z \leftarrow \{0,1\}^{3\lambda}$
        return $z$
```

Similar to above, concatenating λ uniform bits with 2λ independently uniform bits has the same effect as sampling 3λ uniform bits. The result of this change is $\mathcal{L}^H_{\text{prg-rand}}$.

Through this sequence of hybrid libraries, we showed that:

$$\mathcal{L}^H_{\text{prg-real}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}_{\text{hyb-7}} \equiv \mathcal{L}^H_{\text{prg-rand}}.$$

Hence, H is a secure PRG. ■

Exercises

5.1. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$. Define $\text{im}(G) = \{y \in \{0,1\}^{\lambda+\ell} \mid \exists s : G(s) = y\}$, the set of possible outputs of G. For simplicity, assume that G is injective (i.e., 1-to-1).

(a) What is $|\text{im}(G)|$, as a function of λ and ℓ?

(b) A string y is chosen randomly from $\{0,1\}^{\lambda+\ell}$. What is the probability that $y \in \text{im}(G)$, expressed as a function of λ and ℓ?

5.2. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be an injective PRG. Consider the following distinguisher:

```
$\mathcal{A}$
$x :=$ QUERY()
for all $s' \in \{0,1\}^\lambda$:
  if $G(s') = x$ then return 1
return 0
```

(a) What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ and $\mathcal{L}^G_{\text{prg-rand}}$? Is it neglgible?

(b) Does this contradict the fact that G is a PRG? Why or why not?

(c) What happens to the advantage if G is not injective?

5.3. Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be an injective PRG, and consider the following distinguisher:

```
$\mathcal{A}$
$x :=$ QUERY()
$s' \leftarrow \{0,1\}^\lambda$
return $G(s') \overset{?}{=} x$
```

What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ from $\mathcal{L}^G_{\text{prg-rand}}$?

Hint: When computing $\Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{prg-rand}}$ outputs 1], separate the probabilities based on whether $x \in \text{im}(G)$ or not. (im(G) is defined in a previous problem)

5.4. For any PRG $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ there will be many strings in $\{0,1\}^{\lambda+\ell}$ that are impossible to get as output of G. Let S be any such set of impossible G-outputs, and consider the following adversary that has S hard-coded:

```
$\mathcal{A}$
$x :=$ QUERY()
return $x \overset{?}{\in} S$
```

What is the advantage of $\mathcal{A}$ in distinguishing $\mathcal{L}^G_{\text{prg-real}}$ from $\mathcal{L}^G_{\text{prg-rand}}$? Why does an adversary like this one not automatically break every PRG?

5.5. Show that the scheme from Section 5.2 does not have *perfect* one-time secrecy, by showing that there must exist two messages m_1 and m_2 whose ciphertext distributions differ.

Hint: There must exist strings $s_1, s_2 \in \{0,1\}^{2\lambda}$ where $s_1 \in \text{im}(G)$, and $s_2 \notin \text{im}(G)$. Use these two strings to find two messages m_1 and m_2 whose ciphertext distributions assign different probabilities to s_1 and s_2. Note that it is legitimate for an attacker to “know” s_1 and s_2, as these are properties of G alone, and independent of the random choices made when executing the scheme.

5.6. (a) Let f be any function. Show that the following function G is **not** a secure PRg (even if f is!). Describe a successful distinguisher and explicitly compute its advantage:

```
$G(s)$:
  return $s \| f(s)$
```

(b) Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ be a candidate PRG. Suppose there is a polynomial-time algorithm V with the property that it inverts G with non-negligible probability. That is,

$$\Pr_{s \leftarrow \{0,1\}^\lambda} \Big[V(G(s)) = s \Big] \text{ is non-negligible.}$$

Show that if an algorithm V exists with this property, then G is not a secure PRG. In other words, construct a distinguisher contradicting the PRG-security of G and show that it achieves non-negligible distinguishing advantage.

Note: Don't assume anything about the output of V other than the property shown above. In particular, V might very frequently output the "wrong" thing.

5.7. In the "PRG feedback" construction H in Section 5.4, there are two calls to G. The security proof applies the PRG security rule to both of them, starting with the first. Describe what happens when you try to apply the PRG security of G to these two calls in the opposite order. Does the proof still work, or must it be in the order that was presented?

5.8. Let $\ell' > \ell > 0$. Extend the "PRG feedback" construction to transform any PRG of stretch ℓ into a PRG of stretch ℓ'. Formally define the new PRG and prove its security using the security of the underlying PRG.

5.9. Let $G : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ be a secure length-tripling PRG. For each function below, state whether it is also a secure PRG. If the function is a secure PRG, give a proof. If not, then describe a successful distinguisher and explicitly compute its advantage.

(a)

```
$H(s)$:
  $x := G(s)$
  $y :=$ first $\lambda$ bits of $x$
  $z :=$ last $\lambda$ bits of $x$
  return $G(y) \| G(z)$
```

(b)

```
$H(s)$:
  $x := G(s)$
  $y :=$ first $2\lambda$ bits of $x$
  return $y$
```

(c)

```
$H(s)$:
  $x := G(s)$
  $y := G(s)$
  return $x \| y$
```

(d)

```
$H(s)$:
  $x := G(s)$
  $y := G(0^\lambda)$
  return $x \| y$
```

(e)

```
$H(s)$:
  $x := G(s)$
  $y := G(0^\lambda)$
  return $x \oplus y$
```

(f)

```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{3\lambda}$

$H(s)$:
  $x := G(s_{\text{left}})$
  $y := G(s_{\text{right}})$
  return $x \oplus y$
```

(g)

```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{6\lambda}$

$H(s)$:
  $x := G(s_{\text{left}})$
  $y := G(s_{\text{right}})$
  return $x \| y$
```

5.10. Let $G : \{0,1\}^{\lambda} \to \{0,1\}^{3\lambda}$ be a secure length-tripling PRG. Prove that each of the following functions is also a secure PRG:

(a)

```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{4\lambda}$

$H(s)$:
  $y := G(s_{\text{right}})$
  return $s_{\text{left}} \| y$
```

Note that H includes half of its input directly in the output. How do you reconcile this fact with the conclusion of Exercise 5.6(b)?

(b)

```
// $H : \{0,1\}^{2\lambda} \to \{0,1\}^{3\lambda}$

$H(s)$:
  return $G(s_{\text{left}})$
```

5.11. A frequently asked question in cryptography forums is whether it's possible to determine which PRG implementation was used by looking at output samples.

Let G_1 and G_2 be two PRGs with matching input/output lengths. Define two libraries $\mathcal{L}^{G_1}_{\text{which-prg}}$ and $\mathcal{L}^{G_2}_{\text{which-prg}}$ as follows:

```
$\mathcal{L}^{G_1}_{\text{which-prg}}$

QUERY():
  $s \leftarrow \{0,1\}^{\lambda}$
  return $G_1(s)$
```

```
$\mathcal{L}^{G_2}_{\text{which-prg}}$

QUERY():
  $s \leftarrow \{0,1\}^{\lambda}$
  return $G_2(s)$
```

Prove that if G_1 and G_2 are both secure PRGs, then $\mathcal{L}^{G_1}_{\text{which-prg}} \approx \mathcal{L}^{G_2}_{\text{which-prg}}$ — that is, it is infeasible to distinguish which PRG was used simply by receiving output samples.

★ 5.12. Prove that if PRGs exist, then P ≠ NP.

Hint: $\{y \mid \exists s : G(s) = y\} \in \text{NP}$.

Note: This implies that $\mathcal{L}^{G}_{\text{prg-real}} \not\equiv \mathcal{L}^{G}_{\text{prg-rand}}$ for all G. That is, there can be no PRG that is secure against computationally unbounded distinguishers.

6 Pseudorandom Functions

A pseudorandom generator allows us to take a small amount of uniformly sampled bits, and "amplify" them into a larger amount of uniform-looking bits. A PRG must run in polynomial time, so the length of its pseudorandom output can only be polynomial in the security parameter. But what if we wanted *even more* pseudorandom output? Is it possible to take λ uniformly sampled bits and generate 2^λ pseudorandom bits?

Perhaps surprisingly, the answer is yes. The catch is that we have to change the rules slightly. Since a PRG must run in polynomial time, it does not have time to even *write down* an exponentially long output (let alone compute it). On top of that, a distinguisher would not have enough running time to even *read* it.

To avoid these quite serious problems, we settle for *random access* to the bits of the very large pseudorandom output. Imagine augmenting a PRG to take an extra input i. Given the short seed $s \in \{\mathtt{0},\mathtt{1}\}^\lambda$ and index i, the PRG should output (in polynomial time) the ith bit of this exponentially-long pseudorandom output. In this way, the short seed *implicitly* determines an exponentially long output which never needs to be *explicitly* written down. If F is the function in question, then the key/seed k implicitly determines the long pseudorandom output:

$$F(k,\mathtt{0}\cdots\mathtt{00}) \| F(k,\mathtt{0}\cdots\mathtt{01}) \| F(k,\mathtt{0}\cdots\mathtt{10}) \| \cdots \| F(k,\mathtt{1}\cdots\mathtt{10}) \| F(k,\mathtt{1}\cdots\mathtt{11}).$$

Since we have changed the necessary syntax, we refer to such a function F as a **pseudorandom function (PRF)** rather than a PRG.

6.1 Definition

You can think of a PRF as a way to achieve random access to a very long [pseudo]random string. You can also think of that long string as a "lookup table" for a function $F(k,\cdot)$. The string is broken up into blocks of length *out*, and the rth block of the string is $F(k,r)$, where we are treating r as both an integer and a length-*in* binary string. A distinguisher is allowed to query this function, and the results should look as though they came from a *random function* — a function whose lookup table was chosen uniformly at random. This perspective is the source of the name "pseudorandom *function*;" it's an object that (when the seed is chosen uniformly) is indistinguishable from a random function, to polynomial-time computations.

Under this perspective, we can consider a function with any output length, not just with a single bit of output. Consider the set $\mathcal{F}$ of functions from $\{\mathtt{0},\mathtt{1}\}^{in}$ to $\{\mathtt{0},\mathtt{1}\}^{out}$. If F is a good PRF, then when instantiated with a uniformly chosen k, the function $F(k,\cdot)$ should look just like a function chosen uniformly from the set $\mathcal{F}$, when given query access to the function.

We can therefore formalize security for PRFs by defining two libraries that provide an interface to query a function on arbitrary inputs. The two libraries differ in whether the responses are determined by evaluating a PRF (with randomly chosen seed) or by evaluating a function, all of whose outputs have been chosen independently at random.

For technical reasons, it is important that the libraries for the PRF definition run in polynomial time. This means we must be careful about the library that provides access to a truly random function. In polynomial time we cannot sample the entire lookup table of a random function, since such a lookup table can be exponential in size. Instead, outputs of the function are sampled on demand and cached for later in an associative array. The formal definition looks like this:

Definition 6.1 (PRF security) *Let $F : \{0,1\}^\lambda \times \{0,1\}^{in} \to \{0,1\}^{out}$ be a deterministic function. We say that F is a secure **pseudorandom function (PRF)** if $\mathcal{L}^F_{\text{prf-real}} \approx \mathcal{L}^F_{\text{prf-rand}}$, where:*

```
$\mathcal{L}^F_{\text{prf-real}}$

$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{in}$):
    return $F(k,x)$
```

```
$\mathcal{L}^F_{\text{prf-rand}}$

$T$ := empty assoc. array

QUERY($x \in \{0,1\}^{in}$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{out}$
    return $T[x]$
```

Discussion

- We measure the security of a PRF by comparing its outputs to those of a "random function." The idea of a "random function" is tricky. We do **not** mean a function that gives different outputs when called twice on the same input. Instead, we mean a function whose lookup table is chosen uniformly at random, after which the lookup table is absolutely fixed. Calling the function twice on the same input will result in the same output.

  But due to the technical limitations mentioned above, we don't actually write the code of $\mathcal{L}_{\text{prf-rand}}$ in this way. The lookup table is too large to sample and store explicitly. Rather, we have to sample the values appearing in the lookup table on-demand. But we see that once they are chosen, the variable T ensures that they remain fixed.

  We see also that in $\mathcal{L}_{\text{prf-real}}$, the function F is deterministic and the key k is static between calls to QUERY. So calling QUERY twice on the same input gives the same answer.

- If the input length *in* is large (*e.g.*, if $in = \lambda$), then the calling program will not be able to query the function on all inputs $x \in \{0,1\}^{in}$. In fact, it can only query on a *negligible fraction* of the input space $\{0,1\}^{in}$!

- Suppose we know that the calling program is **guaranteed** to never call the QUERY subroutine on the same x twice. Then there is no need for $\mathcal{L}_{\text{prf-rand}}$ to keep track of T. The library could just as well forget the value that it gave in response to x since we have a guarantee that it will never be requested again. In that case, we can considerably simplify the code of $\mathcal{L}_{\textsf{prf-rand}}$ as follows:

```
QUERY($x \in \{0,1\}^{in}$):
  $z \leftarrow \{0,1\}^{out}$
  return $z$
```
.

Instantiations

In implementations of cryptographic systems, "provable security" typically starts only above the level of PRFs. In other words, we typically build a cryptographic system and prove its security under the *assumption* that we have used a secure PRF as one of the building blocks. PRFs are a kind of "ground truth" of applied cryptography.

By contrast, the algorithms that we use in practice as PRFs are not *proven* to be secure PRFs based on any simpler assumption. Rather, these algorithms are subjected to intense scrutiny by cryptographers, they are shown to resist all known classes of attacks, etc. All of these things taken together serve as evidence supporting the assumption that these algorithms are secure PRFs.

An example of a conjectured PRF is the Advanced Encryption Standard (AES). It is typically classified as a *block cipher*, which refers to a PRF with some extra properties that will be discussed in the next chapter. In particular, AES can be used with a security parameter of $\lambda \in \{128, 192, 256\}$, and has input/output length $in = out = 128$. The amount of scrutiny applied to AES since it was first published in 1998 is considerable. To date, it is not known to be vulnerable to any severe attacks. The best known attacks on AES are only a tiny bit faster than brute-force, and still vastly beyond the realm of feasible computation.

6.2 Attacking Insecure PRFs

We can gain more understanding of the PRF security definition by attacking insecure PRF constructions. Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ be a length-doubling PRG, and define a "PRF" F as follows:

```
$F(k,x)$:
  return $G(k) \oplus x$
```

Note the resemblance to our one-time secret encryption scheme. Indeed, we have previously shown that $G(k) \oplus m$ is a valid method for one-time encryption, and that the resulting ciphertexts are pseudorandom. Despite that, the above construction is **not** a secure PRF!

So let us try to attack the PRF security of F. That means writing down a distinguisher that behaves as differently as possible in the presence of the two $\mathcal{L}^F_{\textsf{prf-}\star}$ libraries. Importantly, we want to show that *even if G is an excellent PRG*, F is still not a secure PRF. So we **must not** try to break the PRG security of G. We are trying to break the inappropriate way that G is used to construct a PRF.

We want to construct a distinguisher that uses the interface of the $\mathcal{L}_{\textsf{prf-}\star}$ libraries. That is, it should make some calls to the QUERY subroutine and make a decision based on the answers it gets. The QUERY subroutine takes an argument, so we must specify which arguments to use.

One observation we can make is that if a calling program sees *only one* value of the form $G(k) \oplus x$, it will look pseudorandom. This is essentially what we showed in Section 5.2. So we should be looking for a calling program that makes more than one call to QUERY.

If we make two calls to QUERY — say, on inputs x_1 and x_2 — the responses will be $G(k) \oplus x_1$ and $G(k) \oplus x_2$. To be a secure PRF, these responses must look *independent* and uniform. Do they? A moment's thought shows that they do not appear to be independent. The XOR of these two values is always $x_1 \oplus x_2$, and this is a value that is known to the calling program!

At a more philosophical level, we wanted to figure out exactly how F is using the PRG in an inappropriate way. The security of a PRG comes from the fact that its seed is uniformly chosen and never used elsewhere (revisit Definition 5.1 and appreciate that the s variable is private and falls out of scope after $G(s)$ is computed), but this F allows the same seed to be used twice in different contexts where the results are supposed to be independent. This is precisely the way in which F uses G inappropriately.

We can therefore package all of our observations into the following distinguisher:

```
$\mathcal{A}$
pick $x_1, x_2 \in \{0,1\}^{2\lambda}$ *arbitrarily* so that $x_1 \neq x_2$
$z_1 \leftarrow$ QUERY$(x_1)$
$z_2 \leftarrow$ QUERY$(x_2)$
return $z_1 \oplus z_2 \overset{?}{=} x_1 \oplus x_2$
```

Clearly $\mathcal{A}$ runs in polynomial time in λ. Let us compute its advantage:

- When $\mathcal{A}$ is linked to $\mathcal{L}^F_{\text{prf-real}}$, the library will choose a key k. Then z_1 is set to $G(k) \oplus x_1$ and z_2 is set to $G(k) \oplus x_2$. So $z_1 \oplus z_2$ is *always* equal to $x_1 \oplus x_2$, and $\mathcal{A}$ outputs 1. That is, $\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-real}} \Rightarrow 1] = 1$.

- When $\mathcal{A}$ is linked to $\mathcal{L}^F_{\text{prf-rand}}$, the responses of the two calls to QUERY will be chosen uniformly and independently because different arguments to QUERY were used. Consider the moment in time when the second call to QUERY is about to happen. At that point, x_1, x_2, and z_1 have all been determined, while z_2 is about to be chosen uniformly by the library. Using the properties of XOR, we see that $\mathcal{A}$ will output 1 if and only if z_2 is chosen to be precisely the fixed value $x_1 \oplus x_2 \oplus z_1$. This happens only with probability $1/2^{2\lambda}$. That is, $\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-rand}} \Rightarrow 1] = 1/2^{2\lambda}$.

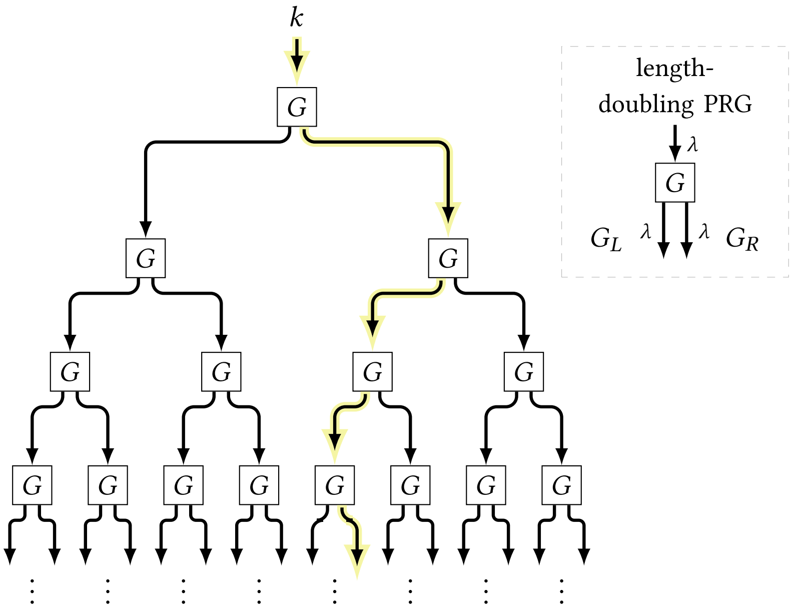
The advantage of $\mathcal{A}$ is therefore $|1 - 1/2^{2\lambda}|$ which is non-negligible (in fact, it is negligibly close to 1). This shows that F is not a secure PRF.

⋆ 6.3 A Theoretical Construction of a PRF from a PRG

Despite the fact that secure PRFs are often simply *assumed* in practice, we show in this section that secure PRFs can in fact be *provably* constructed from secure PRGs. The resulting PRF is much more impractical than what one would typically use in practice, but it serves to illustrate the connection between these two fundamental primitives.

We have already seen that it is possible to feed the output of a PRG back into the PRG again, to extend its stretch. We can in fact use the same approach to extend a PRG into a PRF. The trick is to feed PRG outputs back into the PRG in the structure of a **binary tree** (similar to Exercise 5.9(a)) rather than a linked list. The leaves of the tree correspond to final outputs of the PRF. Importantly, the binary tree can be exponentially large, and yet random access to its leaves is possible in polynomial time by simply traversing the relevant portion of the tree. This construction of a PRF is known as the GGM construction, after its creators Goldreich, Goldwasser, and Micali (because cryptography likes three letter acronyms).

In more detail, let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ be a length-doubling PRG. For convenience, let $G_L(k)$ and $G_R(k)$ denote the first λ bits and last λ bits of $G(k)$, respectively. The main idea is to imagine a complete binary tree of height *in* (*in* will be the input length of the PRF), where each node in the tree represents an application of G. If a node gets input v, then it sends $G_L(v)$ to its left child and sends $G_R(v)$ to its right child. There will be 2^{in} leaves, whose values will be the outputs of the PRF. To access the leaf with index $x \in \{0,1\}^{in}$. we can traverse the tree from root to leaf, taking left and right turns at each node according to the bits of x. Below is a visualization of the construction, where the highlighted path corresponds to the computation of $F(k, 1001\cdots)$.



Construction 6.2 (GGM PRF)

```
in = arbitrary        F(k,x):
out = λ                 v := k
                        for i = 1 to in:
                          if x_i = 0 then v := G_L(v)
                          if x_i = 1 then v := G_R(v)
                        return v
```

Claim 6.3 *If G is a secure PRG, then Construction 6.2 is a secure PRF.*

As mentioned above, it is helpful to think about the internal states computed by the PRF in terms of an exponentially large binary tree. Let us give each node in this tree a unique name in binary. The name of the root is the empty string ϵ. The left and right children of a node x have names $x\texttt{0}$ and $x\texttt{1}$, respectively. Now imagine an associative array T mapping node names to λ-bit strings. Assign the PRF key k to the root, so $T[\epsilon] = k$. Then for every node x, set $T[x\texttt{0}] = G_L(T[x])$ and $T[x\texttt{1}] = G_R(T[x])$.

Then for all *in*-bit strings x, we have that $T[x] = F(k,x)$. The PRF algorithm can be thought of as an *efficient* way of coping with this exponentially large associative array. Starting with a uniform value placed at the root ($T[\epsilon]$), the PRF proceeds along the path from root to a leaf x by applying either G_L or G_R according to the bits of x.

Think about the top two layers of this tree. We have $T[\epsilon] = k$, $T[\texttt{0}] = G_L(T[\epsilon])$, and $T[\texttt{1}] = G_R(T[\epsilon])$. The security of the PRG G suggests that we can instead replace $T[\texttt{0}]$ and $T[\texttt{1}]$ with uniformly chosen λ-bit strings. Then when traversing the path from root to leaf, we skip the root and repeatedly apply G_L/G_R starting at either $T[\texttt{0}]$ or $T[\texttt{1}]$. Applying the PRG security of G allows us to "chop off" the top layer of the tree.

Taking this farther, we can imagine "chopping off" the top D layers of the tree. That is, setting $T[v] \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$ for $|v| \leqslant D$. Then on input x, we let p denote the first D bits of x, skip ahead to $T[p]$, and only then start applying G_L/G_R following the bits of x.

In this way we can eventually replace the entire associative array T with uniform random strings. At the end of this process, the leaves — which are the PRF outputs — will all be uniformly random, which is the goal in a proof of PRF security.

This is essentially the approach taken in our proof, but with one important caveat. If we follow the summary above, then as D grows larger we will need to initialize an exponential number of entries in T with random values. The resulting program will not be polynomial-time, and we cannot apply PRG security. The way around this is to observe that the calling program can only ever "ask for" a polynomial-size subset of our large binary tree. So instead of *eagerly* sampling all the required randomness upfront, we can *lazily* sample it at the last possible moment (just before it is needed).

Proof We prove the claim using a sequence of hybrids. In previous proofs, the number of hybrids was a fixed constant, but here the number of hybrids depends on the input-length parameter *in*. So we will do something a little different. Consider the following hybrid library:

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

$T$ := empty assoc. array

QUERY($x$):
  $p$ := first $D$ bits of $x$
  if $T[p]$ undefined:
    $T[p] \leftarrow \{\texttt{0},\texttt{1}\}^\lambda$
  $v := T[p]$
  for $i = D + 1$ to $in$:
    if $x_i = \texttt{0}$ then $v := G_L(v)$
    if $x_i = \texttt{1}$ then $v := G_R(v)$
  return $v$
```

The code of this library has an undefined variable D, which you should think of as a pre-processor macro if you're familiar with C programming. As the analysis proceeds, we will consider hard-coding particular values in place of D, but for any single execution of the library, D will be a fixed constant. We will write $\mathcal{L}_{\text{hyb-1}}[d]$ to denote the library that you get after hard-coding the statement $D = d$ (*i.e.*, "`#define D` d") into the library.

This library $\mathcal{L}_{\text{hyb-1}}$ essentially "chops off" the top D levels of the conceptual binary tree, as described before the proof. On input x, it skips ahead to node p in the tree, where p are the first D bits of x. Then it lazily samples the starting value $T[p]$ and proceeds to apply G_L/G_R as before.

Importantly, let's see what happens with extreme choices of $D \in \{0, in\}$. These cases are shown below with the code simplified to the right:

```
$\mathcal{L}_{\text{hyb-1}}[0]$

$T :=$ empty assoc. array                   $k :=$ undefined
                                            // $k$ is alias for $T[\epsilon]$
QUERY($x$):
  $p :=$ first $0$ bits of $x$                 $p = \epsilon$
  if $T[p]$ undefined:                       if $k$ undefined:
    $T[p] \leftarrow \{0,1\}^\lambda$                 $k \leftarrow \{0,1\}^\lambda$
  $v := T[p]$
  for $i = 0 + 1$ to $in$:                      $v := F(k,x)$
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$                                 return $F(k,x)$
```

When fixing $D = 0$, we always have $p = \epsilon$, so the only entry of T that is accessed is $T[\epsilon]$. Renaming $T[\epsilon]$ to k, we see that $\mathcal{L}_{\text{hyb-1}}[0] \equiv \mathcal{L}^F_{\text{prf-real}}$. The only difference is that the PRF key k is sampled at the very last minute instead of when the library is initialized.

```
$\mathcal{L}_{\text{hyb-1}}[in]$

$T :=$ empty assoc. array

QUERY($x$):
  $p :=$ first $in$ bits of $x$                $p = x$
  if $T[p]$ undefined:                       if $T[x]$ undefined:
    $T[p] \leftarrow \{0,1\}^\lambda$                 $T[x] \leftarrow \{0,1\}^\lambda$
  $v := T[p]$
  for $i = in + 1$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$             // unreachable
    if $x_i = 1$ then $v := G_R(v)$
  return $v$                                 return $T[x]$
```

When fixing $D = in$, we always have $p = x$ and the body of the for-loop is unreachable. In that case, we can see that $\mathcal{L}_{\text{hyb-1}}[in] \equiv \mathcal{L}^F_{\text{prf-rand}}$.

So we can see that the value of D in $\mathcal{L}_{\text{hyb-1}}$ is a way of "smoothly interpolating" between the two $\mathcal{L}_{\text{prf-}\star}$ endpoints of the hybrid sequence. If we can show that $\mathcal{L}_{\text{hyb-1}}[d] \approx \mathcal{L}_{\text{hyb-1}}[d+1]$ for all d, then we will complete the proof.

To show the rest of the proof we introduce two more hybrid libraries:

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

```
$T$ := empty assoc. array

QUERY($x$):
  $p$ := first $D$ bits of $x$
  if $T[p]$ undefined:
    $T[p] \leftarrow \{0,1\}^\lambda$
    $T[p\|0] := G_L(T[p])$
    $T[p\|1] := G_R(T[p])$
  $p'$ := first $D + 1$ bits of $x$
  $v := T[p']$
  for $i = D + 2$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$
```

For any value d, the libraries $\mathcal{L}_{\textsf{hyb-1}}[d]$ and $\mathcal{L}_{\textsf{hyb-2}}[d]$ have identical behavior. The only difference is that whenever $T[p]$ is sampled at level d of the tree, $\mathcal{L}_{\textsf{hyb-2}}$ now *eagerly* populates its two children $T[p\|0]$ and $T[p\|1]$ at level $d + 1$. We have more or less unrolled the first iteration of the for-loop, although we are now also computing values for both "sibling" nodes $p\|0$ and $p\|1$, not just the one along the path to x.

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

```
$T$ := empty assoc. array

QUERY($x$):
  $p$ := first $D$ bits of $x$
  if $T[p]$ undefined:
    $T[p\|0] \leftarrow \{0,1\}^\lambda$
    $T[p\|1] \leftarrow \{0,1\}^\lambda$
  $p'$ := first $D + 1$ bits of $x$
  $v := T[p']$
  for $i = D + 2$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$
```

For any value d, we claim that $\mathcal{L}_{\textsf{hyb-2}}[d] \approx \mathcal{L}_{\textsf{hyb-3}}[d]$. The difference between these libraries is in how $T[p\|0]$ and $T[p\|1]$ are computed. In $\mathcal{L}_{\textsf{hyb-3}}$, they are derived from applying a PRG to $T[p]$, a value that is never used elsewhere. In $\mathcal{L}_{\textsf{hyb-3}}$, they are chosen uniformly. We do not show all the intermediate steps here, but by factoring out the 3 lines in the if-statement of $\mathcal{L}_{\textsf{hyb-2}}$, we can apply the PRG security of G to obtain $\mathcal{L}_{\textsf{hyb-3}}$.

Finally, we claim that $\mathcal{L}_{\textsf{hyb-3}}[d] \equiv \mathcal{L}_{\textsf{hyb-1}}[d + 1]$ for any value d. Indeed, both libraries assign values to T via $T[v] \leftarrow \{0,1\}^\lambda$ for v that are $d + 1$ bits long. The $\mathcal{L}_{\textsf{hyb-1}}$ library assigns these values lazily at the last moment, while $\mathcal{L}_{\textsf{hyb-3}}$ also assigns them whenever a "sibling" node is requested.

Putting all of these observations together, we have:

$$\begin{aligned}
\mathcal{L}^F_{\textsf{prf-real}} &\equiv \mathcal{L}_{\textsf{hyb-1}}[0] \equiv \mathcal{L}_{\textsf{hyb-2}}[0] \approx \mathcal{L}_{\textsf{hyb-3}}[0] \\
&\equiv \mathcal{L}_{\textsf{hyb-1}}[1] \equiv \mathcal{L}_{\textsf{hyb-2}}[1] \approx \mathcal{L}_{\textsf{hyb-3}}[1] \\
&\vdots \\
&\equiv \mathcal{L}_{\textsf{hyb-1}}[in-1] \equiv \mathcal{L}_{\textsf{hyb-2}}[in-1] \approx \mathcal{L}_{\textsf{hyb-3}}[in-1] \\
&\equiv \mathcal{L}_{\textsf{hyb-1}}[in] \equiv \mathcal{L}^F_{\textsf{prf-rand}}
\end{aligned}$$

Hence, F is a secure PRF. ■

Exercises

6.1. In this problem, you will show that it is hard to determine the key of a PRF by querying the PRF.

Let F be a candidate PRF, and suppose there exists a program $\mathcal{A}$ such that:

$$\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-real}} \text{ outputs } k] \text{ is non-negligible.}$$

In the above expression, k refers to the private variable within $\mathcal{L}_{\text{prf-real}}$.

Prove that if such an $\mathcal{A}$ exists, then F is not a secure PRF. Use $\mathcal{A}$ to construct a distinguisher that violates the PRF security definition.

6.2. Let F be a secure PRF, and let $m \in \{0,1\}^{out}$ be a (known) fixed string. Define:

$$F_m(k,x) = F(k,x) \oplus m.$$

Prove that for every m, F_m is a secure PRF.

6.3. Let F be a secure PRF with λ-bit outputs, and let G be a PRG with stretch ℓ. Define

$$F'(k,r) = G(F(k,r)).$$

So F' has outputs of length $\lambda + \ell$. Prove that F' is a secure PRF.

6.4. Let F be a secure PRF with $in = 2\lambda$, and let G be a length-doubling PRG. Define

$$F'(k,x) = F(k,G(x)).$$

We will see that F' is not necessarily a PRF.

(a) Prove that if G is injective then F' is a secure PRF. *Hint:* you should not even need to use the fact that G is a PRG.

★ (b) Exercise 5.10(b) constructs a secure length-doubling PRG that ignores half of its input. Show that F' is insecure when instantiated with such a PRG. Give a distinguisher and compute its advantage.

Note: You are not attacking the PRF security of F, nor the PRG security of G. You are attacking the invalid way in which they have been combined.

6.5. Let F be a secure PRF, and let $m \in \{0,1\}^{in}$ be a fixed (therefore known to the adversary) string. Define the new function

$$F_m(k,x) = F(k,x) \oplus F(k,m).$$

Show that F_m is **not** a secure PRF. Describe a distinguisher and compute its advantage.

6.6. Let F be a secure PRF. Let $\overline{x}$ denote the bitwise complement of the string x. Define the new function:

$$F'(k,x) = F(k,x) \| F(k,\overline{x}).$$

Show that F' is **not** a secure PRF. Describe a distinguisher and compute its advantage.

6.7. Suppose F is a secure PRF. Define the following function F' as:

$$F'(k,x\|x') = F(k,x) \oplus F(k,x \oplus x').$$

Here, x and x' are each in bits long, where in is the input length of F. Show that F' is **not** a secure PRF. Describe a distinguisher and compute its advantage.

6.8. Define a PRF F whose key k we write as $(k_1,\ldots,k_{in})$, where each k_i is a string of length out. Then F is defined as:

$$F(k,x) = \bigoplus_{i \mid x_i=1} k_i.$$

Show that F is **not** a secure PRF. Describe a distinguisher and compute its advantage.

6.9. Define a PRF F whose key k is an $in \times 2$ array of out-bit strings, whose entries we refer to as $k[i,b]$. Then F is defined as:

$$F(k,x) = \bigoplus_{i=1}^{in} k[i,x_i].$$

Show that F is **not** a secure PRF. Describe a distinguisher and compute its advantage.

7 Pseudorandom Permutations

A pseudorandom function behaves like a randomly chosen function. In particular, a randomly chosen function need not have a mathematical inverse. But very often in cryptography it is convenient to have cryptographically interesting algorithms which can be efficiently inverted.

A **pseudorandom permutation (PRP)** is essentially a PRF that is invertible (when given the key). Since the PRP is known to be a permutation, we do not require it to be indistinguishable from a randomly chosen function. Rather, we require it to be indistinguishable from a randomly chosen *permutation.*

7.1 Definitions

Definition 7.1 (PRP syntax)

Let $F, F^{-1} : \{\texttt{0},\texttt{1}\}^\lambda \times \{\texttt{0},\texttt{1}\}^{blen} \to \{\texttt{0},\texttt{1}\}^{blen}$ be deterministic and efficiently computable functions. Then F is a ***pseudorandom permutation (PRP)*** *if for all keys $k \in \{\texttt{0},\texttt{1}\}^\lambda$, and for all $x \in \{\texttt{0},\texttt{1}\}^{blen}$, we have:*

$$F^{-1}(k, F(k, x)) = x.$$

That is, if $F^{-1}(k, \cdot)$ is the inverse of $F(k, \cdot)$. In particular, this implies that $F(k, \cdot)$ is a permutation on $\{\texttt{0},\texttt{1}\}^{blen}$ for every key k. We refer to blen as the ***blocklength*** *of F and any element of $\{\texttt{0},\texttt{1}\}^{blen}$ as a* ***block****.*

Outside of the world of academic cryptography, pseudorandom permutations are typically called **block ciphers**. We will use both terms interchangeably.

As mentioned above, our security definition will require a PRP to be indistinguishable from a randomly chosen *permutation* on $\{\texttt{0},\texttt{1}\}^{blen}$, rather than a randomly chosen *function* from $\{\texttt{0},\texttt{1}\}^{blen}$ to itself. In practical terms, we must modify the libraries that define PRF security.

As with PRFs, we could consider sampling a permutation on $\{\texttt{0},\texttt{1}\}^{blen}$ uniformly at random, but this requires exponential time when *blen* is large. Instead, we will sample the truth table of a random permutation on-the-fly, and store the mappings in an associative array T in case a query is repeated. This is the same approach taken in our PRF definition.

However, we must ensure that we always sample truth table values that are consistent with a permutation. It suffices to ensure that the function is *injective.* Injectivity is guaranteed by making sure that no two inputs map to the same output. Concretely, when we sample the value of the function at input x, we choose it uniformly among the *set of values not used as function outputs so far.* This ensures that the library's behavior is consistent with a random permutation. The formal details are below:

Definition 7.2 (PRP security)

Let $F : \{\texttt{0},\texttt{1}\}^\lambda \times \{\texttt{0},\texttt{1}\}^{blen} \to \{\texttt{0},\texttt{1}\}^{blen}$ be a deterministic function. For an associative array T, define:

$$\text{range}(T) \overset{\text{def}}{=} \{v \mid \exists x : T[x] = v\}.$$

We say that F is a ***secure PRP*** *if $\mathcal{L}^F_{\textsf{prp-real}} \approx \mathcal{L}^F_{\textsf{prp-rand}}$, where:*

```
$\mathcal{L}^F_{\textsf{prp-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{blen}$):
  return $F(k,x)$
```

```
$\mathcal{L}^F_{\textsf{prp-rand}}$
$T :=$ empty assoc. array

QUERY($x \in \{0,1\}^{blen}$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{blen} \setminus \textsf{range}(T)$
  return $T[x]$
```

The changes from the PRF definition are highlighted in yellow. In particular, the $\mathcal{L}_{\textsf{prp-real}}$ and $\mathcal{L}_{\textsf{prf-real}}$ libraries are identical.

7.2 Switching Lemma

We have modified $\mathcal{L}_{\textsf{prf-rand}}$ to provide access to a random permutation rather than a random function. How significant is the change that we made?

Consider the $\mathcal{L}_{\textsf{prf-rand}}$ library implementing a function whose domain is $\{0,1\}^\lambda$. This domain is so large that a calling program can see only a *negligible fraction* of the function's behavior. In that case, it seems unlikely that the calling program could tell the difference between a random *permutation* and a random (arbitrary) *function.*

More formally, we expect that $\mathcal{L}_{\textsf{prf-rand}}$ (which realizes a random function) and $\mathcal{L}_{\textsf{prp-rand}}$ (which realizes a random permutation) will be indistinguishable, when the parameters are such that their interfaces are the same ($blen = in = out = \lambda$). This is indeed true, as the following lemma demonstrates.

Lemma 7.3 (PRP switching) *Let $\mathcal{L}_{\textsf{prf-rand}}$ and $\mathcal{L}_{\textsf{prp-rand}}$ be defined as in Definitions 6.1 & 7.2, with parameters $in = out = blen = \lambda$. Then $\mathcal{L}_{\textsf{prf-rand}} \approx \mathcal{L}_{\textsf{prp-rand}}$.*

Proof Recall the replacement-sampling lemma, Lemma 4.10, which showed that the following libraries are indistinguishable:

```
$\mathcal{L}_{\textsf{samp-L}}$

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$
```

```
$\mathcal{L}_{\textsf{samp-R}}$
$R := \emptyset$

SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

$\mathcal{L}_{\text{samp-L}}$ samples values with replacement, and $\mathcal{L}_{\text{samp-R}}$ samples values without replacement. Now consider the following library $\mathcal{L}^*$:

```
$\mathcal{L}^*$
$T$ := empty assoc. array

QUERY($x \in \{0,1\}^\lambda$):
  if $T[x]$ undefined:
    $T[x] \leftarrow$ SAMP()
  return $T[x]$
```

When we link $\mathcal{L}^* \diamond \mathcal{L}_{\text{samp-L}}$ we obtain $\mathcal{L}_{\text{prf-rand}}$ since the values in $T[x]$ are sampled uniformly. When we link $\mathcal{L}^* \diamond \mathcal{L}_{\text{samp-R}}$ we obtain $\mathcal{L}_{\text{prp-rand}}$ since the values in $T[x]$ are sampled uniformly subject to having no repeats. Then from Lemma 4.10, we have:

$$\mathcal{L}_{\text{prf-rand}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{samp-L}} \approx \mathcal{L}^* \diamond \mathcal{L}_{\text{samp-R}} \equiv \mathcal{L}_{\text{prp-rand}},$$

which completes the proof. ■

Discussion

- It is possible to define PRFs/PRPs and random functions over domains of any size. However, the switching lemma applies only when the domains are sufficiently large — in this case, at least λ bits long. This comes from the fact that $\mathcal{L}_{\text{samp-L}}$ and $\mathcal{L}_{\text{samp-R}}$ in the proof are indistinguishable only when dealing with long (length-λ) strings (look at the proof of Lemma 4.10 to recall why).

  Exercise 7.3 asks you to show that a random permutation over *small domains* can be distinguished from a random function.

- The switching lemma refers only to the idealized $\mathcal{L}_{\text{prf-rand}}$ and $\mathcal{L}_{\text{prp-rand}}$ libraries. But a *secure* PRF has behavior that is indistinguishable to that of $\mathcal{L}_{\text{prf-rand}}$, and similarly a secure PRP to that of $\mathcal{L}_{\text{prp-rand}}$. That means that a secure PRF with $in = out = \lambda$ will have outputs that are indistinguishable from those of a secure PRP with $blen = \lambda$.

7.3 Feistel Ciphers

A PRF takes an input x and "scrambles" it to give a pseudorandom output. A PRP asks for more: there should be a way to "unscramble" the result and recover the original x. This seems much harder! Nevertheless, in this section we will see an elegant and simple way to transform a (not necessarily invertible) PRF into a PRP!

Definition 7.4 (Feistel transform)

*Let $f : \{0,1\}^n \to \{0,1\}^n$ be any function. The **Feistel transform of** f is the function* $\text{FSTL}_f : \{0,1\}^{2n} \to \{0,1\}^{2n}$ *defined by:*

$$\text{FSTL}_f(L,R) = (R, f(R) \oplus L).$$

We treat the $2n$-bit input and output of FSTL_f as a pair of n-bit inputs; thus L and R each have n bits.

Surprisingly, even though f need not be invertible, the Feistel transform of f is *always* invertible! See for yourself. Define $\textsc{fstl}_f^{-1}(X,Y) = (Y \oplus f(X), X)$. Then,

$$\begin{aligned}\textsc{fstl}_f^{-1}(\textsc{fstl}_f(L,R)) &= \textsc{fstl}_f^{-1}(R, f(R) \oplus L) \\ &= \big((f(R) \oplus L) \oplus f(R), R\big) \\ &= (L,R).\end{aligned}$$

In fact, $\textsc{fstl}_f$ and its inverse are essentially mirror images of each other: see Figures 7.1 and 7.2. The similarity of $\textsc{fstl}_f$ and $\textsc{fstl}_f^{-1}$ makes hardware and software implementations much simpler, as many components can be reused.

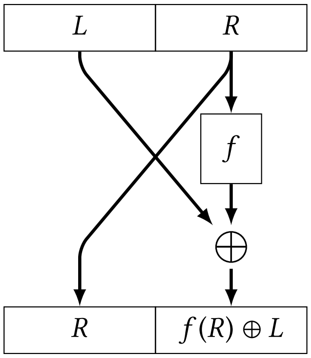


Figure 7.1: *Feistel transform of f.*

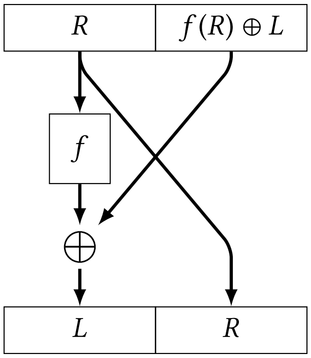


Figure 7.2: *Inverse Feistel transform of f.*

Feistel Ciphers

A **Feistel cipher** is a block cipher built by composing several Feistel transforms. For example, let $\circ$ denote composition of functions, so $f \circ g$ denotes the function $x \mapsto f(g(x))$. Then

$$\text{FSTL}_{f_3} \circ \text{FSTL}_{f_2} \circ \text{FSTL}_{f_1}$$

is a 3-round Feistel cipher whose **round functions** are f_1, f_2, and f_3. Its inverse would then be:

$$\text{FSTL}_{f_1}^{-1} \circ \text{FSTL}_{f_2}^{-1} \circ \text{FSTL}_{f_3}^{-1}.$$

Note how the round functions are reversed.

It is useful for the round functions of a Feistel cipher to be different. But rather than using totally unrelated functions, a typical Feistel cipher uses a single **keyed round function**; that is, the round function is the same for each round, but it takes an extra argument (a **round key**) which is different in each round.

For example, $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$ has the syntax of a keyed round function, where the first argument of F is the round key. For each round key k, the function $F(k,\cdot)$ maps $\{0,1\}^n$ to itself, so $F(k,\cdot)$ is a suitable round function. The following is a 3-round keyed Feistel cipher with (keyed) round function F (also Figures 7.3 & 7.4):

$$\text{FSTL}_{F(k_3,\cdot)} \circ \text{FSTL}_{F(k_2,\cdot)} \circ \text{FSTL}_{F(k_1,\cdot)}.$$

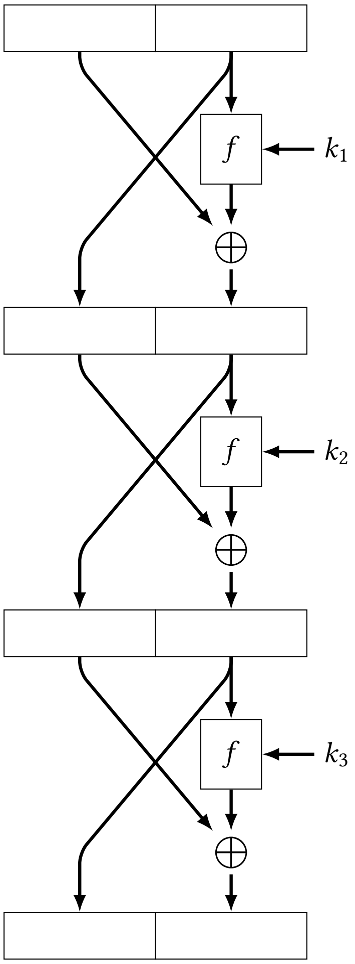


Figure 7.3: *A typical 3-round Feistel cipher with keyed round function. The key schedule is* (k_1, k_2, k_3).

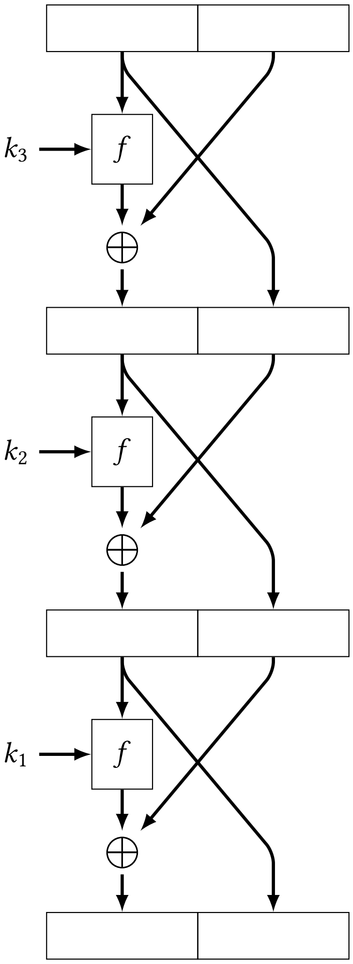


Figure 7.4: *The inverse of the Feistel cipher in Figure 7.3. Note that the key schedule is reversed.*

The sequence of keys k_1, k_2, k_3 is called the **key schedule** of the cipher. To invert this Feistel cipher, we take its mirror image and reverse the key schedule. Below is a procedural way to compute and invert an r-round keyed Feistel cipher C, with key schedule $k_1, \ldots, k_r$ and keyed round function F:

```
$C\big((k_1, \ldots, k_r), (L, R)\big)$:
  $V_{-1} := L$; $V_0 := R$
  for $i = 1$ to $r$:
    $V_i := F(k_i, V_{i-1}) \oplus V_{i-2}$
  return $(V_{r-1}, V_r)$

$C^{-1}\big((k_1, \ldots, k_r), (L, R)\big)$:
  $V_{r-1} := L$; $V_r := R$
  for $i = r$ downto 1:
    $V_{i-2} := F(k_i, V_{i-1}) \oplus V_i$
  return $(V_{-1}, V_0)$
```

Interestingly, when F is a secure PRF and k_1, k_2, k_3 are chosen uniformly and independently, such a 3-round Feistel cipher is a pseudorandom permutation (when the blocklength is large enough). In the exercises, you will show that 1 and 2 rounds do not lead to a secure PRP.

Theorem 7.5 (Luby-Rackoff) *Let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a secure PRF with in = out = λ, and define $F^* : \{0,1\}^{3\lambda} \times \{0,1\}^{2\lambda} \to \{0,1\}^{2\lambda}$ as:*

$$F^*\Big((k_1,k_2,k_3),(L,R)\Big) \overset{\text{def}}{=} \mathit{FSTL}_{F(k_3,\cdot)}\big(\mathit{FSTL}_{F(k_2,\cdot)}\big(\mathit{FSTL}_{F(k_1,\cdot)}(L,R)\big)\big).$$

Then F^* is a secure PRP.

to-do *The proof is just slightly trickier than what I would prefer including in the notes. I will think more about finding helpful indistinguishability lemmas that make the proof more manageable.*

The PRP in question has the following form:

```
$F^*\Big((k_1,k_2,k_3),(L,R)\Big)$:
  $X := F(k_1,R) \oplus L$
  $Y := F(k_2,X) \oplus R$
  $Z := F(k_3,Y) \oplus X$
  return $(Y,Z)$
```

The idea behind the proof is to show that F^* is a secure PRF. Then, because its blocklength is large enough, the switching lemma shows that it is also a secure PRP. The main idea is:

- *We can apply the PRF security to the 3 different PRF instances. The result is $T_1[R]$, $T_2[X]$ and $T_3[Y]$ in place of the PRF calls, where T_i are the truth tables of distinct random functions (sampled on the fly).*
- *Now it suffices to show that the calling program will never find distinct (L,R) pairs that lead to duplicate X or Y values. As long as that's the case, $T_2[X]$ and $T_3[Y]$ will always be uniformly random for distinct inputs to F^*.*
- *It is possible to show that the calling program cannot find (L,R) pairs that lead to the same X value. This is a quite subtle part, since it depends on X not being visible to the calling program.*
- *Given that X is distinct for all distinct inputs, it can then be shown that Y is distinct as well.*

Instantiations

The Data Encryption Standard (DES) was the most commonly used block cipher, until it was phased out in favor of the newer Advanced Encryption Standard (AES). DES was designed as a standard Feistel network with 16 rounds. Its blocklength is 64 bits, so the input/output length of the round function is 32 bits.

While we have theoretical results (like the one above) about the security of Feistel ciphers with *independent* round keys, Feistel ciphers in practice typically do not use independent keys. Rather, their round keys are derived in some way from a master key. DES follows this pattern, with each 48-bit round key being derived from a 56-bit **master key**. The reason for this discrepancy stems from the fact that an r-round Feistel network with *independent* keys will have an $r\lambda$-bit master key but give only λ bits of security. In practice, it is desirable to use a small λ-bit master key but derive from it many round keys in a way that "mixes" the entire master key into all of the rounds in some way.

7.4 Strong Pseudorandom Permutations

Although every PRP has an inverse, it is not necessarily true that a PRP and its inverse have comparable security properties (the exercises explore this idea further). Yet, it is sometimes very convenient to argue about pseudorandom behavior of F^{-1}. For instance, F^{-1} is used to decrypt ciphertexts in many encryption schemes that we will see, and unpredictability in F^{-1} is useful when an adversary generates invalid ciphertexts.

It is possible to insist that both F and F^{-1} are PRPs, which would mean that $\mathcal{L}_{\textsf{prp-real}}$ and $\mathcal{L}_{\textsf{prp-rand}}$ are indistinguishable when instantiated with F and when instantiated with F^{-1}. In doing so, we have some interactions where a distinguisher queries F and other interactions where a distinguisher queries F^{-1}. A stronger requirement would be to allow the distinguisher to query both F and F^{-1} in a *single* interaction. If a PRP is indistinguishable from a random permutation under that setting, then we say it is a **strong** PRP (SPRP). In our previous example, such security would be useful to model an adversary who can see encryptions of plaintexts (where encryption involves evaluating F) and who can also ask for decryptions of invalid ciphertexts (where decryption involves evaluating F^{-1}).

The security definition for an SPRP then considers two libraries. The common interface of these two libraries provides *two* subroutines: one for forward queries and one for reverse queries. In the $\mathcal{L}_{\textsf{sprp-real}}$ library, these subroutines are implemented by calling the PRP or its inverse accordingly. In the $\mathcal{L}_{\textsf{sprp-rand}}$ library, we would like to emulate the behavior of a randomly chosen permutation that can be queried in both directions. For convenience, we therefore maintain two associative arrays to maintain the forward and backward mappings. The formal details are below:

Definition 7.6 (SPRP security) *Let $F : \{0,1\}^{\lambda} \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ be a deterministic function. We say that F is a* ***secure***

***strong pseudorandom permutation (SPRP)** if $\mathcal{L}^F_{\text{sprp-real}} \approx \mathcal{L}^F_{\text{sprp-rand}}$, where:*

```
$\mathcal{L}^F_{\text{sprp-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{blen}$):
  return $F(k,x)$

INVQUERY($y \in \{0,1\}^{blen}$):
  return $F^{-1}(k,y)$
```

```
$\mathcal{L}^F_{\text{sprp-real}}$
$T, T^{-1}$ := empty assoc. arrays

QUERY($x \in \{0,1\}^{blen}$):
  if $T[x]$ undefined:
    $y \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
    $T[x] := y$; $T^{-1}[y] := x$
  return $T[x]$

INVQUERY($y \in \{0,1\}^{blen}$):
  if $T^{-1}[y]$ undefined:
    $x \leftarrow \{0,1\}^{blen} \setminus \text{range}(T^{-1})$
    $T^{-1}[y] := x$; $T[x] := y$
  return $T^{-1}[y]$
```

Earlier we showed that a 3-round Feistel network can be used to construct a PRP from a PRF. The resulting PRP is *not* a *strong* PRP. However, a 4-round Feistel network *is* a strong PRP! We present the following theorem without proof:

Theorem 7.7 (Luby-Rackoff) *Let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a secure PRF with in = out = λ, and define $F^* : \{0,1\}^{4\lambda} \times \{0,1\}^{2\lambda} \to \{0,1\}^{2\lambda}$ as:*

$$F^*\Big((k_1,k_2,k_3,k_4),(L,R)\Big) \overset{\text{def}}{=} \mathit{FSTL}_{F(k_4,\cdot)}\big(\mathit{FSTL}_{F(k_3,\cdot)}\big(\mathit{FSTL}_{F(k_2,\cdot)}\big(\mathit{FSTL}_{F(k_1,\cdot)}(L,R)\big)\big)\big).$$

*Then F^* is a secure **strong** PRP.*

Exercises

7.1. Let F be a secure PRP with blocklength $blen = 128$. Then for each k, the function $F(k,\cdot)$ is a permutation on $\{0,1\}^{128}$. Suppose I choose a permutation on $\{0,1\}^{128}$ uniformly at random. What is the probability that the permutation I chose agrees with a permutation of the form $F(k,\cdot)$? Compute the probability as an actual number — is it a reasonable probability or a tiny one?

7.2. Suppose $R : \{0,1\}^n \to \{0,1\}^n$ is chosen uniformly among all such functions. What is the probability that there exists an $x \in \{0,1\}^n$ such that $R(x) = x$?

Hint: First find the probability that $R(x) \neq x$ for all x. Simplify your answer using the approximation $(1-y) \approx e^{-y}$.

7.3. In this problem, you will show that the PRP switching lemma holds only for large domains. Let $\mathcal{L}_{\text{prf-rand}}$ and $\mathcal{L}_{\text{prp-rand}}$ be as in Lemma 7.3. Choose any small value of $blen = in = out$ that you like, and show that $\mathcal{L}_{\text{prf-rand}} \not\approx \mathcal{L}_{\text{prp-rand}}$ with those parameters. Describe a distinguisher and compute its advantage. *Hint:* remember that the distinguisher needs to run in polynomial time in λ, but not necessarily polynomial in *blen*.

7.4. Let $f : \{0,1\}^{in} \to \{0,1\}^{out}$ be a (not necessarily invertible) function. The Feistel transform described in this section works only when $in = out$.

Describe a modification of the Feistel transform that works even when the round function satisfies $in \neq out$. The result should be an invertible with input/output length $in + out$. Be sure to show that your proposed transform is invertible!

7.5. Show that any function F that is a 1-round keyed Feistel cipher **cannot** be a secure PRP. That is, construct a distinguisher to demonstrate that $\mathcal{L}^F_{\text{prp-real}} \not\approx \mathcal{L}^F_{\text{prp-rand}}$, knowing only that F is a 1-round Feistel cipher. In particular, the purpose is to attack the Feistel transform and not the round function, so your attack should work no matter what the round function is.

7.6. Show that any function F that is a 2-round keyed Feistel cipher **cannot** be a secure PRP. As above, your distinguisher should work without knowing what the round functions are, and the attack should work with different (independent) round functions for the 2 rounds.

Hint: Make two queries, where the second query depends on the answer to the first. With carefully chosen queries, it is possible to identify a property that is always satisfied by a 2-round Feistel network but that is rarely satisfied by a random function.

7.7. Show that any function F that is a 3-round keyed Feistel cipher **cannot** be a secure *strong* PRP. As above, your distinguisher should work without knowing what the round functions are, and the attack should work with different (independent) round functions.

7.8. In this problem you will show that PRPs are hard to invert without the key (if the blocklength is large enough). Let F be a candidate PRP with blocklength $blen \geqslant \lambda$. Suppose there is a program $\mathcal{A}$ where:

$$\Pr_{y \leftarrow \{0,1\}^{blen}} \left[\mathcal{A}(y) \diamond \mathcal{L}^F_{\text{prf-real}} \text{ outputs } F^{-1}(k,y) \right] \text{ is non-negligible.}$$

In the above expression, $\mathcal{A}(y)$ indicates that $\mathcal{A}$ receives y as an input, and k refers to the private variable within $\mathcal{L}_{\text{prf-real}}$. So, when given the ability to evaluate F in the forward direction only (via $\mathcal{L}_{\text{prf-real}}$), $\mathcal{A}$ can invert a uniformly chosen block y.

Prove that if such an $\mathcal{A}$ exists, then F is not a secure PRP. Use $\mathcal{A}$ to construct a distinguisher that violates the PRP security definition. Where do you use the fact that $blen \geqslant \lambda$? How do you deal with the fact that $\mathcal{A}$ may give the wrong answer with high probability?

★ 7.9. Construct a PRP F such that F^{-1} is **not** a PRP.

Hint: Take an existing PRP and modify it so that $F(k,k) = 0^{blen}$. Argue that the result is still a PRP, and show an attack against the PRP security definition applied to F^{-1}.

8 Security against Chosen Plaintext Attacks

We've already seen a definition that captures security of encryption when an adversary is allowed to see just one ciphertext encrypted under the key. Clearly a more useful scheme would guarantee security against an adversary who sees an *unlimited* number of ciphertexts.

Fortunately we have arranged things so that we get the "correct" security definition when we modify the earlier definition in a natural way. We simply let the libraries choose a (secret) key for encryption and allow the calling program to request any number of plaintexts to be encrypted under that key. More formally:

Definition 8.1 (CPA security) *Let Σ be an encryption scheme. We say that Σ has* ***security against chosen-plaintext attacks (CPA security)*** *if $\mathcal{L}^{\Sigma}_{\text{cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{cpa-R}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{cpa-L}}$
$k \leftarrow \Sigma.\text{KeyGen}$
CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, m_L)$
  return $c$
```

```
$\mathcal{L}^{\Sigma}_{\text{cpa-R}}$
$k \leftarrow \Sigma.\text{KeyGen}$
CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, m_R)$
  return $c$
```

CPA security is often called "IND-CPA" security, meaning "indistinguishability of ciphertexts under chosen-plaintext attack."

8.1 Implications of CPA Security

CPA security is a deceptively simple definition. In this section we will discuss some of the implications of the definition.

CPA Security Protects All Partial Information

A reasonable (informal) security definition for encryption is that "the ciphertext should not leak any partial information about the plaintext." For example, an attacker should not be able to guess, say, the least significant bit or the language of the plaintext, or whether the plaintext is written in ALL-CAPS, etc.

Looking at the CPA security definition, it may not be immediately clear that it ensures protection against partial information leakage. Luckily it does, as we will see from the following illustration.

Let's consider the simple example of an adversary who wishes to guess the least significant bit of a plaintext. Of course, this is already possible with probability 1/2, without even looking at the plaintext. If an adversary can guess with probability signficantly better than 1/2, however, we would accuse the encryption scheme of somehow leaking the least significant bit of the plaintext.

Consider an encryption scheme Σ, and let $\mathsf{lsb}(m)$ denote the least significant bit of a string m. Suppose that there exists an efficient algorithm $\mathcal{B}$ that can guess $\mathsf{lsb}(m)$ given only an encryption of m. That is:

$$\forall m \in \Sigma.\mathcal{M} : \Pr_{k \leftarrow \Sigma.\mathcal{K}}[\mathcal{B}(\Sigma.\mathsf{Enc}(k,m)) = \mathsf{lsb}(m)] \geqslant 1/2 + \epsilon.$$

We will show that ϵ must be negligible if Σ has CPA security. That is, $\mathcal{B}$ cannot guess $\mathsf{lsb}(m)$ much better than chance.

Consider the following distinguisher:

```
$\mathcal{A}$
choose arbitrary $m_0 \in \Sigma.\mathcal{M}$ with $\mathsf{lsb}(m_0) = 0$
choose arbitrary $m_1 \in \Sigma.\mathcal{M}$ with $\mathsf{lsb}(m_1) = 1$
$c := \textsc{query}(m_0, m_1)$
return $\mathcal{B}(c)$
```

.

What happens when this distinguisher is linked to the libraries that define CPA security?

- In $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-L}}$, the ciphertext c encodes plaintext m_0, whose least significant bit is 0. So the probability that $\mathcal{B}$ will output 1 is at most $1/2 - \epsilon$.

- In $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-R}}$, the ciphertext c encodes plaintext m_1, whose least significant bit is 1. So the probability that $\mathcal{B}$ will output 1 is at least $1/2 + \epsilon$.

This distinguisher is efficient and its advantage is at least $|(1/2 + \epsilon) - (1/2 - \epsilon)| = 2\epsilon$. But if Σ is CPA-secure, this advantage must be negligible. Hence ϵ is negligible.

Another way to look at this example is: if the ciphertexts of an encryption scheme leak some partial information about plaintexts, then it is possible to break CPA security. Simply challenge the CPA libraries with two plaintexts whose partial information is different. Then detecting the partial information will tell you which library you are linked to.

Hopefully you can see that there was nothing special about least-significant bits of plaintexts here. Any partial information can be used for the attack.

We could also imagine expanding the capabilities of $\mathcal{B}$. Suppose $\mathcal{B}$ could determine some partial information about a ciphertext by also asking for chosen plaintexts to be encrypted under the same key. Since this extra capability can be done within the CPA libraries, we can still use such a $\mathcal{B}$ to break CPA security if $\mathcal{B}$ is successful.

Impossibility of Deterministic Encryption

We will now explore a not-so-obvious side effect of CPA security, which was first pointed out by Goldwasser & Micali[1]: CPA-secure encryption **cannot be deterministic.** By that,

[1] Shafi Goldwasser & Sivlio Micali: *Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information.* 1982. This paper along with another one led to a Turing Award for the authors.

we mean that calling $\mathsf{Enc}(k,m)$ twice with the same key and same plaintext *must* lead to different outputs, if the scheme is CPA secure.

To see why, consider the following distinguisher $\mathcal{A}$:

```
$\mathcal{A}$
arbitrarily choose distinct plaintexts $x, y \in \mathcal{M}$
$c_1 := \textsc{query}(x,x)$
$c_2 := \textsc{query}(x,y)$
return $c_1 \overset{?}{=} c_2$
```

When executed as $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-L}}$, we see that c_1 and c_2 will be encryptions of the same plaintext x. When executed as $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-R}}$, those ciphertexts will be encryptions of different plaintexts x and y.

Now, suppose the encryption algorithm Enc is a **deterministic** function of the key and the plaintext. When linked to $\mathcal{L}_{\text{cpa-L}}$, we must have $c_1 = c_2$, so $\mathcal{A}$ will always output 1. When linked to $\mathcal{L}_{\text{cpa-R}}$, we must always have $c_1 \neq c_2$ (since otherwise correctness would be violated). Hence, we have described a distinguisher with advantage 1; the scheme is not CPA-secure.

So if deterministic encryption schemes are not secure, *what information is leaking* exactly? We can figure it out by comparing the differences between $\mathcal{A} \diamond \mathcal{L}_{\text{cpa-L}}$ and $\mathcal{A} \diamond \mathcal{L}_{\text{cpa-R}}$. The distinguisher that we constructed is able to tell whether two ciphertexts encrypt the same plaintext. In short, it is able to perform *equality tests* on encrypted data. Ciphertexts are leaking an equality predicate.

We are only now seeing this subtlety arise because this is the first time our security definition allows an adversary to see multiple ciphertexts encrypted under the same key.

This example illustrates a fundamental property of CPA security:

> *If an encryption scheme is CPA-secure, then calling the* Enc *algorithm twice with the same plaintext and key* must *result in different ciphertexts. (For if not, the attacker $\mathcal{A}$ above violates CPA security.)*

We can use this observation as a kind of sanity check. Any time an encryption scheme is proposed, you can ask whether it has a deterministic encryption algorithm. If so, then the scheme cannot be CPA-secure.

Later in this chapter we will see how to construct an encryption scheme whose encryption algorithm is *randomized* and that achieves CPA security.

8.2 Pseudorandom Ciphertexts

You may have noticed a common structure in previous security proofs for encryption (Theorem 2.8 & Claim 5.4). We start with a library in which plaintext m_L was being encrypted. Through a sequence of hybrids we arrive at a hybrid library in which ciphertexts are chosen uniformly at random. This is typically the “half-way point” of the proof, since the steps used before/after this step are mirror images of each other (and with m_R in place of m_L).

CPA security demands that encryptions of m_L are indistinguishable from encryptions of m_R. But is not important that these ciphertext distributions are anything in particular; it's only important that they are indistinguishable for all plaintexts. But in the schemes that we've seen (and indeed, in most schemes that we will see in the future), those distributions happen to look like the uniform distribution. Of course, we have a word for "a distribution that looks like the uniform distribution" — *pseudorandom.*

We can formalize what it means for an encryption scheme to have pseudorandom ciphertexts, and prove that pseudorandom ciphertexts imply CPA security. Then in the future it is enough to show that new encryption schemes have pseudorandom ciphertexts. These proofs will typically be about half as long, since won't have to give a sequence of hybrids whose second half is the "mirror image" of the first half. The idea of getting to a half-way point and then using mirror-image steps is captured once and for all in the proof that pseudorandom ciphertexts implies CPA security.

Definition 8.2 (CPA$ security) *Let Σ be an encryption scheme. We say that Σ has **pseudorandom ciphertexts in the presence of chosen-plaintext attacks (CPA$ security)** if $\mathcal{L}^{\Sigma}_{\text{cpa\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$
$k \leftarrow \Sigma.\text{KeyGen}$
CHALLENGE($m \in \Sigma.\mathcal{M}$):
    $c := \Sigma.\text{Enc}(k, m)$
    return $c$
```

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$
CHALLENGE($m \in \Sigma.\mathcal{M}$):
    $c \leftarrow \Sigma.\mathcal{C}$
    return $c$
```

This definition is also called "IND$-CPA", meaning "indistinguishable from random under chosen plaintext attacks."

Claim 8.3 *Let Σ be an encryption scheme. If Σ has CPA$ security, then it also has CPA security.*

Proof We want to prove that $\mathcal{L}^{\Sigma}_{\text{cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{cpa-R}}$, *using* the assumption that $\mathcal{L}^{\Sigma}_{\text{cpa\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$. The sequence of hybrids follows:

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

```
$\mathcal{L}^{\Sigma}_{\text{cpa-L}}$
$k \leftarrow \Sigma.\text{KeyGen}$
CHALLENGE($m_L, m_R$):
    $c := \Sigma.\text{Enc}(k, m_L)$
    return $c$
```

The starting point is $\mathcal{L}^{\Sigma}_{\text{cpa-L}}$, as expected.

```
CHALLENGE($m_L, m_R$):
    $c$ := CHALLENGE′($m_L$)
    return $c$
◇
$\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$
$k \leftarrow \Sigma$.KeyGen
CHALLENGE′($m$):
    $c := \Sigma.\mathsf{Enc}(k, m)$
    return $c$
```

It may look strange, but we have further factored out the call to Enc into a subroutine. It's no accident that the subroutine exactly matches $\mathcal{L}_{\text{cpa\$-real}}$. Since the two libraries have a slight mismatch in their interface (CHALLENGE in $\mathcal{L}_{\text{cpa-L}}$ takes two arguments while CHALLENGE′ in $\mathcal{L}_{\text{cpa\$-real}}$ takes one), the original library still "makes the choice" of which of m_L, m_R to encrypt.

```
CHALLENGE($m_L, m_R$):
    $c$ := CHALLENGE′($m_L$)
    return $c$
◇
$\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$
CHALLENGE′($m$):
    $c \leftarrow \Sigma.\mathcal{C}$
    return $c$
```

We have replaced $\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$ with $\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$. By our assumption, the change is indistinguishable.

```
CHALLENGE($m_L, m_R$):
    $c$ := CHALLENGE′( $m_R$ )
    return $c$
◇
$\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$
CHALLENGE′($m$):
    $c \leftarrow \Sigma.\mathcal{C}$
    return $c$
```

We have changed the argument being passed to CHALLENGE′. This has no effect on the library's behavior since CHALLENGE′ completely ignores its argument in these hybrids.

```
CHALLENGE($m_L, m_R$):
    $c$ := CHALLENGE′($m_R$)
    return $c$
◇
$\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$
$k \leftarrow \Sigma$.KeyGen
CHALLENGE′($m$):
    $c := \Sigma.\mathsf{Enc}(k, m)$
    return $c$
```

The mirror image of a previous step; we replace $\mathcal{L}_{\text{cpa\$-rand}}$ with $\mathcal{L}_{\text{cpa\$-real}}$.

```
$\mathcal{L}^{\Sigma}_{\text{cpa-R}}$:
$\mathcal{L}^{\Sigma}_{\text{cpa-R}}$
$k \leftarrow \Sigma$.KeyGen
CHALLENGE($m_L, m_R$):
    $c := \Sigma.\mathsf{Enc}(k, m_R)$
    return $c$
```

The $\mathcal{L}_{\text{cpa\$-real}}$ library has been inlined, and the result is $\mathcal{L}^{\Sigma}_{\text{cpa-R}}$.

The sequence of hybrids shows that $\mathcal{L}^{\Sigma}_{\text{cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{cpa-R}}$, as desired. ■

8.3 CPA-Secure Encryption from PRFs

CPA security presents a significant challenge; its goals seem difficult to reconcile. On the one hand, we need an encryption method that is randomized, so that each plaintext m is mapped to a large number of potential ciphertexts. On the other hand, the decryption method must be able to recognize all of these various ciphertexts as being encryptions of m.

Fortunately, both of these problems can be solved by an appropriate use of PRFs. Intuitively, the only way to encrypt a plaintext is to mask it with an *independent*, pseudorandom one-time pad. Both the sender and receiver must be able to derive these one-time pads from their (short) shared encryption key. Furthermore, they should be able to derive *any polynomial in* λ number of these one-time pads, in order to send any number of messages.

A PRF is then a good fit for the problem. A short PRF key can be used to derive an *exponential* amount of pseudorandom outputs ($F(k,x_1)$, $F(k,x_2)$, ...) which can be used as one-time pads. The only challenge is therefore to decide which PRF outputs to use. The most important factor is that these one-time pads should never be repeated, as we are aware of the pitfalls of reusing one-time pads.

One way to use the PRF is with a counter as its input. That is, the ith message is encrypted using $F(k,i)$ as a one-time pad. While this works, it has the downside that it *requires both sender and receiver to maintain state.*

The approach taken below is to simply choose a value r at random and use $F(k,r)$ as the one-time pad to mask the plaintext. If r is also sent as part of the ciphertext, then the receiver can also compute $F(k,r)$ and recover the plaintext. Furthermore, a value r would be repeated (and hence its mask $F(k,r)$ would be reused) only with negligible probability. It turns out that this construction does indeed achieve CPA security (more specifically, CPA\$ security):

Construction 8.4 *Let F be a secure PRF with in $= \lambda$. Define the following encryption scheme based on F:*

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

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

Enc($k,m$):
    $r \leftarrow \{0,1\}^\lambda$
    $x := F(k,r) \oplus m$
    return $(r,x)$

Dec($k,(r,x)$):
    $m := F(k,r) \oplus x$
    return $m$
```

Correctness of the scheme can be verified by inspection.

Claim 8.5 *Construction 8.4 has CPA\$ security if F is a secure PRF.*

Proof Note that we are proving that the scheme has pseudorandom ciphertexts (CPA\$ security). This implies that the scheme has standard CPA security as well. The sequence of hybrids is shown below:

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

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$
$k \leftarrow \{0,1\}^{\lambda}$

CHALLENGE($m$):
    $r \leftarrow \{0,1\}^{\lambda}$
    $x := F(k,r) \oplus m$
    return $(r,x)$
```

The starting point is $\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$. The details of Σ have been inlined.

```
CHALLENGE($m$):
    $r \leftarrow \{0,1\}^{\lambda}$
    $z :=$ QUERY($r$)
    $x := z \oplus m$
    return $(r,x)$
```

$\diamond$

```
$\mathcal{L}^{F}_{\text{prf-real}}$
$k \leftarrow \{0,1\}^{\lambda}$

QUERY($r$):
    return $F(k,r)$
```

The statements pertaining to the PRF have been factored out into a subroutine, matching $\mathcal{L}^{F}_{\text{prf-real}}$.

```
CHALLENGE($m$):
    $r \leftarrow \{0,1\}^{\lambda}$
    $z :=$ QUERY($r$)
    $x := z \oplus m$
    return $(r,x)$
```

$\diamond$

```
$\mathcal{L}^{F}_{\text{prf-rand}}$
$T :=$ empty

QUERY($r$):
    if $T[r]$ undefined:
        $T[r] \leftarrow \{0,1\}^{out}$
    return $T[r]$
```

We have replaced $\mathcal{L}^{F}_{\text{prf-real}}$ with $\mathcal{L}^{F}_{\text{prf-rand}}$. From the PRF security of F, these two hybrids are indistinguishable.

At this point in the proof, it seems like we are done. Ciphertexts have the form (r,x), where r is chosen uniformly and x is the result of encrypting the plaintext with what appears to be a one-time pad. Looking more carefuly, however, the "one-time" pad is $T[r]$ — a value that could potentially be used more than once if r is ever repeated!

Put differently, a PRF gives independently random(-looking) outputs on *distinct inputs*. But in our current hybrid there is no guarantee that PRF inputs are distinct! Our proof must explicitly contain reasoning about why PRF inputs are unlikely to be repeated. We do so by appealing to the sampling-with-replacement lemma of Lemma 4.10.

We first factor out the sampling of r values into a subroutine. The subroutine corresponds to the $\mathcal{L}_{\text{samp-L}}$ library of Lemma 4.10:

```
CHALLENGE($m$):
    $r \leftarrow$ SAMP()
    $z :=$ QUERY($r$)
    $x := z \oplus m$
    return $(r,x)$
```

$\diamond$

```
$\mathcal{L}^{F}_{\text{prf-rand}}$
$T :=$ empty

QUERY($r$):
    if $T[r]$ undefined:
        $T[r] \leftarrow \{0,1\}^{out}$
    return $T[r]$
```

$\diamond$

```
$\mathcal{L}_{\text{samp-L}}$

SAMP():
    $r \leftarrow \{0,1\}^{\lambda}$
    return $r$
```

Next, $\mathcal{L}_{\text{samp-L}}$ is replaced by $\mathcal{L}_{\text{samp-R}}$. By Lemma 4.10, the difference is indistinguishable:

```
CHALLENGE($m$):
  $r \leftarrow$ SAMP()
  $z :=$ QUERY($r$)
  $x := z \oplus m$
  return $(r, x)$
◇
$\mathcal{L}^F_{\text{prf-rand}}$
$T :=$ empty
QUERY($r$):
  if $T[r]$ undefined:
    $T[r] \leftarrow \{0,1\}^{out}$
  return $T[r]$
◇
$\mathcal{L}_{\text{samp-R}}$
$R := \emptyset$
SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

By inspection, the arguments to QUERY are *guaranteed* to never repeat in the previous hybrid, so the $\mathcal{L}_{\text{prf-rand}}$ library can be greatly simplified. In particular, the if-condition in $\mathcal{L}_{\text{prf-rand}}$ is always true. Simplifying has no effect on the library's output behavior:

```
CHALLENGE($m$):
  $r \leftarrow$ SAMP()
  $z :=$ QUERY($r$)
  $x := z \oplus m$
  return $(r, x)$
◇
QUERY($r$):
  $t \leftarrow \{0,1\}^{out}$
  return $t$
◇
$\mathcal{L}_{\text{samp-R}}$
$R := \emptyset$
SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

Now we are indeed using unique one-time pads to mask the plaintext. We are in much better shape than before. Recall that our goal is to arrive at a hybrid in which the outputs of CHALLENGE are chosen uniformly. These outputs include the value r, but now r is no longer being chosen uniformly! We must revert r back to being sampled uniformly, and then it is a simple matter to argue that the outputs of CHALLENGE are generated uniformly.

```
CHALLENGE($m$):
  $r \leftarrow$ SAMP()
  $z :=$ QUERY($r$)
  $x := z \oplus m$
  return $(r, x)$
◇
QUERY($r$):
  $t \leftarrow \{0,1\}^{out}$
  return $t$
◇
$\mathcal{L}_{\text{samp-L}}$
SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$
```

As promised, $\mathcal{L}_{\text{samp-R}}$ has been replaced by $\mathcal{L}_{\text{samp-L}}$. The difference is indistinguishable due to Lemma 4.10.

```
CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda$
  $z \leftarrow \{0,1\}^{out}$
  $x := z \oplus m$
  return $(r, x)$
```

All of the subroutine calls have been inlined; no effect on the library's output behavior.

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

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$
CHALLENGE($m$):
  $r \leftarrow \{0,1\}^{\lambda}$
  $x \leftarrow \{0,1\}^{out}$
  return $(r, x)$
```

We have applied the one-time pad rule with respect to variables z and x. In the interest of brevity we have omitted some familiar steps (factor out, replace library, inline) that we have seen several times before. The resulting library is precisely $\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$ since it samples uniformly from $\Sigma.\mathcal{C} = \{0,1\}^{\lambda} \times \{0,1\}^{out}$.

The sequence of hybrids shows that $\mathcal{L}^{\Sigma}_{\text{cpa\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$, so Σ has pseudorandom ciphertexts. ■

Exercises

to-do *Add problems about composing CPA-secure schemes. Or perhaps these can go in the chapter about CCA security where they provide a good contrast.*

8.1. Let Σ be an encryption scheme, and suppose there is a program $\mathcal{A}$ that recovers the key from a chosen plaintext attack. More precisely, $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa}} \text{ outputs } k]$ is non-negligible, where $\mathcal{L}^{\Sigma}_{\text{cpa}}$ is defined as:

```
$\mathcal{L}^{\Sigma}_{\text{cpa}}$
$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, m)$
  return $c$
```

Prove that if such an $\mathcal{A}$ exists, then Σ does not have CPA security. Use $\mathcal{A}$ as a subroutine in a distinguisher that violates the CPA security definition.

In other words, CPA security implies that it should be hard to determine the key from seeing encryptions of chosen plaintexts.

8.2. Let Σ be an encryption scheme with CPA\$ security. Let Σ' be the encryption scheme defined by:

$$\Sigma'.\text{Enc}(k, m) = \texttt{00} \| \Sigma.\text{Enc}(k, m)$$

The decryption algorithm in Σ' simply throws away the first two bits of the ciphertext and then calls $\Sigma.\text{Dec}$.

(a) Does Σ' have CPA\$ security? Prove or disprove (if disproving, show a distinguisher and calculate its advantage).

(b) Does Σ' have CPA security? Prove or disprove (if disproving, show a distinguisher and calculate its advantage).

8.3. Suppose a user is using Construction 8.4 and an adversary observes two ciphertexts that have the same r value.

(a) What exactly does the adversary learn about the plaintexts in this case?

(b) How do you reconcile this with the fact that in the proof of Claim 8.5 there is a hybrid where r values are *never* repeated?

8.4. Let F be a secure PRP with blocklength $blen = \lambda$. Below are several encryption schemes, each with $\mathcal{K} = \mathcal{M} = \{0,1\}^\lambda$ and $\mathcal{C} = (\{0,1\}^\lambda)^2$. For each one:

- Give the corresponding Dec algorithm.
- State whether the scheme has CPA security. (Assume KeyGen samples the key uniformly from $\{0,1\}^\lambda$.) If so, then give a security proof. If not, then describe a successful adversary and compute its distinguishing bias.

(a)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $z := F(k, m) \oplus r$
    return $(r, z)$
```

(b)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $s := r \oplus m$
    $x := F(k, r)$
    return $(s, x)$
```

(c)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $x := F(k, r \oplus m)$
    return $(r, x)$
```

(d)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $x := F(k, r)$
    $y := r \oplus m$
    return $(x, y)$
```

(e)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $x := F(k, r)$
    $y := F(k, r) \oplus m$
    return $(x, y)$
```

(f)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $x := F(k, r)$
    $y := r \oplus F(k, m)$
    return $(x, y)$
```

(g)

```
Enc(k, m):
    $s_1 \gets \{0,1\}^\lambda$
    $s_2 := s_1 \oplus m$
    $x := F(k, s_1)$
    $y := F(k, s_2)$
    return $(x, y)$
```

★ (h)

```
Enc(k, m):
    $r \gets \{0,1\}^\lambda$
    $x := F(k, m \oplus r) \oplus r$
    return $(r, x)$
```

Hint: In all security proofs, use the PRP switching lemma (Lemma 7.3) since F is a PRP. Parts (b) and (c) differ by an application of Exercise 2.1.

8.5. Let Σ be an encryption scheme with plaintext space $\mathcal{M} = \{0,1\}^n$ and ciphertext space $\mathcal{C} = \{0,1\}^n$. Prove that Σ cannot have CPA security.

Conclude that direct application of a PRP to the plaintext is not a good choice for an encryption scheme.

★ 8.6. In all of the CPA-secure encryption schemes that we'll ever see, ciphertexts are at least λ bits longer than plaintexts. This problem shows that such **ciphertext expansion** is essentially unavoidable for CPA security.

Let Σ be an encryption scheme with plaintext space $\mathcal{M} = \{0,1\}^n$ and ciphertext space $C = \{0,1\}^{n+\ell}$. Show that there exists a distinguisher that distinguishes the two CPA libraries with advantage $\Omega(1/2^\ell)$.

Hint: As a warmup, consider the case where each plaintext has *exactly* 2^ℓ possible ciphertexts. However, this need not be true in general. For the general case, choose a random plaintext m and argue that with "good probability" (that you should precisely quantify) m has at most $2^{\ell+1}$ possible ciphertexts.

8.7. Show that an encryption scheme Σ has CPA security if and only if the following two libraries are indistinguishable:

```
$\mathcal{L}^{\Sigma}_{\text{left}}$
$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\text{Enc}(k, m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{right}}$
$k \leftarrow \Sigma.\text{KeyGen}$

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

★ 8.8. Let Σ_1 and Σ_2 be encryption schemes with $\Sigma_1.\mathcal{M} = \Sigma_2.\mathcal{M} = \{0,1\}^n$.

Consider the following approach for encrypting plaintext $m \in \{0,1\}^n$: First, secret-share m into s_1 and s_2, as in the 2-out-of-2 secret-sharing scheme in Construction 3.5. Then encrypt s_1 under Σ_1 and s_2 under Σ_2.

(a) Formally describe this encryption method, including the key generation and decryption procedure.

(b) Prove that the scheme has CPA security if **at least one of** $\{\Sigma_1, \Sigma_2\}$ has CPA security. In other words, it is not necessary that *both* Σ_1 and Σ_2 are secure.

9 Block Cipher Modes of Operation

Block ciphers / PRPs can only act on a single block (element of $\{0,1\}^{blen}$) of data at a time. In the previous section we showed at least one way to use a PRP (in fact, a PRF sufficed) to achieve CPA-secure encryption of a single block of data. This prompts the question, *can we use PRFs/PRPs to encrypt data that is longer than a single block?*

A **block cipher mode** specifies how to handle data that spans multiple blocks. Block cipher modes are where block ciphers really shine. There are modes for (CPA-secure) encryption, modes for data integrity, modes that achieve both privacy and integrity, modes for hard drive encryption, modes that gracefully recover from errors in transmission, modes that are designed to croak upon transmission errors, and so on. There is something of a cottage industry of clever block cipher modes, each with their own unique character and properties. Think of this chapter as a tour through the most common modes.

9.1 Common Modes

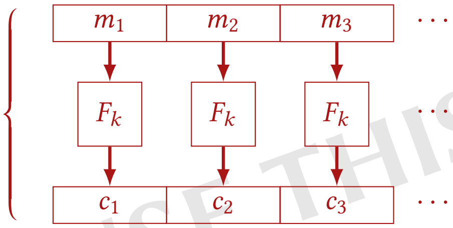
In this section, we will consider encryption modes for long plaintexts. As usual, *blen* will denote the blocklength of the block cipher F. In our diagrams, we'll write F_k as shorthand for $F(k,\cdot)$. When m is the plaintext, we will write $m = m_1 m_2 \cdots m_\ell$, where each m_i is a single block (so ℓ is the length of the plaintext measured in blocks). For now, we will assume that m is an exact multiple of the block length.

ECB: Electronic Codebook (NEVER NEVER USE THIS! NEVER‼)

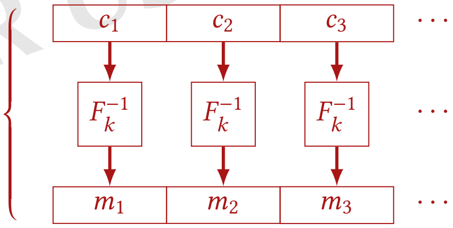
The most obvious way to use a block cipher to encrypt a long message is to just apply the block cipher independently to each block. The only reason to know about this mode is to know never to use it (and to scold anyone who does). It can't be said enough times: **never use ECB mode!** It does not provide security of encryption; can you see why?

Construction 9.1
(ECB Mode)

```
$\text{Enc}(k, m_1 \cdots m_\ell)$:
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i)$
  return $c_1 \cdots c_\ell$
```



```
$\text{Dec}(k, c_1 \cdots c_\ell)$:
  for $i = 1$ to $\ell$:
    $m_i := F^{-1}(k, c_i)$
  return $m_1 \cdots m_\ell$
```



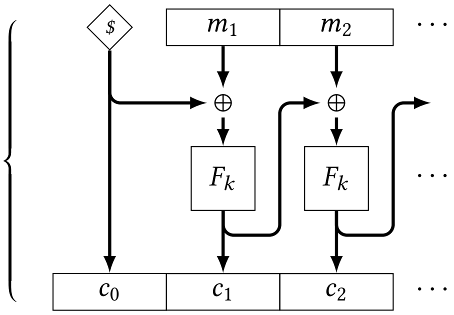
CBC: Cipher Block Chaining

CBC is by far the most common block cipher mode in everyday use. It is used primarily to achieve CPA (CPA$) security, but we will see later that a variant can also be used for data integrity properties.

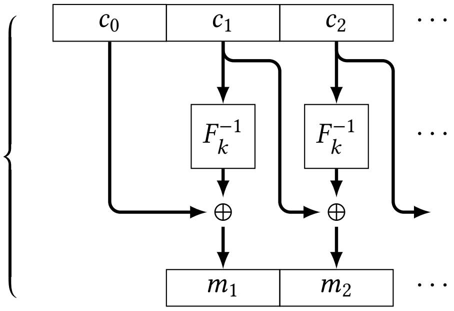
Since CBC mode results in CPA-secure encryption, it's no surprise that its encryption algorithm is *randomized.* In particular, CBC mode specifies that a random block is chosen, which is called the **initialization vector (IV).** The IV is included in the output, and as a result, encrypting ℓ blocks under CBC mode results in $\ell + 1$ blocks of output.

Construction 9.2
(CBC Mode)

```
$\text{Enc}(k, m_1 \cdots m_\ell)$:
  $c_0 \leftarrow \{0,1\}^{blen}$;
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i \oplus c_{i-1})$
  return $c_0 c_1 \cdots c_\ell$
```



```
$\text{Dec}(k, c_0 \cdots c_\ell)$:
  for $i = 1$ to $\ell$:
    $m_i := F^{-1}(k, c_i) \oplus c_{i-1}$
  return $m_1 \cdots m_\ell$
```



CTR: Counter

Counter mode, like CBC mode, begins with the choice of a random IV block $r \leftarrow \{0,1\}^{blen}$. Then the sequence

$$F(k,r); \quad F(k,r+1); \quad F(k,r+2); \quad \cdots$$

is used as a long one-time pad to mask the plaintext. Here, we interpret a block as an element of $\mathbb{Z}_{2^{blen}}$, so addition is performed modulo 2^{blen}.

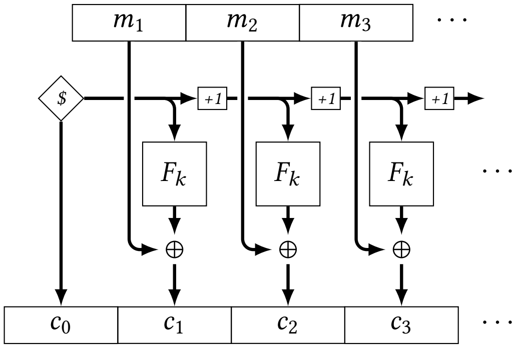
Counter mode has a few nice features that lead many designers to favor it over CBC:

- Both encryption and decryption use F only in the forward direction (the decryption algorithm for CTR is left as an exercise). In particular, this means that F can be a PRF that is not a PRP. Modes with this property are sometimes called *stream cipher modes.* They can be easily adapted to support plaintexts that are not an exact multiple of the blocklength.

- It is one of the few modes for which encryption can be parallelized. That is, once the IV has been chosen, the ith block of ciphertext can be computed without first computing the previous $i-1$ blocks.

Construction 9.3
(CTR Mode)

```
Enc(k, $m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^{blen}$
  $c_0 := r$
  for $i = 1$ to $\ell$:
     $c_i := F(k,r) \oplus m_i$
     $r := r + 1 \% 2^{blen}$
  return $c_0 \cdots c_\ell$
```



OFB: Output Feedback

OFB mode is another stream cipher mode that uses F only in the forward direction. As with the other modes, a random IV block r is chosen at the beginning. Then the sequence

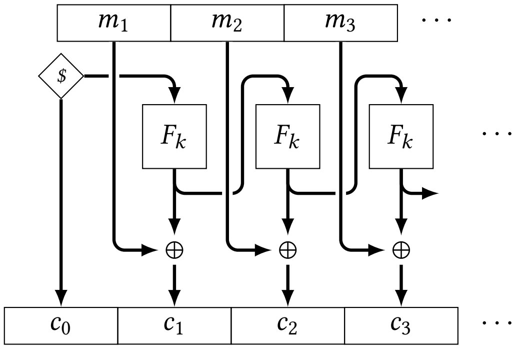
$$F(k,r); \quad F(k,F(k,r)); \quad F(k,F(k,F(k,r))); \quad \cdots$$

is used as a one-time pad to mask the plaintext. The name "output feedback" comes from this idea of repeatedly feeding the output of F into itself.

OFB is not nearly as widespread as CBC or CTR, but we include it here for self-serving reasons: it has the easiest security proof!

Construction 9.4
(OFB Mode)

```
Enc($k, m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^{blen}$
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r := F(k, r)$
    $c_i := r \oplus m_i$
  return $c_0 \cdots c_\ell$
```



9.2 CPA Security for Variable-Length Plaintexts

Block cipher modes allow us to encrypt a plaintext consisting of any number of blocks. In these modes, the length of the ciphertext depends on the length of the plaintext. So in that regard, these encryption schemes do not hide *all information* about the plaintext — in particular, they do not hide its length! We would run into problems trying to prove CPA security of these modes with respect to plaintext space $\mathcal{M} = (\{0,1\}^{blen})^*$. A successful distinguisher would break CPA security simply by chosing m_L and m_R of different lengths (measured in blocks) and then inspecting the length of the resulting ciphertext.

Leaking the length of the plaintext (measured in blocks) seems like a rather minor concession. But as we just described, it is necessary to modify our very conservative security definitions to allow any leakage at all.

For CPA security, we argued that if a distinguisher queries the CHALLENGE subroutine with plaintexts of different length in the CPA libraries, we cannot expect the libraries to be indistinguishable. The easiest way to avoid this situation is to simply insist that $|m_L| = |m_R|$. This would allow the adversary to make the challenge plaintexts differ in any way of his/her choice, *except in their length.* We can interpret the resulting security definition to mean that the scheme would hide all partial information about the plaintext apart from its length.

From now on, when discussing encryption schemes that support *variable-length* plaintexts, CPA security will refer to the following updated libraries:

```
$\mathcal{L}^{\Sigma}_{\text{cpa-L}}$

$k \leftarrow \Sigma$.KeyGen

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma$.Enc($k, m_L$)
  return $c$
```

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

$k \leftarrow \Sigma$.KeyGen

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma$.Enc($k, m_R$)
  return $c$
```

In the definition of CPA$ security (pseudorandom ciphertexts), the $\mathcal{L}_{\text{cpa\$-rand}}$ library responds to queries with uniform responses. Since these responses must look like ciphertexts, they must have the appropriate length. For example, for the modes discussed in this chapter, an ℓ-block plaintext is expected to be encrypted to an $(\ell + 1)$-block ciphertext. So, based on the length of the plaintext that is provided, the library must choose the appropriate ciphertext length. We are already using $\Sigma.\mathcal{C}$ to denote the set of possible ciphertexts of an encryption scheme Σ. So let's extend the notation slightly and write $\Sigma.\mathcal{C}(\ell)$ denote the set of possible ciphertexts for plaintexts of length ℓ. Then when discussing encryption schemes supporting variable-length plaintexts, CPA$ security will refer to the following libraries:

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-real}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\mathsf{Enc}(k, m)$
  return $c$
```

```
$\mathcal{L}^{\Sigma}_{\text{cpa\$-rand}}$
CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.\mathcal{C}(|m|)$
  return $c$
```

In the exercises, you are asked to prove that, with respect to these updated security definitions, CPA$ security implies CPA security as before.

Don't Take Length-Leaking for Granted!

We have just gone from requiring encryption to leak *no partial information* to casually allowing some specific information to leak. Let us not be too hasty about this!

If we want to truly support plaintexts of *arbitrary* length, then leaking the length is in fact unavoidable. But that doesn't mean it is consequence-free. By observing only the length of encrypted network traffic, many serious attacks are possible. Here are several examples:

- When accessing Google maps, your browser receives many image tiles that comprise the map that you see. Each image tile has the same pixel dimensions. However, they are compressed to save resources, and not all images compress as significantly as others. Every region of the world has its own rather unique "fingerprint" of image-tile lengths. So even though traffic to and from Google maps is encrypted, the sizes of the image tiles are leaked. This can indeed be used to determine the region for which a user is requesting a map.[1] The same idea applies to auto-complete suggestions in a search form.

- Variable-bit-rate (VBR) encoding is a common technique in audio/video encoding. When the stream is carrying less information (*e.g.*, a scene with a fixed camera position, or a quiet section of audio), it is encoded at a lower bit rate, meaning that each unit of time is encoded in fewer bits. In an encrypted video stream, the changes in bit rate are reflected as changes in packet length. When a user is watching a movie on Netflix or a Youtube video (as opposed to a live event stream), the bit-rate changes are consistent and predictable. It is therefore rather straight-forward to determine which video is being watched, even on an encrypted connection, just by observing the packet lengths.

[1] http://blog.ioactive.com/2012/02/ssl-traffic-analysis-on-google-maps.html

- ▶ VBR is also used in many encrypted voice chat programs. Attacks on these tools have been increasing in sophistication. The first attacks on encrypted voice programs showed how to identify who was speaking (from a set of candidates), just by observing the stream of ciphertext sizes. Later attacks could determine the language being spoken. Eventually, when combined with sophisticated linguistic models, it was shown possible to even identify individual words to some extent!

It's worth emphasizing again that none of these attacks involve any attempt to break the encryption. The attacks rely solely on the fact that encryption leaks the length of the plaintexts.

9.3 Security of OFB Mode

In this section we will prove that OFB mode has pseudorandom ciphertexts (when the blocklength is $blen = \lambda$ bits). OFB encryption and decryption both use the forward direction of F, so OFB provides security even when F is not invertible. Therefore we will prove security assuming F is simply a PRF.

Claim 9.5 *OFB mode (Construction 9.4) has CPA\$ security, if its underlying block cipher F is a secure PRF with parameters $in = out = \lambda$.*

Proof The general structure of the proof is very similar to the proof used for the PRF-based encryption scheme in the previous chapter (Construction 8.4). This is no coincidence: if OFB mode is restricted to plaintexts of a single block, we obtain exactly Construction 8.4!

The idea is that each ciphertext block (apart from the IV) is computed as $c_i := r \oplus m_i$. By the one-time pad rule, it suffices to show that the r values are independently pseudorandom. Each r value is the result of a call to the PRF. These PRF outputs will be independently pseudorandom only if all of the *inputs* to the PRF are *distinct.* In OFB mode, we use the *output* r of a previous PRF call as *input* to the next, so it is highly unlikely that this PRF output r matches a past PRF-input value. To argue this more precisely, the proof includes hybrids in which r is chosen without replacement (before changing r back to uniform sampling).

The formal sequence of hybrid libraries is given below:

$\mathcal{L}^{\text{OFB}}_{\text{cpa\$-real}}$:

```
$\mathcal{L}^{\text{OFB}}_{\text{cpa\$-real}}$
$k \leftarrow \{0,1\}^\lambda$

CHALLENGE($m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^\lambda$
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r := F(k, r)$
    $c_i := r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

The starting point is $\mathcal{L}^{\text{OFB}}_{\text{cpa\$-real}}$, shown here with the details of OFB filled in.

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^\lambda$
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r :=$ QUERY($r$)
    $c_i := r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$

$k \leftarrow \{0,1\}^\lambda$

QUERY($r$):
  return $F(k,r)$
```

The statements pertaining to the PRF F have been factored out in terms of $\mathcal{L}^F_{\text{prf-real}}$.

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^\lambda$
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r :=$ QUERY($r$)
    $c_i := r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$

$T :=$ empty

QUERY($x$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^\lambda$
  return $T[x]$
```

$\mathcal{L}^F_{\text{prf-real}}$ has been replaced by $\mathcal{L}^F_{\text{prf-rand}}$. By the PRF security of F, the change is indistinguishable.

Next, all of the statements that involve sampling values for the variable r are factored out in terms of the $\mathcal{L}_{\text{samp-L}}$ library from Lemma 4.10:

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r :=$ SAMP()
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r :=$ QUERY($r$)
    $c_i := r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

$\diamond$

```
$T :=$ empty

QUERY($x$):
  if $T[x]$ undefined:
    $T[x] :=$ SAMP()
  return $T[x]$
```

$\diamond$

```
$\mathcal{L}_{\text{samp-L}}$

SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$
```

$\mathcal{L}_{\text{samp-L}}$ is then replaced by $\mathcal{L}_{\text{samp-R}}$. By Lemma 4.10, this change is indistinguishable:

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r :=$ SAMP()
  $c_0 := r$
  for $i = 1$ to $\ell$:
    $r :=$ QUERY($r$)
    $c_i := r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

$\diamond$

```
$T :=$ empty

QUERY($x$):
  if $T[x]$ undefined:
    $T[x] :=$ SAMP()
  return $T[x]$
```

$\diamond$

```
$\mathcal{L}_{\text{samp-R}}$

$R := \emptyset$

SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

Arguments to QUERY are never repeated i this hybrid, so the middle library can be significantly simplified:

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r$ := SAMP()
  $c_0$ := $r$
  for $i = 1$ to $\ell$:
    $r$ := QUERY($r$)
    $c_i$ := $r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

◇

```
QUERY($x$):
  $t$ := SAMP()
  return $t$
```

◇

```
$\mathcal{L}_{\text{samp-R}}$
$R := \emptyset$

SAMP():
  $r \leftarrow \{0,1\}^\lambda \setminus R$
  $R := R \cup \{r\}$
  return $r$
```

Next, $\mathcal{L}_{\text{samp-R}}$ is replaced by $\mathcal{L}_{\text{samp-L}}$. By Lemma 4.10, this change is indistinguishable:

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r$ := SAMP()
  $c_0$ := $r$
  for $i = 1$ to $\ell$:
    $r$ := QUERY($r$)
    $c_i$ := $r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

◇

```
QUERY($x$):
  $t$ := SAMP()
  return $t$
```

◇

```
$\mathcal{L}_{\text{samp-L}}$
SAMP():
  $r \leftarrow \{0,1\}^\lambda$
  return $r$
```

Subroutine calls to QUERY and SAMP are inlined:

```
CHALLENGE($m_1 \cdots m_\ell$):
  $r \leftarrow \{0,1\}^\lambda$
  $c_0$ := $r$
  for $i = 1$ to $\ell$:
    $r \leftarrow \{0,1\}^\lambda$
    $c_i$ := $r \oplus m_i$
  return $c_0 c_1 \cdots c_\ell$
```

Finally, the one-time pad rule is applied within the for-loop (omitting some common steps). Note that in the previous hybrid, each value of r is used *only once* as a one-time pad. The $i = 0$ case has also been absorbed into the for-loop. The result is $\mathcal{L}^{\text{OFB}}_{\text{cpa\$-rand}}$, since OFB encrypts plaintexts of ℓ blocks into $\ell + 1$ blocks.

```
$\mathcal{L}^{\text{OFB}}_{\text{cpa\$-rand}}$
CHALLENGE($m_1 \cdots m_\ell$):
  for $i = 0$ to $\ell$:
    $c_i \leftarrow \{0,1\}^\lambda$
  return $c_0 c_1 \cdots c_\ell$
```

The sequence of hybrids shows that $\mathcal{L}^{\text{OFB}}_{\text{cpa\$-real}} \approx \mathcal{L}^{\text{OFB}}_{\text{cpa\$-rand}}$, and so OFB mode has pseudorandom ciphertexts. ■

We proved the claim assuming F is a PRF only, since OFB mode does not require F to be invertible. Since we assume a PRF with parameters $in = out = \lambda$, the PRP switching lemma (Lemma 7.3) shows that OFB is secure also when F is a PRP with blocklength $n = \lambda$.

9.4 Padding & Ciphertext Stealing

So far we have assumed that all plaintexts are exact multiples of the blocklength. But data in the real world is not always so accommodating. How are block ciphers used in practice with data that has arbitrary length?

Padding

While real-world data does not always come in multiples of the blocklength (typically 128 or 256 bits), it does almost always come in multiples of *bytes* (8 bits). So for now let us assume that the data is represented as bytes. We will write bytes in hex, for example `8f`.

The most natural way to deal with plaintexts of odd lengths is to add some null bytes (byte `00`, consisting of all zeroes) to fill out the last block. However, this leads to problems when decrypting: how can one distinguish between null bytes added for padding and null bytes that were part of the original plaintext?

A **padding scheme** is a method to encode arbitrary-length data into data that is a multiple of the blocklength. Importantly, the padding scheme must be reversible! Many padding schemes are in use, and we show just one below.

The ANSI X.923 standard defines a padding scheme, for block length 128 bits = 16 bytes. In the algorithms below, $|x|$ refers to the length of string x, measured in *bytes*, not bits.

Construction 9.6 (ANSI X.923)

```
PAD(x):
    $\ell := 16 - (|x| \% 16)$
    return $x \| \textsc{padstr}(\ell)$

PADSTR(ℓ):
    return $\underbrace{\texttt{00} \cdots \texttt{00}}_{\ell-1}\ \boxed{\ell}$

UNPAD(y):
    if not $\textsc{validpad}(y)$ then return err
    $\ell :=$ last byte of $y$ (as integer)
    return first $|y| - \ell$ bytes of $y$

VALIDPAD(y):
    if $|y| \% 16 \neq 0$ then return false
    $\ell :=$ last byte of $y$ (as integer)
    if $\ell \notin \{1, \ldots, 16\}$ then return false
    return $\textsc{padstr}(\ell) \overset{?}{=}$ last $\ell$ bytes of $y$
```

Example *Below are some blocks (16 bytes) with valid and invalid X.923 padding:*

01	34	11	d9	81	88	05	57	1d	73	c3	**00**	**00**	**00**	**00**	**05**	⇒ *valid*
95	51	05	4a	d6	5a	a3	44	af	b3	85	00	00	**00**	**00**	**03**	⇒ *valid*
5b	1c	01	41	5d	53	86	4e	e4	94	13	e8	7a	89	c4	**71**	⇒ *invalid*
d4	0d	d8	7b	53	24	c6	d1	af	5f	d6	f6	**00**	**c0**	**00**	**04**	⇒ *invalid*

Note what happens in $\textsc{pad}(x)$ when x is already a multiple of 16 bytes in length. In that case, an additional 16 bytes are added — `00` ⋯ `00` `10`, since `10` is the hex encoding of 16. This is necessary because the unpadded x might already end with something that looks like valid ANSI X.923 padding!

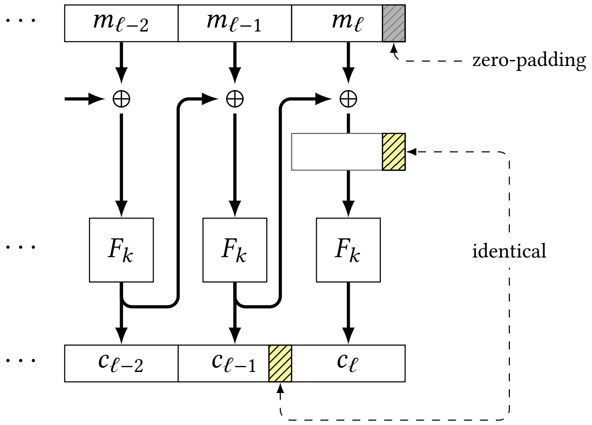
With a padding scheme available, one simply pads a string before encrypting and unpads the result after decrypting.

Ciphertext Stealing

Another approach with a provocative name is **ciphertext stealing** (CTS, if you are not yet tired of three-leter acronyms), which results in ciphertexts that are not a multiple of the blocklength. A plaintext of s bits will be encrypted to a ciphertext of $\mathit{blen} + s$ bits, where *blen* is the length of the extra IV block.

As an example, we will show how ciphertext stealing applies to CBC mode. Suppose the blocklength is *blen* and the last plaintext block m_ℓ has $\mathit{blen} - j$ bits. We extend m_ℓ with j zeroes and perform CBC encryption as usual.

Focus on the last j bits of block $c_{\ell-1}$. Because m_ℓ has been padded with zeroes, these final j bits of $c_{\ell-1}$ are also the last j bits of $F^{-1}(k, c_\ell)$, as shown in the diagram below.



Now imagine if we simply threw away the last j bits of $c_{\ell-1}$. These bits are somewhat redundant, so the decryption procedure would still have enough information to proceed. In particular, the decryption procedure could reconstruct the "missing" j bits via $F^{-1}(k, c_\ell)$. Furthermore, the fact that the ciphertext would be j bits shorter than a full block is a signal about how many bits should be treated in this special way (and how long the plaintext should be).

In general, the ciphertext stealing technique is to simply remove a carefully selected portion of the ciphertext, in a way that signals the original length of the plaintext and allows for correct decryption. For CBC mode, the appropriate bits to remove are the last j bits of the *next-to-last* ciphertext block. When decrypting, the last j bits of $F^{-1}(k, c_\ell)$ can be appended to $c_{\ell-1}$ and decryption can proceed as usual. The exercises ask you to prove the security of CBC mode with ciphertext stealing.

It is convenient in an implementation for the boundaries between blocks to be in predictable places. For that reason, it is slightly awkward to remove j bits from the *middle* of the ciphertext, as we have done here. So in practice, the last two blocks of the ciphertext are typically interchanged. For the example above, the resulting ciphertext (after ciphertext stealing) would be:

$$c_0\ c_1\ c_2 \cdots c_{\ell-3}\ c_{\ell-2}\ c_\ell\ c'_{\ell-1}, \text{ where } c'_{\ell-1} \text{ is the first } \mathit{blen} - j \text{ bits of } c_{\ell-1}.$$

That way, all ciphertext blocks except the last one are the full *blen* bits long, and the boundaries between blocks appear consistently every *blen* bits. This optimization does add some significant edge cases to any implementation. One must also decide what to do when the plaintext *is* an exact multiple of the blocklength — should the final two ciphertext blocks be swapped even in this case? Below we present an implementation of ciphertext stealing (CTS) that does *not* swap the final two blocks in this case. This means that it collapses to plain CBC mode when the plaintext is an exact multiple of the block length.

Construction 9.7 (CBC-CTS)

```
$\underline{\text{Enc}(k, m_1 \cdots m_\ell)}$:
  // each $m_i$ is blen bits,
  // except possibly $m_\ell$

  $j := blen - |m_\ell|$
  $m_\ell := m_\ell \| 0^j$
  | $c_0 \leftarrow \{0,1\}^{blen}$:
  | for $i = 1$ to $\ell$:
  |   $c_i := F(k, m_i \oplus c_{i-1})$
  if $j \neq 0$:
    remove final $j$ bits of $c_{\ell-1}$
    swap $c_{\ell-1}$ and $c_\ell$
  return $c_0 c_1 \cdots c_\ell$
```

```
$\underline{\text{Dec}(k, c_0 \cdots c_\ell)}$:
  // each $c_i$ is blen bits,
  // except possibly $c_\ell$

  $j := blen - |c_\ell|$
  if $j \neq 0$:
    swap $c_{\ell-1}$ and $c_\ell$
    $x :=$ last $j$ bits of $F^{-1}(k, c_\ell)$
    $c_{\ell-1} := c_{\ell-1} \| x$
  | for $i = 1$ to $\ell$:
  |   $m_i := F^{-1}(k, c_i) \oplus c_{i-1}$
  remove final $j$ bits of $m_\ell$
  return $m_1 \cdots m_\ell$
```

The marked lines correspond to plain CBC mode.

Exercises

9.1. Prove that a block cipher in ECB mode does not provide CPA security. Describe a distinguisher and compute its advantage.

9.2. Describe OFB decryption mode.

9.3. Describe CTR decryption mode.

9.4. CBC mode:

(a) In CBC-mode encryption, if a single bit of the plaintext is changed, which ciphertext blocks are affected (assume the same IV is used)?

(b) In CBC-mode decryption, if a single bit of the ciphertext is changed, which plaintext blocks are affected?

9.5. Prove that CPA\$ security for variable-length plaintexts implies CPA security for variable-length plaintexts. Where in the proof do you use the fact that $|m_L| = |m_R|$?

9.6. Suppose that instead of applying CBC mode to a block cipher, we apply it to one-time pad. In other words, we replace every occurrence of $F(k, \star)$ with $k \oplus \star$ in the code for CBC encryption. Show that the result does not have CPA security. Describe a distinguisher and compute its advantage.

9.7. Prove that there is an attacker that runs in time $O(2^{\lambda/2})$ and that can break CPA security of CBC mode encryption with constant probability.

9.8. Below are several block cipher modes for encryption, applied to a PRP F with blocklength $blen = \lambda$. For each of the modes:

- Describe the corresponding decryption procedure.
- Show that the mode does **not** have CPA-security. That means describe a distinguisher and compute its advantage.

(a)

```
Enc($k, m_1 \cdots m_\ell$):
  $r_0 \leftarrow \{0,1\}^\lambda$
  $c_0 := r_0$
  for $i = 1$ to $\ell$:
    $r_i := F(k, m_i)$
    $c_i := r_i \oplus r_{i-1}$
  return $c_0 \cdots c_\ell$
```

(b)

```
Enc($k, m_1 \cdots m_\ell$):
  $c_0 \leftarrow \{0,1\}^\lambda$
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i) \oplus c_{i-1}$
  return $c_0 \cdots c_\ell$
```

(c)

```
Enc($k, m_1 \cdots m_\ell$):
  $c_0 \leftarrow \{0,1\}^\lambda$
  $m_0 := c_0$
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i) \oplus m_{i-1}$
  return $c_0 \cdots c_\ell$
```

(d)

```
Enc($k, m_1 \cdots m_\ell$):
  $c_0 \leftarrow \{0,1\}^\lambda$
  $r_0 := c_0$
  for $i = 1$ to $\ell$:
    $r_i := r_{i-1} \oplus m_i$
    $c_i := F(k, r_i)$
  return $c_0 \cdots c_\ell$
```

Mode (a) is similar to CBC, except the XOR happens after, rather than before, the block cipher application. Mode (c) is essentially the same as CBC decryption.

9.9. Suppose you observe a CBC ciphertext and two of its blocks happen to be identical. What can you deduce about the plaintext? State some non-trivial property of the plaintext *that doesn't depend on the encryption key.*

9.10. The CPA\$-security proof for CBC encryption has a slight complication compared to the proof of OFB encryption. Recall that the important part of the proof is arguing that all inputs to the PRF are distinct.

In OFB, outputs of the PRF were fed directly into the PRF as inputs. The adversary had no influence over this process, so it wasn't so bad to argue that all PRF inputs were distinct (with probability negligibly close to 1).

By contrast, CBC mode takes an output block from the PRF, XOR's it with a plaintext block (which is after all *chosen by the adversary*), and uses the result as input to the next PRF call. This means we have to be a little more careful when arguing that CBC mode gives distinct inputs to all PRF calls (with probability negligibly close to 1).

(a) Prove that the following two libraries are indistinguishable:

```
$\mathcal{L}_{\text{left}}$
samp($m \in \{0,1\}^\lambda$):
    $r \leftarrow \{0,1\}^\lambda$
    return $r$
```

```
$\mathcal{L}_{\text{right}}$
$R := \emptyset$

samp($m \in \{0,1\}^\lambda$):
    $r \leftarrow \{r' \in \{0,1\}^\lambda \mid r' \oplus m \notin R\}$
    $R := R \cup \{r \oplus m\}$
    return $r$
```

(b) Using part (a), and the security of the underlying PRF, prove the CPA$-security of CBC mode.

Hint: In $\mathcal{L}_{\text{right}}$, let R correspond to the set of all inputs sent to the PRF. Let m correspond to the next plaintext block. Instead of sampling r (the output of the PRF) uniformly as in $\mathcal{L}_{\text{left}}$, we sample r so that $r \oplus m$ has never been used as a PRF-input before. This guarantees that the next PRF call will be on a "fresh" input.

Note: Appreciate how important it is that the adversary chooses plaintext block m before "seeing" the output r from the PRF (which is included in the ciphertext).

★ 9.11. Prove that CTR mode achieves CPA$ security.

Hint: Try to characterize the event that an IV is chosen so that the block cipher is called on the same value twice. Argue that this event happens with negligible probability, by defining an appropriate pair of libraries and showing they are indistinguishable.

9.12. Let F be a secure PRF with $out = in = \lambda$ and let $F^{(2)}$ denote the function $F^{(2)}(k,r) = F(k,F(k,r))$.

(a) Prove that $F^{(2)}$ is also a secure PRF.

(b) What if F is a secure PRP with blocklength *blen*? Is $F^{(2)}$ also a secure PRP?

9.13. In this question we investigate the subtleties of initialization vectors and how their choice contributes to CPA security. In CBC mode (indeed, in every mode discussed here), the IV is included in the ciphertext, so it is something that the adversary eventually sees anyway.

(a) Describe a modification to the CPA or CPA$ security definition that allows the adversary to *choose* the IV that is used. Then show that no such security is provided by CBC mode. In other words, describe a distinguisher for the new libraries you defined, and compute its advantage.

(b) Describe a modification to the CPA or CPA$ security definition that allows the adversary to choose the IV that is used, *but is not allowed to re-use an IV.* Does CBC mode satisfy this kind of security? If so, give a proof for your new kind of security notion; if not, then describe a distinguisher and compute its advantage.

(c) Describe a modification to the CPA or CPA$ security definition in which the library chooses IVs uniformly (as in the standard security definition) but allows the adversary to *see in advance what the next IV will be,* before choosing the plaintext. Does CBC mode satisfy this kind of security? If so, give a proof for your new kind of security notion; if not, then describe a distinguisher and compute its advantage.

In these problems, you are being asked to come up with a new security definition, so you must modify the libraries that comprise the security definition. There could be many "correct" ways to do this. All that matters is that the new interface of the libraries gives the calling program a way to do the things that are specified. In part (c) you might even want to add another subroutine to the interface.

9.14. The previous problem shows that the IV in CBC mode encryption is somewhat fragile. Suppose that in order to protect the IV, we send it through the block cipher before including it in the ciphertext. In other words, we modify CBC encryption as follows:

```
$\textsf{Enc}(k, m_1 \cdots m_\ell)$:
  $c_0 \leftarrow \{0,1\}^{blen}$;
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i \oplus c_{i-1})$
  $c'_0 := F(k, c_0)$
  return $c'_0\, c_1 \cdots c_\ell$
```

To decrypt, we first compute $c_0 := F^{-1}(k, c'_0)$ and proceed as in usual CBC decryption.

(a) Show that the resulting scheme no longer satisfies CPA security. Describe a successful distinguisher and compute its advantage.

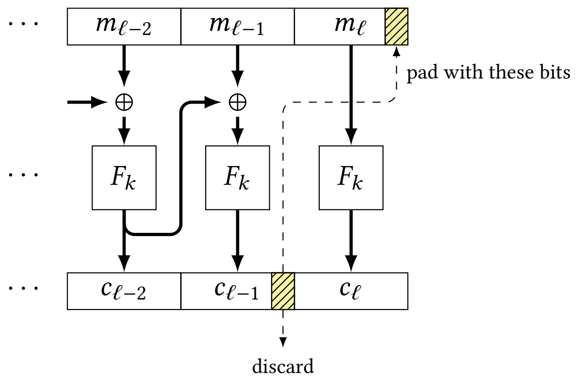
(b) Show that if we encrypt the IV c_0 with an independently chosen key k', the resulting scheme does in fact still achieve CPA\$ security. Your security proof can use the fact that standard CBC encryption has CPA\$ security.

9.15. Describe how to extend CTR and OFB modes to deal with plaintexts of arbitrary length (without using padding). Why is it so much simpler than CBC ciphertext stealing?

9.16. The following technique for ciphertext stealing in CBC was proposed in the 1980s and was even adopted into commercial products. Unfortunately, it's insecure.

Suppose the final plaintext block m_ℓ is $blen - j$ bits long. Rather than padding the final block with zeroes, it is padded with *the last j bits of ciphertext block $c_{\ell-1}$*. Then the padded block m_ℓ is sent through the PRP to produce the final ciphertext block c_ℓ. Since the final j bits of $c_{\ell-1}$ are recoverable from c_ℓ, they can be discarded.

If the final block of plaintext is already *blen* bits long, then standard CBC mode is used.



Show that the scheme does **not** satisfy CPA\$ security. Describe a distinguisher and compute its advantage.

Hint: ask for several encryptions of plaintexts whose last block is $blen - 1$ bits long.

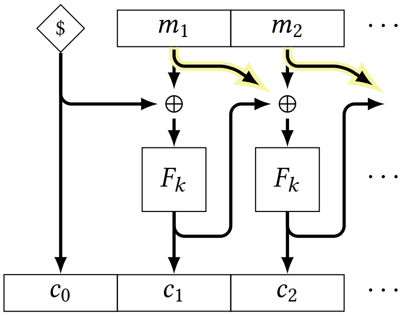
9.17. Prove that *any* CPA-secure encryption remains CPA-secure when augmented by padding the input.

9.18. Prove that CBC with ciphertext stealing has CPA\$ security. You may use the fact that CBC mode has CPA\$ security when restricted to plaintexts whose length is an exact multiple of the blocklength (*i.e.*, CBC mode without padding or ciphertext stealing).

Hint: Let CBC denote standard CBC mode restricted to plaintext space $\mathcal{M} = (\{0,1\}^{blen})^*$, and let CBC-CTS denote CBC mode with ciphertext stealing (so $\mathcal{M} = \{0,1\}^*$). Observe that it is easy to implement a call to $\mathcal{L}^{\text{CBC-CTS}}_{\text{cpa\$-real}}$ by a related call to $\mathcal{L}^{\text{CBC}}_{\text{cpa\$-real}}$ plus a small amount of additional processing.

9.19. Propagating CBC (PCBC) mode refers to the following variant of CBC mode:

```
$\mathsf{Enc}(k, m_1 \cdots m_\ell)$:
  $c_0 \leftarrow \{0,1\}^{blen}$;
  $m_0 := 0^{blen}$
  for $i = 1$ to $\ell$:
    $c_i := F(k, m_i \oplus c_{i-1} \oplus m_{i-1})$
  return $c_0 c_1 \cdots c_\ell$
```



(a) Describe PCBC decryption.

(b) Assuming that standard CBC mode has CPA\$-security (for plaintexts that are exact multiple of the block length), prove that PCBC mode also has CPA\$-security (for the same plaintext space).

Hint: Write PCBC encryption using plain CBC encryption as a subroutine.

(c) Consider the problem of adapting CBC ciphertext stealing to PCBC mode. Suppose the final plaintext block m_ℓ has $blen - j$ bits, and we pad it with the final j bits of the previous plaintext block $m_{\ell-1}$. Show that discarding the last j bits of $c_{\ell-1}$ still allows for correct decryption and results in CPA\$ security.

Hint: See Exercise 9.18.

(d) Suppose the final plaintext block is padded using the final j bits of the previous *ciphertext* block $c_{\ell-1}$. Although correct decryption is still possible, the construction is no longer secure. Show an attack violating the CPA\$-security of this construction. Why doesn't the proof approach from part (c) work?

Hint: Ask for several encryptions of plaintexts whose last block is 1 bit long.

10 Chosen Ciphertext Attacks

In this chapter we discuss the limitations of the CPA security definition. In short, the CPA security definition considers only the information leaked to the adversary by *honestly-generated* ciphertexts. It does not, however, consider what happens when an adversary is allowed to inject its own *maliciously crafted* ciphertexts into an honest system. If that happens, then even a CPA-secure encryption scheme can fail in spectacular ways. We begin by seeing such an example of spectacular and surprising failure, called a padding oracle attack:

10.1 Padding Oracle Attacks

Imagine a webserver that receives CBC-encrypted ciphertexts for processing. When receiving a ciphertext, the webserver decrypts it under the appropriate key and then checks whether the plaintext has valid X.923 padding (Construction 9.6).

Importantly, suppose that the *observable behavior of the webserver changes depending on whether the padding is valid.* You can imagine that the webserver gives a special error message in the case of invalid padding. Or, even more cleverly (but still realistic), the *difference in response time* when processing a ciphertext with invalid padding is enough to allow the attack to work.[1] The *mechanism* for learning padding validity is not important — what is important is simply the fact that an attacker has some way to determine whether a ciphertext encodes a plaintext with valid padding. No matter how the attacker comes by this information, we say that the attacker has access to a **padding oracle**, which gives the same information as the following subroutine:

```
PADDINGORACLE($c$):
  $m := \text{Dec}(k,c)$
  return VALIDPAD($m$)
```

We call this a padding *oracle* because it answers only one specific kind of question about the input. In this case, the answer that it gives is always a single boolean value.

It does not seem like a padding oracle is leaking useful information, and that there is no cause for concern. Surprisingly, we can show that an attacker who doesn't know the encryption key k can use a padding oracle alone to *decrypt **any** ciphertext of its choice!* This is true no matter what else the webserver does. As long as it leaks this one bit of information on ciphertexts that the attacker can choose, it might as well be leaking everything.

[1] This is why VALIDPAD should not actually be implemented the way it is written in Construction 9.6. A *constant-time* implementation — in which every execution path through the subroutine takes the same number of CPU cycles — is required.

to-do *Discuss some historical real-world examples: Vaudenay 2002 padding oracle attacks, Jager-Somorovsky 2011 XML encryption*

Malleability of CBC Encryption

Recall the definition of CBC decryption. If the ciphertext is $c = c_0 \cdots c_\ell$ then the ith plaintext block is computed as:

$$m_i := F^{-1}(k, c_i) \oplus c_{i-1}.$$

From this we can deduce two important facts:

- Two consecutive blocks (c_{i-1}, c_i) taken in isolation are a valid encryption of m_i. Looking ahead, this fact allows the attacker to focus on decrypting a single block at a time.
- XORing a ciphertext block with a known value (say, x) has the effect of XORing the corresponding plaintext block by the same value. In other words, for all x, the ciphertext $(c_{i-1} \oplus x, c_i)$ decrypts to $m_i \oplus x$:

$$\text{Dec}(k, (c_{i-1} \oplus x, c_i)) = F^{-1}(k, c_i) \oplus (c_{i-1} \oplus x) = (F^{-1}(k, c_i) \oplus c_{i-1}) \oplus x = m_i \oplus x.$$

If we send such a ciphertext $(c_{i-1} \oplus x, c_i)$ to the padding oracle, we would therefore learn whether $m_i \oplus x$ is a (single block) with valid padding. By carefully choosing different values x and asking questions of this form to the padding oracle, we can eventually learn *all of* m_i. We summarize this capability with the following subroutine:

```
// suppose c encrypts an (unknown) plaintext $m_1 \cdots m_\ell$
// does $m_i \oplus x$ (by itself) have valid padding?

CHECKXOR($c, i, x$):
  return PADDINGORACLE($c_{i-1} \oplus x, c_i$)
```

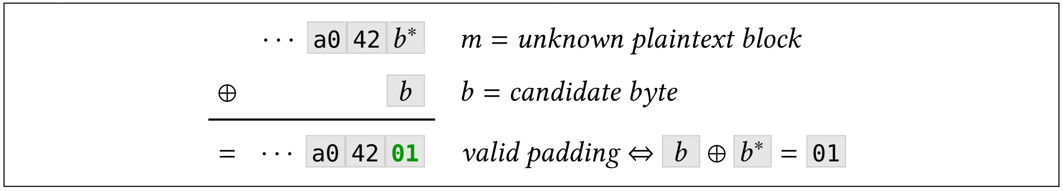
Given a ciphertext c that encrypts an unknown message m, we can see that an adversary can generate another ciphertext whose contents are *related to m in a predictable way.* This property of an encryption scheme is called **malleability.**

Learning the Last Byte of a Block

We now show how to use the CHECKXOR subroutine to determine the last byte of a plaintext block m. There are two cases to consider, depending on the contents of m. The attacker does not initially know which case holds:

For the first (and easier) of the two cases, suppose the second-to-last byte of m is nonzero. We will try every possible byte b and ask whether $m \oplus b$ has valid padding. Since m is a block and b is a single byte, when we write $m \oplus b$ we mean that b is extended on the left with zeroes. Since the second-to-last byte of m (and hence $m \oplus b$) is nonzero, only one of these possibilities will have valid padding — the one in which $m \oplus b$ ends in byte `01`. Therefore, if b is the candidate byte that succeeds (*i.e.*, $m \oplus b$ has valid padding) then the last byte of m must be $b \oplus$ `01`.

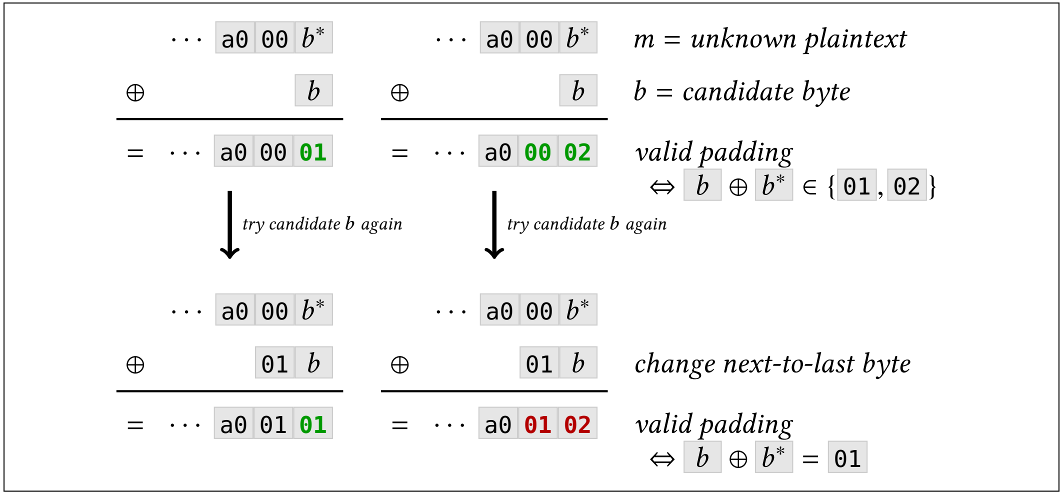
Example *Using* LEARNLASTBYTE *to learn the last byte b^* of the plaintext block m:*



··· a0 42 b^*
m = unknown plaintext block
⊕ b
b = candidate byte
= ··· a0 42 01
valid padding ⇔ $b \oplus b^*$ = 01

For the other case, suppose the second-to-last byte of m is zero. Then $m \oplus b$ will have valid padding for *several* candidate values of b:

Example *Using* LEARNLASTBYTE *to learn the last byte b^* of the plaintext block m:*



Whenever more than one candidate b value yields valid padding, we know that the second-to-last byte of m is zero (in fact, by counting the number of successful candidates, we can know exactly how many zeroes precede the last byte of m).

If the second-to-last byte of m is zero, then the second-to-last byte of $m \oplus$ 01 b is nonzero. The only way for both strings $m \oplus$ 01 b and $m \oplus b$ to have valid padding is when $m \oplus b$ ends in byte 01. We can re-try all of the successful candidate b values again, this time with an extra nonzero byte in front. There will be a unique candidate b that is successful in both rounds, and this will tell us that the last byte of m is $b \oplus$ 01.

The overall approach for learning the last byte of a plaintext block is summarized in the LEARNLASTBYTE subroutine in Figure 10.1. The set B contains the successful candidate bytes from the first round. There are at most 16 elements in B after the first round, since there are only 16 valid possibilities for the last byte of a properly padded block. In the worst case, LEARNLASTBYTE makes $256 + 16 = 272$ calls to the padding oracle (via CHECKXOR).

Learning Other Bytes of a Block

Once we have learned one of the trailing bytes of a plaintext block, it is slightly easier to learn additional ones. Suppose we know the last 3 bytes of a plaintext block, as in the example below. We would like to use the padding oracle to discover the 4th-to-last byte.

Example *Using* LEARNPREVBYTE *to learn the 4th-to-last byte b^* when the last 3 bytes of the block are already known.*

	···	b^*	**a0**	**42**	**3c**	m = *partially unknown plaintext block*
⊕		00	00	00	04	p = PADSTR(4)
⊕			a0	42	3c	s = *known bytes of* m
⊕		b	00	00	00	y = *candidate byte* b *shifted into place*
=	···	00	00	00	04	*valid padding* ⇔ b = b^*

Since we know the last 3 bytes of m, we can calculate a string x such that $m \oplus x$ ends in `00 00 04`. Now we can try XOR'ing the 4th-to-last byte of $m \oplus x$ with different candidate bytes b, and asking the padding oracle whether the result has valid padding. Valid padding only occurs when the result has `00` in its 4th-to-last byte, and this happens exactly when the 4th-to-last byte of m exactly matches our candidate byte b.

The process is summarized in the LEARNPREVBYTE subroutine in Figure 10.1. In the worst case, this subroutine makes 256 queries to the padding oracle.

Putting it all together. We now have all the tools required to decrypt *any ciphertext* using only the padding oracle. The process is summarized below in the LEARNALL subroutine.

In the worst case, 256 queries to the padding oracle are required to learn each byte of the plaintext.[2] However, in practice the number can be much lower. The example in this section was inspired by a real-life padding oracle attack[3] which includes optimizations that allow an attacker to recover each plaintext byte with only 14 oracle queries on average.

10.2 What Went Wrong?

CBC encryption has CPA security. Why didn't CPA security save us from padding oracle attacks? How was an attacker able to completely obliterate the privacy of encryption?

1. First, CBC encryption (in fact, every encryption scheme we've seen so far) has a property called **malleability.** Given an encryption c of an *unknown* plaintext m, it is possible to generate another ciphertext c' whose contents are *related to m in a predictable way.* In the case of CBC encryption, if ciphertext $c_0 \cdots c_\ell$ encrypts a plaintext $m_1 \cdots m_\ell$, then ciphertext $(c_{i-1} \oplus x, c_i)$ encrypts the *related* plaintext $m_i \oplus x$.

   In short, if an encryption scheme is malleable, then it allows information contained in one ciphertext to be "transferred" to another ciphertext.

2. Second, you may have noticed that the CPA security definition makes no mention of the Dec algorithm. The Dec algorithm shows up in our definition for *correctness*,

[2] It might take more than 256 queries to learn the last byte. But whenever LEARNLASTBYTE uses more than 256 queries, the side effect is that you've also learned that some other bytes of the block are zero. This saves you from querying the padding oracle altogether to learn those bytes.

[3] *How to Break XML Encryption*, Tibor Jager and Juraj Somorovsky. ACM CCS 2011.

```
CHECKXOR($c,i,x$):
  // if $c$ encrypts (unknown)
  // plaintext $m_1 \cdots m_\ell$; then
  // does $m_i \oplus x$ (by itself)
  // have valid padding?
  return PADDINGORACLE($c_{i-1} \oplus x, c_i$)

LEARNLASTBYTE($c,i$):
  // deduce the last byte of
  // plaintext block $m_i$
  $B := \emptyset$
  for $b$ = 00 to ff:
    if CHECKXOR($c,i,b$):
      $B := B \cup \{b\}$
  if $|B| = 1$:
    $b$ := only element of $B$
    return $b \oplus$ 01
  else:
    for each $b \in B$:
      if CHECKXOR($c,i,$ 01 $b$ ):
        return $b \oplus$ 01

LEARNPREVBYTE($c,i,s$):
  // knowing that $m_i$ ends in $s$,
  // find rightmost unknown
  // byte of $m_i$
  $p$ := PADSTR($|s|+1$)
  for $b$ = 00 to ff:
    $y := b\ \underbrace{\texttt{00}\cdots\texttt{00}}_{|s|}$
    if CHECKXOR($c,i,p \oplus s \oplus y$):
      return $b$

LEARNBLOCK($c,i$):
  // learn entire plaintext block $m_i$
  $s$ := LEARNLASTBYTE($c,i$)
  do 15 times:
    $b$ := LEARNPREVBYTE($c,i,s$)
    $s := b\|s$
  return $s$

LEARNALL($c$):
  // learn entire plaintext $m_1 \cdots m_\ell$
  $m := \epsilon$
  $\ell$ := number of non-IV blocks in $c$
  for $i = 1$ to $\ell$:
    $m := m\|$LEARNBLOCK($c,i$)
  return $m$
```

Figure 10.1: *Summary of padding oracle attack.*

but it is nowhere to be found in the $\mathcal{L}_{\text{cpa-}\star}$ libraries. Decryption has no impact on CPA security!

But the padding oracle setting involved the Dec algorithm — in particular, the adversary was allowed to see some information about the result of Dec applied to adversarially-chosen ciphertexts. Because of that, the padding oracle setting is no longer modeled well by the CPA security definition.

The bottom line is: give an attacker a malleable encryption scheme and access to any partial information related to decryption, and he/she can get information to leak out in surprising ways. As the padding-oracle attack demonstrates, even if *only a single bit of information* (*i.e.*, the answer to a yes/no question) is leaked about the result of decryption, this is frequently enough to extract the *entire plaintext.*

If we want security even under the padding-oracle scenario, we need a better security definition and encryption schemes that achieve it. That's what the rest of this chapter is about.

Discussion

- **Is this a realistic concern?** You may wonder whether this whole situation is somewhat contrived just to give cryptographers harder problems to solve. Indeed, that was probably a common attitude towards the security definitions introduced in this chapter. However, in 1998, Daniel Bleichenbacher demonstrated a devastating attack against early versions of the SSL protocol. By presenting millions of carefully crafted ciphertexts to a webserver, an attacker could eventually recover arbitrary SSL session keys.

  In practice, it is hard to make the external behavior of a server not depend on the result of decryption. This makes CPA security rather fragile in the real world. In the case of padding oracle attacks, mistakes in implementation can lead to different error messages for invalid padding. In other cases, the timing of the server's response can depend on the decrypted value, in a way that allows similar attacks.

  As we will see, it *is* in fact possible to provide security in these kinds of settings, and with low additional overhead. These days there is rarely a good excuse for using encryption which is only CPA-secure.

- Padding is in the name of the attack. But padding is not the culprit. The culprit is using a (merely) CPA-secure encryption scheme while allowing some information to leak about the result of decryption. The exercises expand on this further.

- The attack seems superficially like brute force, but it is not: The attack makes 256 queries per byte of plaintext, so it costs about 256ℓ queries for a plaintext of ℓ bytes. Brute-forcing the entire plaintext would cost 256^{ℓ} since that's how many ℓ-byte plaintexts there are. So the attack is exponentially better than brute force. The lesson is: brute-forcing small pieces at a time is much better then brute-forcing the entire thing.

10.3 Defining CCA Security

Our goal now is to develop a new security definition — one that considers an adversary that can construct malicious ciphertexts and observe the effects caused by their decryption. We will start with the basic approach of CPA security, where there are left and right libraries who differ only in which of two plaintexts they encrypt.

In a typical system, an adversary might be able to learn only some specific *partial information* about the Dec process. In the padding oracle attack, the adversary was able to learn only whether the result of decryption had valid padding.

However, we are trying to come up with a security definition that is useful *no matter how* the encryption scheme is deployed. How can we possibly anticipate every kind of partial information that might make its way to the adversary in every possible usage of the encryption scheme? The safest choice is to be as pessimistic as possible, as long as we end up with a security notion that we can actually achieve in the end. So **let's just allow the adversary to decrypt arbitrary ciphertexts of its choice.** In other words, if we can guarantee security when the adversary has *full* information about decrypted ciphertexts, then we certainly have security when the adversary learns only *partial* information about decrypted ciphertexts (as in a typical real-world system).

But this presents a significant problem. An adversary can do $c^* := \textsc{challenge}(m_L, m_R)$ to obtain a challenge ciphertext, and then immediately ask for that ciphertext c^* to be decrypted. This will obviously reveal to the adversary whether it is linked to the left or right library from the security definition.

So, simply providing unrestricted Dec access to the adversary cannot lead to a reasonable security definition (it is a security definition that can never be satisfied). But let's imagine the *smallest* possible patch to prevent this immediate and obvious attack. We can allow the adversary to ask for the decryption of **any ciphertext, except those produced in response to CHALLENGE queries.** In doing so, we arrive at the final security definition: security against chosen-ciphertext attacks, or CCA-security:

Definition 10.1 (CCA security) *Let Σ be an encryption scheme. We say that Σ has* ***security against chosen-ciphertext attacks (CCA security)*** *if $\mathcal{L}^{\Sigma}_{\text{cca-L}} \approx \mathcal{L}^{\Sigma}_{\text{cca-R}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{cca-L}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_L)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$
```

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

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_R)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
  if $c \in \mathcal{S}$ return null
  return $\Sigma.\text{Dec}(k, c)$
```

In this definition, the set $\mathcal{S}$ keeps track of the ciphertexts that have been generated by the CHALLENGE subroutine. The DEC subroutine rejects these ciphertexts outright, but will gladly decrypt any other ciphertext of the adversary's choice.

Pseudorandom Ciphertexts

We can also modify the pseudorandom-ciphertexts security definition (CPA\$ security) in a similar way:

Definition 10.2 (CCA\$ security) *Let Σ be an encryption scheme. We say that Σ has* ***pseudorandom ciphertexts in the***

presence of chosen-ciphertext attacks (CCA$ security) *if* $\mathcal{L}^{\Sigma}_{\text{cca\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$, *where:*

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$
$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m \in \Sigma.\mathcal{M})$:
    $c := \Sigma.\text{Enc}(k, m)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\text{Dec}(k, c)$
```

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$
$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m \in \Sigma.\mathcal{M})$:
    $c \leftarrow \Sigma.\mathcal{C}(|m|)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

$\textsc{dec}(c \in \Sigma.\mathcal{C})$:
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\text{Dec}(k, c)$
```

Just like for CPA security, if a scheme has CCA$ security, then it also has CCA security, but not vice-versa. See the exercises.

10.4 CCA Insecurity of Block Cipher Modes

With the padding oracle attack, we already showed that CBC mode does not provide security in the presence of chosen ciphertext attacks. But that attack was quite complicated since the adversary was restricted to learn just 1 bit of information at a time about a decrypted ciphertext. An attack in the full-fledged CCA setting can be much more direct.

Consider the adversary below attacking the CCA security of CBC mode (with block length *blen*)

```
$\mathcal{A}$
$c = c_0 c_1 c_2 := \textsc{challenge}(\texttt{0}^{2blen}, \texttt{1}^{2blen})$
$m := \textsc{dec}(c_0 c_1)$
return $m \overset{?}{=} \texttt{0}^{blen}$
```

It can easily be verified that this adversary achieves advantage 1 distinguishing $\mathcal{L}_{\text{cca-L}}$ from $\mathcal{L}_{\text{cca-R}}$. The attack uses the fact (also used in the padding oracle attack) that if $c_0 c_1 c_2$ encrypts $m_1 m_2$, then $c_0 c_1$ encrypts m_1. Ciphertext $c_0 c_1$ is clearly *related* to $c_0 c_1 c_2$ in an obvious (to us) way, but it is *different* than $c_0 c_1 c_2$, so the $\mathcal{L}_{\text{cca-}\star}$ libraries happily decrypt it.

Perhaps unsurprisingly, there are many very simple ways to catastrophically attack the CCA security of CBC-mode encryption. Here are some more (where $\overline{x}$ denotes the result of flipping every bit in x):

```
$\mathcal{A}'$
$c_0 c_1 c_2 := \textsc{challenge}(\texttt{0}^{2blen}, \texttt{1}^{2blen})$
$m := \textsc{dec}(c_0 c_1 \overline{c_2})$
if $m$ begins with $\texttt{0}^{blen}$ return 1 else return 0
```

```
$\mathcal{A}''$
$c_0 c_1 c_2 := \textsc{challenge}(0^{2blen}, 1^{2blen})$
$m := \textsc{dec}(\overline{c_0} c_1 c_2)$
return $m \overset{?}{=} 1^{blen} 0^{blen}$
```

The first attack uses the fact that modifying c_2 has no effect on the first plaintext block. The second attack uses the fact that flipping every bit in the IV flips every bit in m_1.

10.5 A Simple CCA-Secure Scheme

Recall the definition of a **strong** pseudorandom permutation (PRP) (Definition 7.6). A strong PRP is one that is indistinguishable from a randomly chosen permutation, even to an adversary that can make both *forward* (*i.e.*, to F) and *reverse* (*i.e.*, to F^{-1}) queries.

This concept has some similarity to the definition of CCA security, in which the adversary can make queries to both Enc and its inverse Dec. Indeed, a strong PRP can be used to construct a CCA-secure encryption scheme in a natural way:

Construction 10.3 *Let F be a pseudorandom permutation with block length $blen = n + \lambda$. Define the following encryption scheme with message space $\mathcal{M} = \{0,1\}^n$:*

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

Enc$(k,m)$:
    $r \leftarrow \{0,1\}^\lambda$
    return $F(k, m \| r)$

Dec$(k,c)$:
    $v := F^{-1}(k,c)$
    return first $n$ bits of $v$
```

In this scheme, m is encrypted by appending a random nonce r to it, then applying a PRP. We can informally reason about the security of this scheme as follows:

- Imagine an adversary linked to one of the CCA libraries. As long as the random value r does not repeat, all inputs to the PRP are distinct. The guarantee of a pseudorandom function/permutation is that its outputs (which are the *ciphertexts* in this scheme) will therefore all look independently uniform.

- The CCA library prevents the adversary from asking for c to be decrypted, if c was itself generated by the library. For any other value c' that the adversary asks to be decrypted, the guarantee of a *strong* PRP is that the result will look independently random. In particular, the result will not depend on the choice of plaintexts used to generate challenge ciphertexts. Since this choice of plaintexts is the only difference between the two CCA libraries, these decryption queries (intuitively) do not help the adversary.

Before we proceed with the proof, we first introduce a useful trick:

Claim 10.4 *Suppose a library has an internal variable X that is a* set *of items (stored explicitly as a list, so that its size is always polynomial in the security parameter). Let $n \geqslant \lambda$. Then it has a negligible effect on the calling program to change a statement "$x \leftarrow \{0,1\}^n$" to "$x \leftarrow$*

$\{0,1\}^n \setminus X$”, or vice-versa. In other words, the following two libraries are indistinguishable:

```
⋮
$x \leftarrow \{0,1\}^n$
⋮
```

```
⋮
$x \leftarrow \{0,1\}^n \setminus X$
⋮
```

This claim is essentially a restatement of Exercise 4.7, and I encourage you to prove it yourself. The proof generalizes the birthday bound proven in Section 4.4. Intuitively, $\{0,1\}^n$ is exponentially large (since $n \geqslant \lambda$), but the set X is only polynomially large. It is therefore only negligibly likely that the set X would get “hit” when sampling $x \leftarrow \{0,1\}^n$.

We can finally prove the CCA security of Construction 10.3 as follows:

Claim 10.5 *If F is a strong PRP (Definition 7.6) then Construction 10.3 has CCA\$ security (and therefore CCA security).*

Proof As usual, we prove the claim in a sequence of hybrids.

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

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$
$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda$
  $c := F(k, m \| r)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  return first $n$ bits of $F^{-1}(k,c)$
```

The starting point is $\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$, as expected, where Σ refers to Construction 10.3.

```
$\mathcal{S} := \emptyset$
$T, T^{-1}$ := empty assoc. arrays

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^{\lambda}$
  if $T[m\|r]$ undefined:
    $c \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
    $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $c := T[m\|r]$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen} \setminus \text{range}(T^{-1})$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
  return first $n$ bits of $T^{-1}[c]$
```

We have applied the strong PRP security (Definition 7.6) of F, skipping some standard intermediate steps. We factored out all invocations of F and F^{-1} in terms of the $\mathcal{L}_{\text{sprp-real}}$ library, replaced that library with $\mathcal{L}_{\text{sprp-rand}}$, and finally inlined it.

This proof has some subtleties, so it's a good time to stop and think about what needs to be done. To prove CCA\$-security, we must reach a hybrid in which the responses of CHALLENGE are uniform. In the current hybrid there are two properties in the way of this goal:

- The ciphertext values c are sampled from $\{0,1\}^{blen} \setminus \text{range}(T)$, rather than $\{0,1\}^{blen}$.

- When the if-condition in CHALLENGE is `false`, the return value of CHALLENGE is not a fresh random value but an old, repeated one. This happens when $T[m\|r]$ is already defined. Note that *both* CHALLENGE and DEC assign to T, so either one of these subroutines may be the cause of a pre-existing $T[m\|r]$ value.

Perhaps the most subtle fact about our current hybrid is that arguments of CHALLENGE can affect responses from DEC! In CHALLENGE, the library assigns $m\|r$ to a value $T^{-1}[c]$. Later calls to DEC will not read this value *directly*; these values of c are off-limits due to the guard condition in the first line of DEC. However, DEC samples a value from $\{0,1\}^{blen} \setminus \text{range}(T^{-1})$, which indeed uses the values $T^{-1}[c]$. To show CCA\$ security, we must remove this dependence of DEC on previous values given to CHALLENGE.

```
$\mathcal{S} := \emptyset$;    $\mathcal{R} := \emptyset$
$T, T^{-1}$ := empty assoc. arrays

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda$
  if $T[m\|r]$ undefined:
    $c \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
    $T[m\|r] := c$; $T^{-1}[c] := m\|r$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $c := T[m\|r]$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen} \setminus \text{range}(T^{-1})$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first $n$ bits of $T^{-1}[c]$
```

We have added some book-keeping that is not used anywhere. Every time an assignment of the form $T[m\|r]$ happens, we add r to the set $\mathcal{R}$. Looking ahead, we eventually want to ensure that r is chosen so that the if-statement in CHALLENGE is always taken, and the return value of CHALLENGE is always a *fresh* random value.

```
$\mathcal{S} := \emptyset$;    $\mathcal{R} := \emptyset$
$T, T^{-1}$ := empty assoc. arrays

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  if $T[m\|r]$ undefined:
    $c \leftarrow \{0,1\}^{blen}$
    $T[m\|r] := c$; $T^{-1}[c] := m\|r$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $c := T[m\|r]$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first $n$ bits of $T^{-1}[c]$
```

We have applied Claim 10.4 three separate times (showing only the final result).

In CHALLENGE, we've added a restriction to how r is sampled. This is done to ensure that r never repeats and the if-statement in CHALLENGE is always taken.

In CHALLENGE, we've removed the restriction in how c is sampled. Since c is the final return value of CHALLENGE, this gets us closer to our goal of this return value being uniformly random.

In DEC, we have removed the restriction in how $m\|r$ is sampled. As described above, this is because range(T^{-1}) contains previous arguments of CHALLENGE, and we don't want these arguments to affect the result of DEC in any way.

```
$\mathcal{S} := \emptyset$;   $\mathcal{R} := \emptyset$
$T, T^{-1}$ := empty assoc. array

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first $n$ bits of $T^{-1}[c]$
```

In the previous hybrid, the if-statement in CHALLENGE is *always taken.* This is because if $T[m\|r]$ is already defined, then r would already be in $\mathcal{R}$, but we are sampling r to avoid values in $\mathcal{R}$. We can therefore simply execute the body of the if-statement without actually checking the condition.

```
$\mathcal{S} := \emptyset$;   $\mathcal{R} := \emptyset$
$T, T^{-1}$ := empty assoc. array

CHALLENGE($m$):
  $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  // $T[m\|r] := c$; $T^{-1}[c] := m\|r$
  $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m\|r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first $n$ bits of $T^{-1}[c]$
```

In the previous hybrid, no line of code ever *reads* from T; they only *write* to T. It has no effect to remove a line that assigns to T, so we do so in CHALLENGE.

CHALLENGE also writes to $T^{-1}[c]$, but for a value $c \in \mathcal{S}$. The only line that *reads* from T^{-1} is in DEC, but the first line of DEC prevents it from being reached for such a $c \in \mathcal{S}$. It therefore has no effect to remove this assignment to T^{-1}.

```
$\mathcal{S} := \emptyset$;    // $\mathcal{R} := \emptyset$
$T, T^{-1}$ := empty assoc. array

CHALLENGE($m$):
  // $r \leftarrow \{0,1\}^\lambda \setminus \mathcal{R}$
  $c \leftarrow \{0,1\}^{blen}$
  // $\mathcal{R} := \mathcal{R} \cup \{r\}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m \| r \leftarrow \{0,1\}^{blen}$
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
    // $\mathcal{R} := \mathcal{R} \cup \{r\}$
  return first $n$ bits of $T^{-1}[c]$
```

Consider all the ways that $\mathcal{R}$ is used in the previous hybrid. The first line of CHALLENGE uses $\mathcal{R}$ to sample r, but then r is subsequently used only to further update $\mathcal{R}$ and nowhere else. Both subroutines use $\mathcal{R}$ only to update the value of $\mathcal{R}$. It has no effect to simply remove all lines that refer to variable $\mathcal{R}$.

```
$\mathcal{S} := \emptyset$
$T, T^{-1}$ := empty assoc. array

CHALLENGE($m$):
  $c \leftarrow \{0,1\}^{blen}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  if $T^{-1}[c]$ undefined:
    $m \| r \leftarrow \{0,1\}^{blen}$ \range($T^{-1}$)
    $T^{-1}[c] := m\|r$; $T[m\|r] := c$
  return first $n$ bits of $T^{-1}[c]$
```

We have applied Claim 10.4 to the sampling step in DEC. Now the second if-statement in DEC exactly matches $\mathcal{L}_{\text{sprp-rand}}$.

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

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$
$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

CHALLENGE($m$):
  $c \leftarrow \{0,1\}^{blen}$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return $c$

DEC($c \in \Sigma.\mathcal{C}$):
  if $c \in \mathcal{S}$ return null
  return first $n$ bits of $F^{-1}(k, c)$
```

We have applied the strong PRP security of F to replace $\mathcal{L}_{\text{sprp-rand}}$ with $\mathcal{L}_{\text{sprp-real}}$. The standard intermediate steps (factor out, swap library, inline) have been skipped. The result is $\mathcal{L}_{\text{cca\$-rand}}$.

We showed that $\mathcal{L}^{\Sigma}_{\text{cca\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$, so the scheme has CCA$ security. ■

Exercises

10.1. There is nothing particularly bad about padding schemes. They are only a target because padding is a commonly used structure in plaintexts that is verified upon decryption.

A *null character* is simply the byte `00`. Suppose you have access to the following oracle:

```
ORACLE($c$):
  $m := \text{Dec}(k, c)$
  if $m$ contains any null characters:
    return true
  else return false
```

Suppose you are given $c^* := \text{Enc}(k, m^*)$ for some unknown plaintext m^* and unknown key k. Assume that m^* is a multiple of the blocklength (so no padding is used), and that m^* contains no null characters.

1. Show how to use the oracle to completely decrypt m^*, when Enc uses CBC-mode encryption.
2. Show how to use the oracle to completely decrypt m^*, when Enc uses CTR-mode encryption.

10.2. PKCS#7 is a standard for padding that uses padding strings `01`, `02 02`, `03 03 03`, etc. Show how to decrypt arbitrary CBC-encrypted ciphertexts using a padding oracle that checks for correct PKCS#7 padding.

10.3. Sometimes encryption is as good as decryption, to an adversary.

(a) Suppose you have access to the following **encryption** oracle, where s is a secret that is consistent across all calls:

```
ECBORACLE($m$):
  // $k$, $s$ are secret
  return ECB.Enc($k, m \| s$)
```

Yes, this question is referring to the awful **ECB** encryption mode (Construction 9.1). Describe an attack that efficiently recovers all of s using access to ECBORACLE. Assume that if the length of $m \| s$ is not a multiple of the blocklength, then ECB mode will pad it with null bytes.

Hint: by varying the length of m, you can control where the block-division boundaries are in s.

(b) Now suppose you have access to a CBC encryption oracle, where you can control the IV that is used:

```
CBCORACLE($iv, m$):
  // $k, s$ are secret
  return CBC.Enc($k, iv, m \| s$)
```

Describe an attack that efficiently recovers all of s using access to CBCORACLE. As above, assume that $m\|s$ is padded to a multiple of the blocklength in some way. It is possible to carry out the attack no matter what the padding method is, as long as the padding method is known to the adversary.

10.4. Prove formally that CCA\$ security implies CCA security.

10.5. Let Σ be an encryption scheme with message space $\{0,1\}^n$ and define Σ^2 to be the following encryption scheme with message space $\{0,1\}^{2n}$:

```
KeyGen:
  $k \leftarrow \Sigma$.KeyGen
  return $k$

Enc($k, m$):
  $c_1 := \Sigma$.Enc($k, m_{\text{left}}$)
  $c_2 := \Sigma$.Enc($k, m_{\text{right}}$)
  return $(c_1, c_2)$

Dec($k, (c_1, c_2)$):
  $m_1 := \Sigma$.Dec($k, c_1$)
  $m_2 := \Sigma$.Dec($k, c_2$)
  if err $\in \{m_1, m_2\}$:
    return err
  else return $m_1 \| m_2$
```

(a) Prove that if Σ has CPA security, then so does Σ^2.

(b) Show that even if Σ has CCA security, Σ^2 does not. Describe a successful distinguisher and compute its distinguishing advantage.

10.6. Show that the following block cipher modes do not have CCA security. For each one, describe a successful distinguisher and compute its distinguishing advantage.

(a) OFB mode (b) CBC mode (c) CTR mode

10.7. Show that none of the schemes in Exercise 8.4 have CCA security. For each one, describe a successful distinguisher and compute its distinguishing advantage.

10.8. Alice believes that a CBC encryption of $0^{blen}\|m$ (where m is a single block) gives CCA security, when the Dec algorithm rejects ciphertexts when the first plaintext block is not all zeroes. Her reasoning is:

- The ciphertext has 3 blocks (including the IV). If an adversary tampers with the IV or the middle block of a ciphertext, then the first plaintext block will no longer be all zeroes and the ciphertext is rejected.
- If an adversary tampers with the last block of a ciphertext, then it will decrypt to $0^{blen}\|m'$ where m' is unpredictable from the adversary's point of view. Hence it will leak no information about the original m.

Is she right? Let CBC denote the encryption scheme obtained by using a secure PRF in CBC mode. Below we define an encryption scheme Σ' with message space $\{0,1\}^{blen}$ and ciphertext space $\{0,1\}^{3blen}$:

```
$\Sigma'$.KeyGen:
    $k \leftarrow$ CBC.KeyGen
    return $k$

$\Sigma'$.Enc($k, m$):
    return CBC.Enc($k, 0^{blen} \| m$)

$\Sigma'$.Dec($k, c$):
    $m = m_1 m_2 :=$ CBC.Dec($k, c$)
    if $m_1 = 0^{blen}$:
        return $m_2$
    else return err
```

Show that Σ' does **not** have CCA security. Describe a distinguisher and compute its distinguishing advantage. What part of Alice's reasoning was not quite right?

Hint: Obtain a ciphertext $c = c_0 c_1 c_2$ and another ciphertext $c' = c'_0 c'_1 c'_2$, both with known plaintexts. Ask the library for the decryption of $c_0 c_1 c'_2$.

10.9. CBC and OFB modes are malleable in very different ways. For that reason, Mallory claims that encrypting a plaintext (independently) with both modes results in CCA security, when the Dec algorithm rejects ciphertexts whose OFB and CBC plaintexts don't match. Is she right?

Let CBC denote the encryption scheme obtained by using a secure PRF in CBC mode. Let OFB denote the encryption scheme obtained by using a secure PRF in OFB mode. Below we define an encryption scheme Σ':

```
$\Sigma'$.KeyGen:
    $k_{cbc} \leftarrow$ CBC.KeyGen
    $k_{ofb} \leftarrow$ OFB.KeyGen
    return $(k_{cbc}, k_{ofb})$

$\Sigma'$.Enc($(k_{cbc}, k_{ofb}), m$):
    $c :=$ CBC.Enc($k_{cbc}, m$)
    $c' :=$ OFB.Enc($k_{ofb}, m$)
    return $(c, c')$

$\Sigma'$.Dec($(k_{cbc}, k_{ofb}), (c, c')$):
    $m :=$ CBC.Dec($k_{cbc}, c$)
    $m' :=$ OFB.Dec($k_{ofb}, c'$)
    if $m = m'$:
        return $m$
    else return err
```

Show that Σ' does **not** have CCA security. Describe a distinguisher and compute its distinguishing advantage.

10.10. This problem is a generalization of the previous one. Let Σ_1 and Σ_2 be two (possibly different) encryption schemes with the same message space $\mathcal{M}$. Below we define an encryption scheme Σ':

```
$\Sigma'$.KeyGen:
    $k_1 \leftarrow \Sigma_1$.KeyGen
    $k_2 \leftarrow \Sigma_2$.KeyGen
    return $(k_1, k_2)$

$\Sigma'$.Enc($(k_1, k_2), m$):
    $c_1 := \Sigma_1$.Enc($k_1, m$)
    $c_2 := \Sigma_2$.Enc($k_2, m$)
    return $(c_1, c_2)$

$\Sigma'$.Dec($(k_1, k_2), (c_1, c_2)$):
    $m_1 := \Sigma_1$.Dec($k_1, c_1$)
    $m_2 := \Sigma_2$.Dec($k_2, c_2$)
    if $m_1 = m_2$:
        return $m_1$
    else return err
```

Show that Σ' does **not** have CCA security, even if both Σ_1 and Σ_2 have *CCA* security. Describe a distinguisher and compute its distinguishing advantage.

10.11. Consider any padding scheme consisting of subroutines pad (which adds valid padding to its argument) and validpad (which checks its argument for valid padding). Assume that validpad(pad(x)) $= 1$ for all strings x.

Show that if an encryption scheme Σ has CCA security, then the following two libraries are indistinguishable:

```
$\mathcal{L}^{\Sigma}_{\text{pad-L}}$

$k \leftarrow \Sigma.\text{KeyGen}$

challenge($m_L, m_R \in \Sigma.\mathcal{M}$):
    if $|m_L| \neq |m_R|$ return null
    return $\Sigma.\text{Enc}(k, \text{pad}(m_L))$

paddingoracle($c \in \Sigma.\mathcal{C}$):
    return validpad($\Sigma.\text{Dec}(k, c)$)
```

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

$k \leftarrow \Sigma.\text{KeyGen}$

challenge($m_L, m_R \in \Sigma.\mathcal{M}$):
    if $|m_L| \neq |m_R|$ return null
    return $\Sigma.\text{Enc}(k, \text{pad}(m_R))$

paddingoracle($c \in \Sigma.\mathcal{C}$):
    return validpad($\Sigma.\text{Dec}(k, c)$)
```

That is, a CCA-secure encryption scheme hides underlying plaintexts in the presence of padding-oracle attacks.

Note: The distinguisher can even send a ciphertext c obtained from challenge as an argument to paddingoracle. Your proof should somehow account for this when reducing to the CCA security of Σ.

10.12. Show that an encryption scheme Σ has CCA\$ security if and only if the following two libraries are indistinguishable:

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

$k \leftarrow \Sigma.\text{KeyGen}$

challenge($m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\text{Enc}(k, m)$

dec($c \in \Sigma.\mathcal{C}$):
    return $\Sigma.\text{Dec}(k, c)$
```

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

$k \leftarrow \Sigma.\text{KeyGen}$
$D :=$ empty assoc. array

challenge($m \in \Sigma.\mathcal{M}$):
    $c \leftarrow \Sigma.\mathcal{C}(|m|)$
    $D[c] := m$
    return $c$

dec($c \in \Sigma.\mathcal{C}$):
    if $D[c]$ exists: return $D[c]$
    else: return $\Sigma.\text{Dec}(k, c)$
```

Note: In $\mathcal{L}_{\text{left}}$, the adversary can obtain the decryption of *any* ciphertext via dec. In $\mathcal{L}_{\text{right}}$, the dec subroutine is "patched" (via D) to give reasonable answers to ciphertexts generated in challenge.

11 Message Authentication Codes

When you've signed up for online services, you might have been asked to verify your email address. Typically the service will send you an email that contains a special activation code, or a link to click. How does this work?

A typical solution is for the service to generate a long random activation code. The activation code must be hard to guess — the only way to obtain a valid code should be to have the service generate one and send it to you. The service must also store a database that associates activation codes with email addresses. Such a database becomes fairly inconvenient for the service. Besides the cost to store the data, it is necessary to frequently prune the database of outdated/expired activation codes.

Using cryptography we can accomplish the same thing without any extra storage for the service. The idea is for the service to store only a short cryptographic key. Suppose we had a way of taking a key and a string (like an email address) and generating a unique cryptographic "stamp of approval" for that string. When a user signs up, the service computes the stamp of approval for their email address, and sends it to them as their activation code. Then the server doesn't have to remember any database of activation codes, only the cryptographic key. When a user returns, he/she will present both the activation code and the email address. The server can recompute the stamp of approval corresponding to the email address and check whether it matches the activation code presented by the user.

What kind of security do we require from such a cryptographic mechanism? These "stamps of approval" should be specific to each email address, and hard to guess. Even if I own hundreds of email addresses, use them to sign up, and see the stamps of approval for each one, it should still be hard for me to guess the stamp of approval for an email address that I *don't* own. Importantly, this is not a problem about *hiding* information. We are not trying to use an activation code to encrypt an email address in this setting — there is no need to hide an email address from the owner of that email address! Rather, the problem we care about is making sure an adversary cannot generate/guess a particular piece of data. This is a problem of **authenticity** (only someone with the key could have generated this data) rather than a problem of **privacy**.

11.1 Definition

The above requirements can be achieved using a tool called a message authentication code:

Definition 11.1 (MAC)

*A **message authentication code (MAC)** for message space $\mathcal{M}$ consists of the following algorithms:*

- KeyGen*: samples a key.*
- MAC*: takes a key k and message $m \in \mathcal{M}$ as input, and outputs a **tag** t. The* MAC *algorithm is deterministic.*

It is common to overload the term "MAC" and refer to the tag $t = \mathsf{MAC}(k,m)$ as "the MAC of m."

A MAC is like a signature that can be added to a piece of data. Our security requirement for a MAC is that only someone with the key can generate (and verify) a valid MAC. More precisely, an adversary who sees valid MACs of many (chosen) messages cannot produce a valid MAC of a *different* message (called a **forgery**).

How to Think About Authenticity Properties

How do we make a formal definition about authenticity, when all of the ones we've seen so far have been about privacy? Indeed, the The Prime Directive of Security Definitions of our security definitions is that indistinguishable libraries *hide information* about their internal differences. Before we see the security definition for MACs, let's start a much simpler statement: "no adversary should be able to guess a uniformly chosen value." We can formalize this idea with the following two libraries:

```
$\mathcal{L}_{\text{left}}$
$r \leftarrow \{0,1\}^\lambda$

GUESS($g$):
  return $g \overset{?}{=} r$
```

```
$\mathcal{L}_{\text{right}}$
GUESS($g$):
  return false
```

The left library allows the calling program to attempt to guess a uniformly chosen "target" string. The right library doesn't even bother to verify the calling program's guess — in fact it doesn't even bother to sample a random target string!

Focus on the difference between these two libraries. Their GUESS subroutines give the same output on nearly all inputs. There is only one input r on which they disagree. If a calling program can manage to find that input r, then it can easily distinguish the libraries. Therefore, if the libraries are indistinguishable, it means that the adversary cannot find/generate one of these special inputs! That's the kind of property we want to express.

Indeed, in this case, an adversary who makes q queries to the GUESS subroutine achieves an advantage of at most $q/2^\lambda$. For polynomial-time adversaries, this is a negligible advantage (since q is a polynomial function of λ).

Another way to think about this is to refer to the The Prime Directive of Security Definitions. Suppose we have two libraries, where a subroutine in one library checks some condition (and could return either `true` or `false`), while in the other library it always returns `false`. If the two libraries are indistinguishable, the The Prime Directive of Security Definitions says that their common interface leaks no information about their internal differences. In this case, the interface leaks no information about whether the library is actually checking the condition or always saying `false`. But if a calling program can't tell the difference, then *it must be very hard to find an input for which the answer should be* `true`.

The MAC Security Definition

Let's follow the pattern from the simple example above. We want to say that no adversary can generate a MAC forgery. So our libraries will provide a mechanism to let the adversary *check* whether it has a forgery. One library will actually perform the check, and the other library will simply assume that a forgery can never happen. The two libraries are different only in how they behave when the adversary calls a subroutine on a *true forgery.* So by demanding that the two libraries be indistinguishable, we are actually demanding that it is difficult for the calling program to generate a forgery.

Definition 11.2 (MAC security) *Let Σ be a MAC scheme. We say that Σ is a* **secure MAC** *if $\mathcal{L}^{\Sigma}_{\text{mac-real}} \approx \mathcal{L}^{\Sigma}_{\text{mac-fake}}$, where:*

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

$k \leftarrow \Sigma.\text{KeyGen}$

GETMAC($m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\text{MAC}(k,m)$

VER($m \in \Sigma.\mathcal{M}, t$):
    return $t \overset{?}{=} \Sigma.\text{MAC}(k,m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{mac-fake}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{T} := \emptyset$

GETMAC($m \in \Sigma.\mathcal{M}$):
    $t := \Sigma.\text{MAC}(k,m)$
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $t$

VER($m \in \Sigma.\mathcal{M}, t$):
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

Discussion:

- We do allow the adversary to request MACs of chosen messages, hence the GETMAC subroutine. This means that the adversary is always capable of obtaining valid MACs. However, MACs that were generated by GETMAC don't count as forgeries — only a MAC of a *different* message would be a forgery.

  For this reason, the $\mathcal{L}_{\text{mac-fake}}$ library keeps track of which MACs were generated by GETMAC, in the set $\mathcal{T}$. It is clear that these MACs should always be judged valid by VER. But for all other inputs to VER, $\mathcal{L}_{\text{mac-fake}}$ simply answers false.

- The adversary "wins" by successfully finding *any* forgery — a valid MAC of *any* "fresh" message. The definition doesn't care whether it's the MAC of any particular *meaningful* message.

Other Cool Applications

Although MACs are less embedded in public awareness than encryption, they are extremely useful. A frequent application of MACs is to delegate responsibility for storing data. Rather than storing a database of user-specific information on a server, we can just give the information to the user themself and ask them to bring it back when they want us to do something with it. We append a MAC of the data so that we can detect whether it has been tampered with. That is, the user can either present the data as we intended or else prevent the MAC from verifying (which we can detect). But we don't have to worry about the user actually changing our intended data.

- A **browser cookie** is a small piece of data that a webserver asks the browser to present each time a request is made. When a user logs in for the first time, the server can set a browser cookie saying "this browser session is for user so-and-so" rather than storing a database that associates users and browser sessions. Including a MAC in the cookie ensures that the user can't tamper with their cookies (*e.g.*, to masquerade as a different user).

- When Alice initiates a network connection to Bob, they must perform a **TCP handshake:**

    1. Alice sends a special SYN packet containing her initial sequence number A. In TCP, all packets from Alice to Bob include a sequence number, which helps the parties detect when packets are missing or out of order. It is important that the initial sequence number be random, to prevent other parties from injecting false packets.
    2. Bob sends a special SYN+ACK packet containing $A+1$ (to acknowledge Alice's A value) and the initial sequence number B for his packets.
    3. Alice sends a special ACK packet containing $B + 1$, and then the connection is established.

    When Bob is waiting for step 3, the connection is considered "half-open." While waiting, Bob must remember B so that he can compare to the $B + 1$ that Alice is supposed to send in her final ACK. Typically the operating system allocates only a very limited amount of resources for these half-open connections.

    In the past, it was possible to perform a denial of service attack by starting a huge number of TCP connections with a server, but never sending the final ACK packet. The server's queue for half-open connections fills up, which prevents other legitimate connections from starting.

    A clever backwards-compatible solution to this problem is called **SYN cookies.** The idea is to let Bob choose his initial sequence number B to be a MAC of the client's IP address, port number, and some other values. Now there is nothing to store for half-open connection. When Alice sends the final ACK of the handshake, Bob can recompute the initial sequence number from his MAC key.

These are all cases where the person who *generates* the MAC is the same person who later *verifies* the MAC. You can think of this person as choosing not to store some information, but rather leaving the information with someone else as a "note to self."

There are other useful settings where one party generates a MAC while the other verifies.

- In **two-factor authentication**, a user logs into a service using *something they know* (*e.g.*, a password) and *something they have* (*e.g.*, a mobile phone). The most common two-factor authentication mechanism is called *timed one-time passwords (TOTP).*

When you (as a user) enable two-factor authentication, you generate a secret key k and store it both on your phone and with the service provider. When you wish to log in, you open a simple app on your phone which computes $p = \text{MAC}(k,T)$, where T is the current date + time rounded to a multiple of 30 seconds. The value p is the "timed one-time password." You then log into the service using your usual (long-term) password and the one-time password p. The service provider has k and also knows the current time, so can verify the MAC p.

From the service provider's point of view, the only other place k exists is in the phone of this particular user. Intuitively, the only way to generate a valid one-time password at time T is to be in posession of this phone at time T. Even if an attacker sees both your long-term and one-time password over your shoulder, this does not help him gain access to your account in the future (well, not after 30 seconds in the future).

★ 11.2 A PRF is a MAC

The definition of a PRF says (more or less) that even if you've seen the output of the PRF on several chosen inputs, all other outputs look independently & uniformly random. Furthermore, uniformly chosen values are hard to guess, as long as they are sufficiently long (*e.g.*, λ bits).

In other words, after seeing some outputs of a PRF, any other PRF output will be hard to guess. This is exactly the intuitive property we require from a MAC. And indeed, we will prove in this section that a PRF is a secure MAC. While the claim makes intuitive sense, proving it formally is a little tedious. This is due to the fact that that in the MAC security game, the adversary can make many verification queries VER(m,t) *before* asking to see the correct MAC of m. Dealing with this event is the source of all the technical difficulty in the proof.

We start with a technical claim that captures the idea that "if you can blindly guess at uniformly chosen values and can also ask to see the values, then it is hard to guess a random value before you have seen it."

Claim 11.3 *The following two libraries are indistinguishable:*

```
$\mathcal{L}_{\text{guess-L}}$
$T :=$ empty assoc. array

GUESS($m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda}$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $g \overset{?}{=} T[m]$

REVEAL($m \in \{0,1\}^{in}$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

```
$\mathcal{L}_{\text{guess-R}}$
$T :=$ empty assoc. array

GUESS($m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda}$):
  // returns false if $T[m]$ undefined
  return $g \overset{?}{=} T[m]$

REVEAL($m \in \{0,1\}^{in}$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

Both libraries maintain an associative array T whose values are sampled uniformly the first time they are needed. Calling programs can try to guess these values via the GUESS subroutine, or simply learn them via REVEAL. Note that the calling program can call GUESS$(m,\cdot)$ both *before and after* calling REVEAL(m).

Intuitively, since the values in T are λ bits long, it should be hard to guess $T[m]$ before calling REVEAL(m). That is exactly what we formalize in $\mathcal{L}_{\text{guess-R}}$. In fact, this library doesn't bother to even choose $T[m]$ until REVEAL(m) is called. All calls to GUESS$(m,\cdot)$ made before the first call to REVEAL(m) will return `false`.

Proof We start by introducing two convenient hybrid libraries:

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

$count := 0$
$T :=$ empty assoc. array

GUESS($m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda}$):
  $count := count + 1$
  if $T[m]$ undefined and $count > H$:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $g \overset{?}{=} T[m]$
  // returns false if $T[m]$ undefined

REVEAL($m \in \{0,1\}^{in}$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

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

$count := 0$
$T :=$ empty assoc. array

GUESS($m \in \{0,1\}^{in}, g \in \{0,1\}^{\lambda}$):
  $count := count + 1$
  if $T[m]$ undefined and $count > H$:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
    if $g = T[m]$ and $count = H + 1$:
      return false
  return $g \overset{?}{=} T[m]$
  // returns false if $T[m]$ undefined

REVEAL($m \in \{0,1\}^{in}$):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

Each library references an unspecified variable H. In these libraries, the first several calls to GUESS will be answered with `false`. After some number of calls (determined by H), the GUESS subroutine will start actually comparing $T[m]$ to the given guess, and sample $T[m]$ on demand.

Suppose the calling program makes at most N queries to GUESS (so N is a polynomial in the security parameter). For any fixed value $h \in \{0,\ldots,N\}$, let $\mathcal{L}_{\text{hyb-}i}[h]$ denote one of the above libraries, substituting $H = h$. We can make the following observations:

$\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-1}}[0]$: In $\mathcal{L}_{\text{hyb-1}}[0]$, the clause "$count > 0$" is always true. Hence, the clause can be removed from the if-condition, resulting in $\mathcal{L}_{\text{guess-L}}$.

$\mathcal{L}_{\text{guess-R}} \equiv \mathcal{L}_{\text{hyb-1}}[N]$: In $\mathcal{L}_{\text{hyb-1}}[N]$, the clause "$count > N$" is always false. The entire if-statement in GUESS can be completely removed, resulting in $\mathcal{L}_{\text{guess-R}}$.

$\mathcal{L}_{\text{hyb-1}}[h] \approx \mathcal{L}_{\text{hyb-2}}[h]$: The only difference between libraries is the addition of the highlighted if-statement. Note that there is only one opportunity to trigger this if-statement (in the $(h+1)$th call to GUESS). In the previous line, $T[m]$ has just been chosen uniformly, so the probability of entering this if-statement is only $1/2^{\lambda}$. From Lemma 4.7 we know that two libraries are indistinguishable if their only difference is an if-statement that can be triggered with negligible probability.

$\mathcal{L}_{\text{hyb-2}}[h] \equiv \mathcal{L}_{\text{hyb-1}}[h+1]$: In both $\mathcal{L}_{\text{hyb-2}}[h]$ and $\mathcal{L}_{\text{hyb-1}}[h+1]$, the first $h+1$ calls to GUESS will always return `false`. The only difference is that, in the $(h+1)$th call to GUESS, $\mathcal{L}_{\text{hyb-2}}$ may eagerly sample some value $T[m]$, which in $\mathcal{L}_{\text{hyb-1}}$ would not be sampled until the next call of the form GUESS$(m,\cdot)$ or REVEAL(m). But the method of sampling is the same, only the timing is different. This difference has no effect on the calling program.

Putting everything together, we have:

$$\begin{aligned}\mathcal{L}_{\text{guess-L}} \equiv \mathcal{L}_{\text{hyb-1}}[0] \equiv \mathcal{L}_{\text{hyb-2}}[0] \approx \mathcal{L}_{\text{hyb-1}}[1] \equiv \mathcal{L}_{\text{hyb-2}}[1] \approx \cdots \\ \cdots \approx \mathcal{L}_{\text{hyb-1}}[N] \equiv \mathcal{L}_{\text{guess-R}}.\end{aligned}$$

This completes the proof. ■

We now return to the problem of proving that a PRF is a MAC.

Claim 11.4 *Let F be a secure PRF with input length in and output length out $= \lambda$. Then the scheme* MAC$(k,m) = F(k,m)$ *is a secure MAC for message space* $\{0,1\}^{in}$.

Proof We show that $\mathcal{L}^{F}_{\text{mac-real}} \approx \mathcal{L}^{F}_{\text{mac-fake}}$, using a standard sequence of hybrids.

```
$\mathcal{L}^{F}_{\text{mac-real}}$
$k \leftarrow \{0,1\}^{\lambda}$

GETMAC$(m)$:
  return $F(k,m)$

VER$(m,t)$:
  return $t \overset{?}{=} F(k,m)$
```

The starting point is the $\mathcal{L}_{\text{mac-real}}$ library, with the details of this MAC scheme filled in.

```
GETMAC$(m)$:
  return QUERY$(m)$

VER$(m,t)$:
  return $t \overset{?}{=}$ QUERY$(m)$
```

$\diamond$

```
$\mathcal{L}^{F}_{\text{prf-real}}$
$k \leftarrow \{0,1\}^{\lambda}$

QUERY$(x \in \{0,1\}^{in})$:
  return $F(k,x)$
```

We have factored out the PRF operations in terms of the library $\mathcal{L}_{\text{prf-real}}$ from the PRF security definition.

```
GETMAC(m):
  return QUERY(m)

VER(m,t):
  return $t \overset{?}{=}$ QUERY(m)
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$
$T$ := empty assoc. array

QUERY($x \in \{0,1\}^{in}$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{out}$
  return $T[x]$
```

We have applied the PRF-security of F and replaced $\mathcal{L}_{\text{prf-real}}$ with $\mathcal{L}_{\text{prf-rand}}$.

```
GETMAC(m):
  return REVEAL(m)

VER(m,t):
  return GUESS(m,t)
```

$\diamond$

```
$\mathcal{L}_{\text{guess-L}}$
$T$ := empty assoc. array

GUESS(m,g):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $g \overset{?}{=} T[m]$

REVEAL(m):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

We can express the previous hybrid in terms of the $\mathcal{L}_{\text{guess-L}}$ library from Claim 11.3. The change has no effect on the calling program.

```
GETMAC(m):
  return REVEAL(m)

VER(m,t):
  return GUESS(m,t)
```

$\diamond$

```
$\mathcal{L}_{\text{guess-R}}$
$T$ := empty assoc. array

GUESS(m,g):
  return $g \overset{?}{=} T[m]$

REVEAL(m):
  if $T[m]$ undefined:
    $T[m] \leftarrow \{0,1\}^{\lambda}$
  return $T[m]$
```

We have applied Claim 11.3 to replace $\mathcal{L}_{\text{guess-L}}$ with $\mathcal{L}_{\text{guess-R}}$. This involves simply removing the if-statement from GUESS. As a result, GUESS(m,g) will return `false` if $T[m]$ is undefined.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
    $t :=$ REVEAL($m$)
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $m$

VER($m,t$):
    return GUESS($m,t$)
```

$\diamond$

```
$\mathcal{L}_{\text{guess-R}}$

$T :=$ empty assoc. array

GUESS($m,g$):
    return $g \overset{?}{=} T[m]$

REVEAL($m$):
    if $T[m]$ undefined:
        $T[m] \leftarrow \{0,1\}^\lambda$
    return $T[m]$
```

Extra bookkeeping information is added, but not used anywhere. There is no effect on the calling program.

Consider the hybrid experiment above, and suppose the calling program makes a call to VER(m,t). There are two cases:

- Case 1: there was a previous call to GETMAC(m). In this case, the value $T[m]$ is defined in $\mathcal{L}_{\text{guess-R}}$ and $(m,T[m])$ already exists in $\mathcal{T}$. In this case, the result of GUESS(m,t) (and hence, of VER(m,t)) will be $t \overset{?}{=} T[m]$.
- Case 2: there was no previous call to GETMAC(m). Then there is no value of the form $(m,\star)$ in $\mathcal{T}$. Furthermore, $T[m]$ is undefined in $\mathcal{L}_{\text{guess-R}}$. The call to GUESS($m,t$) will return `false`, and so will the call to VER(m,t) that we consider.

In both cases, the result of VER(m,t) is `true` **if and only if** $(m,t) \in \mathcal{T}$.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
    $t :=$ REVEAL($m$)
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $m$

VER($m,t$):
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

```
$\mathcal{L}_{\text{guess-R}}$

$T :=$ empty assoc. array

GUESS($m,g$):
    return $g \overset{?}{=} T[m]$

REVEAL($m$):
    if $T[m]$ undefined:
        $T[m] \leftarrow \{0,1\}^\lambda$
    return $T[m]$
```

We have modified VER according to the discussion above.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
    $t :=$ QUERY (m)
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $m$

VER($m,t$):
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$

$T :=$ empty assoc. array

QUERY($x \in \{0,1\}^{in}$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{out}$
    return $T[x]$
```

In the previous hybrid, the GUESS subroutine is never called. Removing that unused subroutine and renaming REVEAL to QUERY results in the $\mathcal{L}_{\text{prf-ideal}}$ library from the PRF security definition.

```
$\mathcal{T} := \emptyset$

GETMAC($m$):
    $t :=$ QUERY($m$)
    $\mathcal{T} := \mathcal{T} \cup \{(m,t)\}$
    return $m$

VER($m,t$):
    return $(m,t) \overset{?}{\in} \mathcal{T}$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$

$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{in}$):
    return $F(k,x)$
```

We have applied the PRF security of F again, replacing $\mathcal{L}_{\text{prf-ideal}}$ with $\mathcal{L}_{\text{prf-real}}$.

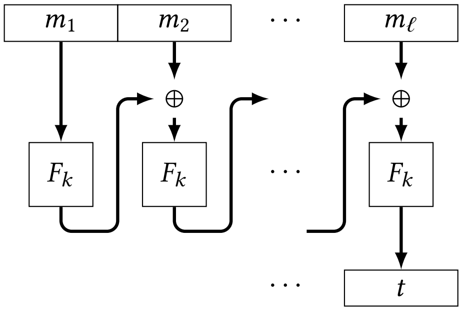
Inlining $\mathcal{L}_{\text{prf-real}}$ in the final hybrid, we see that the result is exactly $\mathcal{L}^F_{\text{mac-fake}}$. Hence, we have shown that $\mathcal{L}^F_{\text{mac-real}} \approx \mathcal{L}^F_{\text{mac-fake}}$, which completes the proof. ■

11.3 CBC-MAC

A PRF typically supports only messages of a fixed length, but we will soon see that it's useful to have a MAC that supports longer messages. A classical approach to extend the input size of a MAC involves the CBC block cipher mode applied to a PRF.

Construction 11.5 (CBC-MAC) *Let F be a PRF with $in = out = \lambda$. Then for every **fixed** parameter ℓ, CBC-MAC refers to the following MAC scheme:*

```
CBCMAC$^F(k, m_1 \cdots m_\ell)$:
    $t := 0^\lambda$:
    for $i = 1$ to $\ell$:
        $t := F(k, m_i \oplus t)$
    return $t$
```



CBC-MAC differs from CBC encryption mode in two important ways: First, there is no initialization vector. Indeed, CBC-MAC is deterministic (you can think of it as CBC encryption but with the initialization vector fixed to all zeroes). Second, CBC-MAC outputs only the last block.

Claim 11.6 *If F is a secure PRF with $in = out = \lambda$, then for every fixed ℓ, CBC-MAC is a secure MAC for message space $\mathcal{M} = \{0,1\}^{\lambda\ell}$.*

to-do *The proof of this claim is slightly beyond the scope of these notes. Maybe I will eventually find a way to incorporate it.*

Note that we have restricted the message space to messages of exactly ℓ blocks. Unlike CBC encryption, CBC-MAC is **not** suitable for messages of variable lengths. If the adversary is allowed to request the CBC-MAC of messages of different lengths, then it is possible for the adversary to generate a forgery (see the homework).

11.4 Encrypt-Then-MAC

Our motivation for studying MACs is that they seem useful in constructing a CCA-secure encryption scheme. The idea is to combine a MAC with a CPA-secure encryption scheme. The decryption algorithm can verify the MAC and raise an error if the MAC is invalid. There are several natural ways to combine a MAC and encryption scheme, but *not all are secure!* (See the exercises.) The safest way is known as encrypt-then-MAC:

Construction 11.7 (Enc-then-MAC) *Let E denote an encryption scheme, and M denote a MAC scheme where $E.\mathcal{C} \subseteq M.\mathcal{M}$ (*i.e.,* the MAC scheme is capable of generating MACs of ciphertexts in the E scheme). Then let EtM denote the **encrypt-then-MAC** construction given below:*

```
$\mathcal{K} = E.\mathcal{K} \times M.\mathcal{K}$
$\mathcal{M} = E.\mathcal{M}$
$\mathcal{C} = E.\mathcal{C} \times M.\mathcal{T}$

KeyGen:
  $k_e \leftarrow E.\text{KeyGen}$
  $k_m \leftarrow M.\text{KeyGen}$
  return $(k_e, k_m)$

Enc$((k_e, k_m), m)$:
  $c \leftarrow E.\text{Enc}(k_e, m)$
  $t := M.\text{MAC}(k_m, c)$
  return $(c, t)$

Dec$((k_e, k_m), (c, t))$:
  if $t \neq M.\text{MAC}(k_m, c)$:
    return err
  return $E.\text{Dec}(k_e, c)$
```

Importantly, the scheme computes a MAC *of the CPA ciphertext*, and not of the plaintext! The result is a CCA-secure encryption scheme:

Claim 11.8 *If E has CPA security and M is a secure MAC, then EtM (Construction 11.7) has CCA security.*

Proof As usual, we prove the claim with a sequence of hybrid libraries:

```
$\mathcal{L}^{EtM}_{\textsf{cca-L}}$

$k_e \leftarrow E.\textsf{KeyGen}$
$k_m \leftarrow M.\textsf{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R)$:
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\textsf{Enc}(k_e, m_L)$
  $t \leftarrow M.\textsf{MAC}(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{ (c, t) \}$
  return $(c, t)$

$\textsc{dec}(c, t)$:
  if $(c, t) \in \mathcal{S}$ return null
  if $t \neq M.\textsf{MAC}(k_m, c)$:
    return err
  return $E.\textsf{Dec}(k_e, c)$
```

The starting point is $\mathcal{L}^{EtM}_{\textsf{cca-L}}$, shown here with the details of the encrypt-then-MAC construction highlighted. Our goal is to eventually swap m_L with m_R. But the CPA security of E should allow us to do just that, so what's the catch?

To apply the CPA-security of E, we must factor out the relevant call to $E.\textsf{Enc}$ in terms of the CPA library $\mathcal{L}^{E}_{\textsf{cpa-L}}$. This means that k_e becomes private to the $\mathcal{L}_{\textsf{cpa-L}}$ library. But k_e is also used in the last line of the library as $E.\textsf{Dec}(k_e, c)$. The *CPA* security library for E provides no way to carry out such $E.\textsf{Dec}$ statements!

```
$k_e \leftarrow E.\textsf{KeyGen}$
$\mathcal{S} := \emptyset$

$\textsc{challenge}(m_L, m_R)$:
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E.\textsf{Enc}(k_e, m_L)$
  $t := \textsc{getmac}(c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
  return $(c, t)$

$\textsc{dec}(c, t)$:
  if $(c, t) \in \mathcal{S}$
    return null
  if not $\textsc{ver}(c, t)$ :
    return err
  return $E.\textsf{Dec}(k_e, c)$
```

$\diamond$

```
$\mathcal{L}^{M}_{\textsf{mac-real}}$

$k_m \leftarrow M.\textsf{KeyGen}$

$\textsc{getmac}(c)$:
  return $M.\textsf{MAC}(k_m, c)$

$\textsc{ver}(c, t)$:
  return $t \overset{?}{=} M.\textsf{MAC}(k_m, c)$
```

The operations of the MAC scheme have been factored out in terms of $\mathcal{L}^{M}_{\textsf{mac-real}}$. Notably, in the $\textsc{dec}$ subroutine the condition "$t \neq M.\textsf{MAC}(k_M, c)$" has been replaced with "not $\textsc{ver}(c, t)$."

```
$k_e \leftarrow E$.KeyGen
$\mathcal{S} := \emptyset$

CHALLENGE($m_L, m_R$):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E$.Enc($k_e, m_L$)
  $t :=$ GETMAC($c$)
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return $(c,t)$

DEC($c,t$):
  if $(c,t) \in \mathcal{S}$
    return null
  if not VER($c,t$):
    return err
  return $E$.Dec($k_e, c$)
```

$\diamond$

```
$\mathcal{L}^{M}_{\text{mac-fake}}$

$k_m \leftarrow M$.KeyGen
$\mathcal{T} := \emptyset$

GETMAC($c$):
  $t := M$.MAC($k_m, c$)
  $\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
  return $t$

VER($c,t$):
  return $(c,t) \overset{?}{\in} \mathcal{T}$
```

We have applied the security of the MAC scheme, and replaced $\mathcal{L}_{\text{mac-real}}$ with $\mathcal{L}_{\text{mac-fake}}$.

```
$k_e \leftarrow E$.KeyGen
$k_m \leftarrow M$.KeyGen
$\mathcal{T} := \emptyset$
$\mathcal{S} := \emptyset$

CHALLENGE($m_L, m_R$):
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E$.Enc($k_e, m_L$)
  $t := M$.MAC($k_m, c$)
  $\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return $(c,t)$

DEC($c,t$):
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{T}$ :
    return err
  return $E$.Dec($k_e, c$)
```

We have inlined the $\mathcal{L}_{\text{mac-fake}}$ library. This library keeps track of a set $\mathcal{S}$ of values for the purpose of the CCA interface, but also a set $\mathcal{T}$ of values for the purposes of the MAC. However, it is clear from the code of this library that $\mathcal{S}$ and $\mathcal{T}$ always have the same contents.

Therefore, the two conditions "$(c,t) \in \mathcal{S}$" and "$(c,t) \notin \mathcal{T}$" in the DEC subroutine are *exhaustive!* The final line of DEC is *unreachable.* This hybrid highlights the intuitive idea that an adversary can either query DEC with a ciphertext generated by CHALLENGE (the $(c,t) \in \mathcal{S}$ case) — in which case the response is null — or with a different ciphertext — in which case the response will be err since the MAC will not verify.

```
$k_e \leftarrow E$.KeyGen
$k_m \leftarrow M$.KeyGen
$\mathcal{S} := \emptyset$

CHALLENGE$(m_L, m_R)$:
  if $|m_L| \neq |m_R|$
    return null
  $c \leftarrow E$.Enc$(k_e, m_L)$
  $t := M$.MAC$(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return $(c,t)$

DEC$(c,t)$:
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{S}$:
    return err
  // unreachable
```

The unreachable statement has been removed and the redundant variables $\mathcal{S}$ and $\mathcal{T}$ have been unified. Note that this hybrid library never uses E.Dec, making it possible to express its use of the E encryption scheme in terms of $\mathcal{L}_{\text{cpa-L}}$.

```
$k_m \leftarrow M$.KeyGen
$\mathcal{S} := \emptyset$

CHALLENGE$(m_L, m_R)$:
  if $|m_L| \neq |m_R|$
    return null
  $c :=$ CHALLENGE$'(m_L, m_R)$
  $t := M$.MAC$(k_m, c)$
  $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
  return $(c,t)$

DEC$(c,t)$:
  if $(c,t) \in \mathcal{S}$
    return null
  if $(c,t) \notin \mathcal{S}$:
    return err
```

$\diamond$

```
$\mathcal{L}^{E}_{\text{cpa-L}}$
$k_e \leftarrow E$.KeyGen

CHALLENGE$'(m_L, m_R)$:
  $c := E$.Enc$(k_e, m_L)$
  return $c$
```

The statements involving the encryption scheme E have been factored out in terms of $\mathcal{L}_{\text{cpa-L}}$.

We have now reached the half-way point of the proof. The proof proceeds by replacing $\mathcal{L}_{\text{cpa-L}}$ with $\mathcal{L}_{\text{cpa-R}}$, applying the same modifications as before (but in reverse order), to finally arrive at $\mathcal{L}_{\text{cca-R}}$. The repetitive details have been omitted, but we mention that when listing the same steps in reverse, the changes appear very bizarre indeed. For instance, we add an unreachable statement to the DEC subroutine; we create a redundant variable $\mathcal{T}$ whose contents are the same as $\mathcal{S}$; we mysteriously change one instance of $\mathcal{S}$ (the condition of the second if-statement in DEC) to refer to the other variable $\mathcal{T}$. Of course, all of this is so that we can factor out the statements referring to the MAC scheme (along with $\mathcal{T}$) in terms of $\mathcal{L}_{\text{mac-fake}}$ and finally replace $\mathcal{L}_{\text{mac-fake}}$ with $\mathcal{L}_{\text{mac-real}}$. ■

Exercises

11.1. Consider the following MAC scheme, where F is a secure PRF with *in* = *out* = λ:

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

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  $m^* := 0^\lambda$
  for $i = 1$ to $\ell$:
    $m^* := m^* \oplus m_i$
  return $F(k, m^*)$
```

Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

11.2. Consider the following MAC scheme, where F is a secure PRF with *in* = *out* = λ:

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

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  $t := 0^\lambda$
  for $i = 1$ to $\ell$:
    $t := t \oplus F(k, m_i)$
  return $t$
```

Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

11.3. Consider the following MAC scheme, where F is a secure PRF with *in* = *out* = λ:

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

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  for $i = 1$ to $\ell$:
    $t_i := F(k, m_i)$
  return $(t_1, \ldots, t_\ell)$
```

Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

11.4. Consider the following MAC scheme, where F is a secure PRF with *in* = 2λ and *out* = λ:

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

MAC($k, m_1 \cdots m_\ell$): // each $m_i$ is $\lambda$ bits
  for $i = 1$ to $\ell$:
    $t_i := F(k, i \| m_i)$
  return $(t_1, \ldots, t_\ell)$
```

In the argument to F, we write $i \| m_i$ to denote the integer i (written as a λ-bit binary number) concatenated with the message block m_i. Show that the scheme is **not** a secure MAC. Describe a distinguisher and compute its advantage.

11.5. Suppose we expand the message space of CBC-MAC to $\mathcal{M} = (\{0,1\}^\lambda)^*$. In other words, the adversary can request a MAC on any message whose length is an exact multiple of the block length λ. Show that the result is **not** a secure MAC. Construct a distinguisher and compute its advantage.

Hint: Request a MAC on two single-block messages, then use the result to forge the MAC of a two-block message.

11.6. Let E be a CPA-secure encryption scheme and M be a secure MAC. Show that the following encryption scheme (called encrypt & MAC) is **not** CCA-secure:

```
$E\&M$.KeyGen:
    $k_e \leftarrow E$.KeyGen
    $k_m \leftarrow M$.KeyGen
    return $(k_e, k_m)$

$E\&M$.Enc$((k_e, k_m), m)$:
    $c \leftarrow E$.Enc$(k_e, m)$
    $t := M$.MAC$(k_m, m)$
    return $(c, t)$

$E\&M$.Dec$((k_e, k_m), (c, t))$:
    $m := E$.Dec$(k_e, c)$
    if $t \neq M$.MAC$(k_m, m)$:
        return err
    return $m$
```

Describe a distinguisher and compute its advantage.

11.7. Let E be a CPA-secure encryption scheme and M be a secure MAC. Show that the following encryption scheme Σ (which I call encrypt-and-encrypted-MAC) is **not** CCA-secure:

```
$\Sigma$.KeyGen:
    $k_e \leftarrow E$.KeyGen
    $k_m \leftarrow M$.KeyGen
    return $(k_e, k_m)$

$\Sigma$.Enc$((k_e, k_m), m)$:
    $c \leftarrow E$.Enc$(k_e, m)$
    $t := M$.MAC$(k_m, m)$
    $c' \leftarrow E$.Enc$(k_e, t)$
    return $(c, c')$

$\Sigma$.Dec$((k_e, k_m), (c, c'))$:
    $m := E$.Dec$(k_e, c)$
    $t := E$.Dec$(k_e, c')$
    if $t \neq M$.MAC$(k_m, m)$:
        return err
    return $m$
```

Describe a distinguisher and compute its advantage.

★ 11.8. In Construction 8.4, we encrypt one plaintext block into two ciphertext blocks. Imagine applying the Encrypt-then-MAC paradigm to this encryption scheme, but (erroneously) computing a MAC of *only* the second ciphertext block.

In other words, let F be a PRF with $in = out = \lambda$, and let M be a MAC scheme for message space $\{0,1\}^\lambda$. Define the following encryption scheme:

```
KeyGen:
    $k_e \leftarrow \{0,1\}^\lambda$
    $k_m \leftarrow M$.KeyGen
    return $(k_e, k_m)$

Enc$((k_e, k_m), m)$:
    $r \leftarrow \{0,1\}^\lambda$
    $x := F(k_e, r) \oplus m$
    $t := M$.MAC$(k_m, x)$
    return $(r, x, t)$

Dec$((k_e, k_m), (r, x, t))$:
    if $t \neq M$.MAC$(k_m, x)$:
        return err
    else return $F(k_e, r) \oplus x$
```

Show that the scheme does **not** have CCA security. Describe a successful attack and compute its advantage.

Hint: Suppose (r, x, t) and (r', x', t') are valid encryptions, and consider:

$$\mathsf{Dec}((k_e, k_m), (r', x, t)) \oplus x \oplus x'.$$

11.9. When we combine different cryptographic ingredients (*e.g.*, combining a CPA-secure encryption scheme with a MAC to obtain a CCA-secure scheme) we generally require the two ingredients to use *separate, independent keys.* It would be more convenient if the entire scheme just used a single λ-bit key.

(a) Suppose we are using Encrypt-then-MAC, where both the encryption scheme and MAC have keys that are λ bits long. Refer to the proof of security in the notes (11.4) and **describe where it breaks down** when we modify Encrypt-then-MAC to use the same key for both the encryption & MAC components:

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

Enc($k$, $m$):
  $c \leftarrow E.\text{Enc}(k, m)$
  $t := M.\text{MAC}(k, c)$
  return $(c, t)$

Dec($k$, $(c, t)$):
  if $t \neq M.\text{MAC}(k, c)$:
    return err
  return $E.\text{Dec}(k, c)$
```

(b) While Encrypt-then-MAC requires independent keys k_e and k_m for the two components, show that they can both be *derived* from a single key using a PRF.

In more detail, let F be a PRF with *in* $= 1$ and *out* $= \lambda$. Prove that the following modified Encrypt-then-MAC construction is CCA-secure:

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

Enc($k^*$, $m$):
  $k_e := F(k^*, 0)$
  $k_m := F(k^*, 1)$
  $c \leftarrow E.\text{Enc}(k_e, m)$
  $t := M.\text{MAC}(k_m, c)$
  return $(c, t)$

Dec($k^*$, $(c, t)$):
  $k_e := F(k^*, 0)$
  $k_m := F(k^*, 1)$
  if $t \neq M.\text{MAC}(k_m, c)$:
    return err
  return $E.\text{Dec}(k_e, c)$
```

You should not have to re-prove all the tedious steps of the Encrypt-then-MAC security proof. Rather, you should apply the security of the PRF in order to reach the *original* Encrypt-then-MAC construction, whose security we already proved (so you don't have to repeat).

12 Hash Functions

A **hash function** is any function that takes arbitrary-length input and has fixed-length output, so $H : \{0,1\}^* \to \{0,1\}^n$. Think of $H(m)$ as a "fingerprint" of m. Calling $H(m)$ a fingerprint suggests that different messages always have different fingerprints. But we know that can't be true — there are infinitely many messages but only 2^n possible outputs of H.[1]

Let's look a little closer. A true "unique fingerprinting function" would have to be *injective* (one-to-one). No hash function can be injective, since there must exist many x and x' with $x \neq x'$ but $H(x) = H(x')$. Let us call (x, x') a **collision** under H if it has this property. We know that collisions must exist, but what if *the problem of finding a collision* was hard for polynomial-time programs? Recall that in this course we often don't care whether something is impossible in principle — it is enough for something to be merely computationally difficult. A hash function for which collision-finding is hard would effectively serve as an injective function for our purposes.

Roughly speaking, a hash function H is **collision-resistant** if no polynomial-time program can find a collision in H. Another good name for such a hash function might be "pseudo-injective." In this chapter we discuss definitions and applications of collision resistance.

12.1 Defining Security

Superficially, it seems like we have already given the formal definition of security: A hash function H is collision-resistant if no polynomial-time algorithm can output a collision under H. Unfortunately, this definition is impossible to achieve!

Fix your favorite hash function H. We argued that collisions in H definitely exist in a mathematical sense. Let x, x' be one such collision, and consider the adversary $\mathcal{A}$ that has x, x' hard-coded and simply outputs them. This adversary runs in constant time and finds a collision in H. Of course, even though $\mathcal{A}$ exists in a mathematical sense, it might be hard to *write down* the code of such an $\mathcal{A}$ given H. But (and this is a subtle technical point!) security definitions consider only the running time of $\mathcal{A}$, and not the effort that goes into *finding the source code* of $\mathcal{A}$.[2]

The way around this problem is to introduce some random choice made by the user of the hash function, who wants collisions to be hard to find. A **hash function family** is a set $\mathcal{H}$ of functions, where each function $H \in \mathcal{H}$ is a hash function with the same output length. We will require that collisions are hard to find, in a hash function *chosen randomly from the family.* This is enough to foil the hard-coded-collision distinguisher mentioned

[1] Somewhere out there is a pigeonhole with infinitely many pigeons in it.

[2] The reason we don't define security this way is that as soon as someone *does* find the code of such an $\mathcal{A}$, the hash function H is "broken" forever. Nothing anyone does (like choosing a new key) can salvage it.

above. Think of a hash function family as having exponentially many functions in it — then no polynomial-time program can have a hard-coded collision for *all* of them.

Now the difficulty of finding collisions rests in the random choice of functions. An adversary can know every fact about $\mathcal{H}$, it just doesn't know which $H \in \mathcal{H}$ it is going to be challenged on to find a collision. It's similar to how the security of other cryptographic schemes rests in the random choice of key. But in this case there is no secrecy involved, only unpredictability. The choice of H is made public to the adversary.

Note also that this definition is a mismatch to the way hash functions are typically used in practice. There, we usually do have a *single* hash function that we rely on and standardize. While it is possible to adapt the definitions and results in this lecture to the setting of fixed hash functions, it's simpler to consider hash function *families.* If you're having trouble connecting the idea of a hash function *family* to reality, imagine taking a standardized hash function like MD5 or SHA3 and considering the family of functions you get by varying the initialization parameters in those standards.

Towards the Formal Definition

The straight-forward way to define collision resistance of a hash family $\mathcal{H}$ is to say that the following two libraries should be indistinguishable:

```
$H \leftarrow \mathcal{H}$

GETH():
  return $H$

CHALLENGE($x, x' \in \{0,1\}^*$):
  return $H(x) \overset{?}{=} H(x')$
```

```
$H \leftarrow \mathcal{H}$

GETH():
  return $H$

CHALLENGE($x, x' \in \{0,1\}^*$):
  return $x \overset{?}{=} x'$
```

Indeed, this is a fine definition of collision resistance, and it follows in the style of previous security definitions. The two libraries give different outputs only when called on (x, x') where $x \neq x'$ but $H(x) = H(x')$ — *i.e.*, an H-collision.

However, it turns out to not be a particularly convenient definition to *use* when proving security of a construction that involves a hash function as a building block. To see why, think back to the security of MACs. The difference between the two $\mathcal{L}_{\text{mac-}\star}$ libraries is in the verification subroutine VER. And indeed, constructions that use MACs as a building block often perform MAC verification. The libraries — in particular, their VER subroutine — are a good fit for how MACs are used.

On the other hand, cryptographic constructions don't usually *explicitly test for collisions* when using a hash function. Rather, they typically compute the hash of some value and move on with their business *under the assumption that a collision has never been encountered.* To model this in a security definition, we use the approach below:

Definition 12.1 *Let $\mathcal{H}$ be a family of hash functions. Then $\mathcal{H}$ is **collision-resistant** if $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}} \approx \mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$, where:*

```
$\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$
$H \leftarrow \mathcal{H}$

GETH():
    return $H$

HASH($x \in \{0,1\}^*$):
    $y := H(x)$
    return $y$
```

```
$\mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$
$H \leftarrow \mathcal{H}$
$H^{-1}$ := empty assoc. array

GETH():
    return $H$

HASH($x \in \{0,1\}^*$):
    $y := H(x)$
    if $H^{-1}[y]$ defined and $H^{-1}[y] \neq x$:
        self destruct
    $H^{-1}[y] := x$
    return $y$
```

Discussion:

- The two libraries have identical behavior, except in the event that $\mathcal{L}_{\text{cr-fake}}$ triggers a "**self destruct**" statement. Think of this statement as an exception that kills the entire program, including the distinguisher. In the case that the distinguisher is killed in this way, we take its output to be 0. Suppose a distinguisher A always outputs 1 under normal, explosion-free execution. Since an explosion happens only in $\mathcal{L}_{\text{cr-fake}}$, A's advantage is simply the probability of an explosion. So if the two libraries are supposed to be indistinguishable, then it must be that **self destruct** happens with only negligible probability.

- In $\mathcal{L}_{\text{cr-fake}}$ we have given the associative array "H^{-1}" a somewhat suggestive name. During normal operation, $H^{-1}[y]$ contains the *unique* value seen by the library whose hash is y. A collision happens when the library has seen two distinct values x and x' that hash to the same y. When the library sees the second of these values x', it computes $H(x') = y$ and discovers that $H^{-1}[y]$ already exists but is not equal to x'. This is the situation in which the library **self destructs**.

- Why do we make the library **self destruct** when it sees a collision, rather than just returning some error indicator? The reason is simply that it's easier to just *assume* there is no collision than to check the return value of HASH after each call.

  Think of $\mathcal{L}_{\text{cr-real}}$ as a world in which you take hashes of things but you might see a collision and never notice. Then $\mathcal{L}_{\text{cr-fake}}$ is a kind of thought-experiment in which some all-seeing judge ends the game immediately if there was ever a collision among the values that you hashed. The judge's presence simplifies things a bit. There's no real need to explicitly check for collisions yourself; you can just go about your business knowing that as long as the game is still going, the hash function is injective among all the values you've seen.

- The libraries have no secrets! H is public (the adversary can freely obtain it through GETH). However, note that the library — not the adversary — chooses H. This random choice of H is the sole source of security.

Since H is public, the adversary doesn't really need to call the subroutine HASH(x) to compute $H(x)$ — he could compute $H(x)$ locally. Intuitively, the library is used in a security proof to model the actions of the "good guys" who are operating on the assumption that H is collision-resistant. If an adversary finds a collision but never causes the "good guys" to evaluate H on it, then the adversary never violates their security assumption.

Other variants of collision-resistance.

There are some other variants of collision-resistance that are often discussed in practice. We don't define them formally, but give the rough idea:

Target collision resistance. Given H and $H(x)$, where $H \leftarrow \mathcal{H}$ and $x \leftarrow \{0,1\}^\ell$ are chosen randomly, it should be infeasible to compute a value x' (possibly equal to x) such that $H(x) = H(x')$.

Second-preimage resistance. Given H and x, where $H \leftarrow \mathcal{H}$ and $x \leftarrow \{0,1\}^\ell$ are chosen randomly, it should be infeasible to compute a value $x' \neq x$ such that $H(x) = H(x')$.

These conditions are weaker than the condition of (plain) collision-resistance, in the sense that if $\mathcal{H}$ is collision-resistant, then $\mathcal{H}$ is also target collision-resistant and second-preimage resistant. Hence, we focus on plain collision resistance in this course.

12.2 Hash-Then-MAC

In this section we'll see a simple application of collision resistance. It is a common theme in cryptography, that instead of dealing with large data it is often sufficient to deal with only a hash of that data. This theme is true in particular in the context of MACs.

One particularly simple way to construct a secure MAC is to use a PRF directly as a MAC. However, PRFs (and in particular, block ciphers) often have a short fixed input length, making them suitable only for MACs of such short messages. To extend such a MAC to longer inputs, it suffices to compute a MAC of the hash of the data. This idea is formalized in the following construction:

Construction 12.2 (Hash-then-MAC) *Let M be a MAC scheme with message space $\mathcal{M} = \{0,1\}^n$ and let $\mathcal{H}$ be a hash family with output length n. Then **hash-then-MAC** (HtM) refers to the following MAC scheme:*

```
                                  $HtM$.KeyGen:           $HtM$.MAC$((k,H),m)$:
$\mathcal{K} = M.\mathcal{K} \times \mathcal{H}$      $k \leftarrow M$.KeyGen      $y := H(m)$
$\mathcal{M} = \{0,1\}^*$              $H \leftarrow \mathcal{H}$              $t := M$.MAC$(k,y)$
                                    return $(k,H)$          return $t$
```

Claim 12.3 *Construction 12.2 is a secure MAC, if $\mathcal{H}$ is collision-resistant and M is a secure MAC.*

Proof We prove the security of HtM using a standard hybrid approach.

```
$\mathcal{L}^{HtM}_{\textsf{mac-real}}$
$k \leftarrow \textsf{KeyGen}$
$H \leftarrow \mathcal{H}$

$\textsc{getmac}(m)$:
    $y := H(m)$
    $t := \textsf{MAC}(k, y)$
    return $t$

$\textsc{ver}(m, t)$:
    $y := H(m)$
    return $t \overset{?}{=} \textsf{MAC}(k, y)$
```

The starting point is $\mathcal{L}^{HtM}_{\textsf{mac-real}}$, shown here with the details of HtM filled in. Our goal is to eventually reach $\mathcal{L}_{\textsf{mac-fake}}$, where the VER subroutine returns `false` unless (m, t) was generated by the GETMAC subroutine.

```
$k \leftarrow \textsf{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$

$\textsc{getmac}(m)$:
    $y := H(m)$
    $t := \textsf{MAC}(k, y)$
    $\mathcal{T} := \mathcal{T} \cup \{(y, t)\}$
    return $t$

$\textsc{ver}(m, t)$:
    $y := H(m)$
    return $(y, t) \overset{?}{\in} \mathcal{T}$
```

We have applied the MAC security of M, omitting the usual details (factor out, swap libraries, inline). Now VER returns `false` unless $(H(m), t) \in \mathcal{T}$.

```
$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$
$H^{-1} := \text{empty}$

GETMAC($m$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  $t := \text{MAC}(k,h)$
  $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
  return $t$

VER($m,t$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := y$
  return $(y,t) \overset{?}{\in} \mathcal{T}$
```

Here we have applied the security of the hash family $\mathcal{H}$. We have factored out all calls to H in terms of $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$, replaced $\mathcal{L}_{\text{cr-real}}$ with $\mathcal{L}_{\text{cr-fake}}$, and then inlined.

```
$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$; $\mathcal{T}' := \emptyset$
$H^{-1} := \text{empty}$

GETMAC($m$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  $t := \text{MAC}(k,y)$
  $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
  $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
  return $t$

VER($m,t$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  return $(y,t) \overset{?}{\in} \mathcal{T}$
```

Now we have simply added another set $\mathcal{T}'$ which is never actually used. We point out two things: First, in GETMAC, (y,t) added to $\mathcal{T}$ if and only if $(H^{-1}[y],t)$ is added to $\mathcal{T}'$. Second, if the last line of VER is reached, then the library has not self destructed. So $H^{-1}[y] = m$, and in fact $H^{-1}[y]$ has never been defined to be anything else. This means the last line of VER is equivalent to:

$$(y,t) \in \mathcal{T} \Leftrightarrow (H^{-1}[y],t) \in \mathcal{T}' \Leftrightarrow (m,t) \in \mathcal{T}'.$$

```
$k \leftarrow$ KeyGen
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$; $\mathcal{T}' := \emptyset$
$H^{-1} :=$ empty

GETMAC($m$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  $t := \text{MAC}(k, y)$
  $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
  $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
  return $t$

VER($m,t$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  return $(m,t) \overset{?}{\in} \mathcal{T}'$
```

We have replaced the condition $(H(m),t) \in \mathcal{T}$ with $(m,t) \in \mathcal{T}'$. But we just argued that these statements are logically equivalent within the library.

```
$k \leftarrow$ KeyGen
$H \leftarrow \mathcal{H}$
$\mathcal{T}' := \emptyset$

GETMAC($m$):
  $y := H(m)$
  $t := \text{MAC}(k, y)$
  $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
  return $t$

VER($m,t$):
  return $(m,t) \overset{?}{\in} \mathcal{T}'$
```

We remove the variable H^{-1} and **self destruct** statements via a standard sequence of changes (*i.e.*, factor out in terms of $\mathcal{L}_{\text{cr-fake}}$, replace with $\mathcal{L}_{\text{cr-real}}$, inline). We also remove the now-unused variable $\mathcal{T}$. The result is $\mathcal{L}^{HtM}_{\text{mac-fake}}$, as desired.

The next-to-last hybrid is the key step where collision-resistance comes into our reasoning. Indeed, a collision would break down the argument. If the adversary manages to find a collision $y = H(m) = H(m')$, then it could be that $(y,t) \in \mathcal{T}$ and $(m,t) \in \mathcal{T}'$ but $(m',t) \notin \mathcal{T}'$. This corresponds to a forgery of HtM in a natural way: Ask for a MAC of m, which is $t = \text{MAC}(k, H(m))$; then t is also a valid MAC of m'. ■

12.3 Merkle-Damgård Construction

Constructing a hash function seems like a challenging task, especially given that it must accept strings of arbitrary length as input. In this section, we'll see one approach for constructing hash functions, called the Merkle-Damgård construction.

Instead of a full-fledged hash function, imagine that we had a collision-resistant function (family) whose inputs were of a single fixed length, but longer than its outputs. In other words, suppose we had a family $\mathcal{H}$ of functions $h : \{0,1\}^{n+t} \to \{0,1\}^n$, where $t > 0$. We call such an h a **compression function**. This is not compression in the usual sense of data compression — we are not concerned about recovering the input from the output. We call it a compression function because it "compresses" its input by t bits (analogous to how a pseudorandom generator "stretches" its input by some amount).

We can apply the standard definition of collision-resistance to a family of *compression* functions, by restricting our interest to inputs of length exactly $n + t$. The functions in the family are not defined for any other input length.

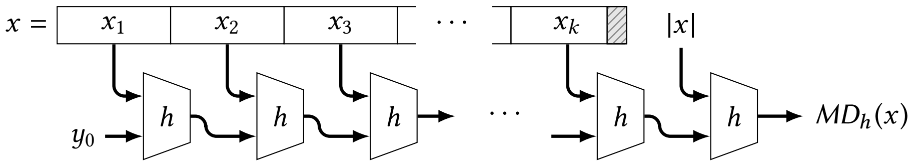
The following construction is one way to extend a compression function into a full-fledged hash function accepting arbitrary-length inputs:

Construction 12.4 (Merkle-Damgård)

*Let $h : \{0,1\}^{n+t} \to \{0,1\}^n$ be a compression function. Then the **Merkle-Damgård transformation** of h is $MD_h : \{0,1\}^* \to \{0,1\}^n$, where:*

```
MDPAD_t(x)
  ℓ := |x|, as length-t binary number
  while |x| not a multiple of t:
    x := x‖0
  return x‖ℓ
```

```
MD_h(x):
  x_1‖···‖x_{k+1} := MDPAD_t(x)
  // each x_i is t bits
  y_0 := 0^n
  for i = 1 to k + 1:
    y_i := h(y_{i-1}‖x_i)
  output y_{k+1}
```



The idea of the Merkle-Damgård construction is to split the input x into blocks of size t. The end of the string is filled out with 0s if necessary. A final block called the "padding block" is added, which encodes the (original) length of x in binary.

We are presenting a simplified version, in which MD_h accepts inputs whose maximum length is $2^t - 1$ bits (the length of the input must fit into t bits). By using multiple padding blocks (when necessary) and a suitable encoding of the original string length, the construction can be made to accomodate inputs of arbitrary length (see the exercises).

The value y_0 is called the **initialization vector** (IV), and it is a hard-coded part of the algorithm. In practice, a more "random-looking" value is used as the initialization vector. Or one can think of the Merkle-Damgård construction as defining a **family** of hash functions, corresponding to the different choices of IV.

Claim 12.5

Let $\mathcal{H}$ be a family of compression functions, and define $MD_{\mathcal{H}} = \{MD_h \mid h \in \mathcal{H}\}$ (a family of hash functions). If $\mathcal{H}$ is collision-resistant, then so is $MD_{\mathcal{H}}$.

Proof

While the proof can be carried out in the style of our library-based security definitions, it's actually much easier to simply show the following: given any collision under MD_h, we can efficiently find a collision under h. This means that any successful adversary violating the collision-resistance of $\mathsf{MD}_{\mathcal{H}}$ can be transformed into a successful adversary violating the collision resistance of $\mathcal{H}$. So if $\mathcal{H}$ is collision-resistant, then so is $\mathsf{MD}_{\mathcal{H}}$.

Suppose that x, x' are a collision under MD_h. Define the values $x_1, \ldots, x_{k+1}$ and $y_1, \ldots, y_{k+1}$ as in the computation of $\mathsf{MD}_h(x)$. Similarly, define $x'_1, \ldots, x'_{k'+1}$ and $y'_1, \ldots, y'_{k'+1}$ as in the computation of $\mathsf{MD}_h(x')$. Note that, in general, k may not equal k'.

Recall that:

$$\mathsf{MD}_h(x) = y_{k+1} = h(y_k \| x_{k+1})$$
$$\mathsf{MD}_h(x') = y'_{k'+1} = h(y'_{k'} \| x'_{k'+1})$$

Since we are assuming $\mathsf{MD}_h(x) = \mathsf{MD}_h(x')$, we have $y_{k+1} = y'_{k'+1}$. We consider two cases:

Case 1: If $|x| \neq |x'|$, then the padding blocks x_{k+1} and $x'_{k'+1}$ which encode $|x|$ and $|x'|$ are not equal. Hence we have $y_k \| x_{k+1} \neq y'_{k'} \| x'_{k'+1}$, so $y_k \| x_{k+1}$ and $y'_{k'} \| x'_{k'+1}$ are a collision under h and we are done.

Case 2: If $|x| = |x'|$, then x and x' are broken into the same number of blocks, so $k = k'$. Let us work backwards from the final step in the computations of $\mathsf{MD}_h(x)$ and $\mathsf{MD}_h(x')$. We know that:

$$\begin{array}{ccc} y_{k+1} & = & h(y_k \| x_{k+1}) \\ = & & \\ y'_{k+1} & = & h(y'_k \| x'_{k+1}) \end{array}$$

If $y_k \| x_{k+1}$ and $y'_k \| x'_{k+1}$ are not equal, then they are a collision under h and we are done. Otherwise, we can apply the same logic again to y_k and y'_k, which are equal by our assumption.

More generally, if $y_i = y'_i$, then either $y_{i-1} \| x_i$ and $y'_{i-1} \| x'_i$ are a collision under h (and we say we are "lucky"), or else $y_{i-1} = y'_{i-1}$ (and we say we are "unlucky"). We start with the premise that $y_k = y'_k$. Can we ever get "unlucky" every time, and not encounter a collision when propagating this logic back through the computations of $\mathsf{MD}_h(x)$ and $\mathsf{MD}_h(x')$? The answer is no, because encountering the unlucky case every time would imply that $x_i = x'_i$ for *all* i. That is, $x = x'$. But this contradicts our original assumption that $x \neq x'$. Hence we must encounter some "lucky" case and therefore a collision in h. ■

to-do *Discuss PGV constructions of compression functions from block ciphers. Will have to introduce ideal cipher model, though.*

12.4 Length-Extension Attacks

We showed that $\mathsf{MAC}(k, H(m))$ is a secure MAC, when MAC is a secure MAC for n-bit messages, and H is collision-resistant. A very tempting way to construct a MAC from a hash function is to simply let $H(k \| m)$ be the MAC of m under key k.

Unfortunately, this method turns out to be insecure in general (although in some special cases it may be safe). In particular, the method is insecure when H is public and constructed using the Merkle-Damgård approach. The key observation is that:

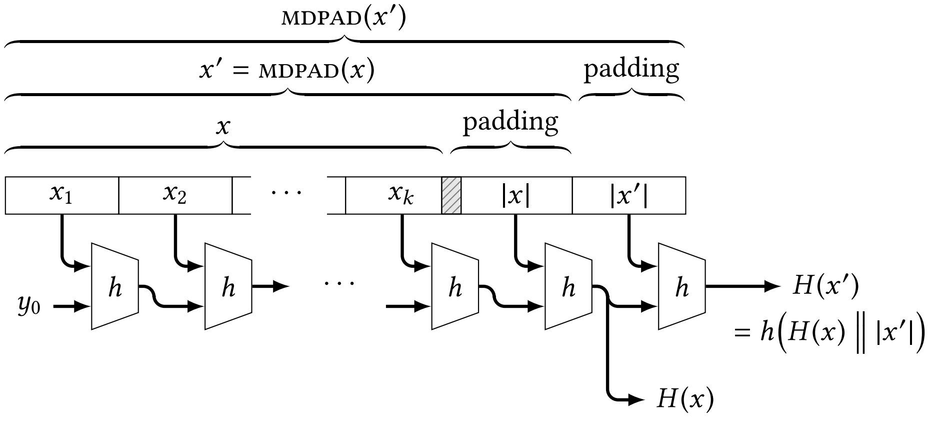


Figure 12.1: *Length-extension attack on the Merkle-Damgård construction*

> *knowing $H(x)$ allows you to predict the hash of any string that begins with $\textsc{mdpad}(x)$.*

In more detail, suppose H is a Merkle-Damgård hash function with compression function h. Imagine computing the hash function on both x and on $x' = \textsc{mdpad}(x)$ separately. The same sequence of blocks will be sent through the compression function, except that when computing $H(\textsc{mdpad}(x))$ we will also call the compression function on an additional padding block (encoding the length of x'). To compute the value of $H(\textsc{mdpad}(x))$, we don't need to know anything about x other than $H(x)$!

The idea applies to any string that begins with $\textsc{mdpad}(x)$. That is, given $H(x)$ it is possible to predict $H(z)$ for any z that begins with $\textsc{mdpad}(x)$. This property is called **length-extension** and it is a side-effect of the Merkle-Damgård construction.

Keep in mind several things:

- The MD construction is still collision-resistant. Length-extension does not help find collisions! We are not saying that x and $\textsc{mdpad}(x)$ have the *same* hash under H, only that knowing the hash of one allows you to predict the hash of the other.

- Length-extension works as long as the compression function h is public, even when x is a secret. As long as we know $H(x)$ and $|x|$, we can compute $\textsc{mdpad}(x)$ and predict the output of H on any string that has $\textsc{mdpad}(x)$ as a prefix.

Going back to the faulty MAC construction, suppose we take $t = H(k\|m)$ to be a MAC of m. For simplicity, assume that the length of $k\|m$ is a multiple of the block length. Knowing t, we can predict the MAC of $m\|z$, where z is the binary encoding of the length of $k\|m$ (i.e., $k\|m\|z = \textsc{mdpad}(k\|m)$). In particular, the MAC of $m\|z$ is $h(t\|z')$ where z' is the binary encoding of the length of $k\|m\|z$.

Importantly, no knowledge of the key k is required to predict the MAC of $m\|z$ given the MAC of m.

to-do *Work through an example*

to-do *Discuss alternatives to Merkle-Damgård. SHA-3 winner Keccak uses sponge construction and supports $H(k\|m)$ as a secure MAC by design.*

Exercises

12.1. Sometimes when I verify an MD5 hash visually, I just check the first few and the last few hex digits, and don't really look at the middle of the hash.

Generate two files with opposite meanings, whose MD5 hashes agree in their first 16 bits (4 hex digits) and in their last 16 bits (4 hex digits). It could be two text files that say opposite things. It could be an image of Mario and an image of Bowser. I don't know, be creative.

As an example, the strings "`subtitle illusive planes`" and "`wantings premises forego`" actually agree in the first 20 and last 20 bits (first and last 5 hex digits) of their MD5 hashes, but it's not clear that they're very meaningful.

```
$ echo -n "subtitle illusive planes" | md5sum
4188d4cdcf2be92a112bdb8ce4500243 -
$ echo -n "wantings premises forego" | md5sum
4188d209a75e1a9b90c6fe3efe300243 -
```

Describe how you generated the files, and how many MD5 evaluations you had to make.

12.2. Let $h : \{0,1\}^{n+t} \to \{0,1\}^n$ be a fixed-length compression function. Suppose we forgot a few of the important features of the Merkle-Damgård transformation, and construct a hash function H from h as follows:

- Let x be the input.
- Split x into pieces $y_0, x_1, x_2, \ldots, x_k$, where y_0 is n bits, and each x_i is t bits. The last piece x_k should be padded with zeroes if necessary.
- For $i = 1$ to k, set $y_i = h(y_{i-1}\|x_i)$.
- Output y_k.

Basically, it is similar to the Merkle-Damgård except we lost the IV and we lost the final padding block.

1. Describe an easy way to find two messages that are broken up into the same number of pieces, which have the same hash value under H.
2. Describe an easy way to find two messages that are broken up into different number of pieces, which have the same hash value under H. *Hint:* Pick any string of length $n + 2t$, then find a shorter string that collides with it.

Neither of your collisions above should involve finding a collision in h.

12.3. I've designed a hash function $H : \{0,1\}^* \to \{0,1\}^n$. One of my ideas is to make $H(x) = x$ if x is an n-bit string (assume the behavior of H is much more complicated on inputs of other lengths). That way, we know with certainty that there are no collisions among n-bit strings. Have I made a good design decision?

12.4. Let H be a hash function and let t be a fixed constant. Define $H^{(t)}$ as:

$$H^{(t)}(x) = \underbrace{H(\cdots H(H}_{t \text{ times}}(x))\cdots).$$

Show that if you are given a collision under $H^{(t)}$ then you can efficiently find a collision under H.

This means that if $\mathcal{H}$ is a collision-resistant hash family then $\mathcal{H}^{(t)} = \{H^{(t)} \mid H \in \mathcal{H}\}$ must also be collision-resistant.

12.5. In this problem, if x and y are strings of the same length, then we write $x \sqsubseteq y$ if $x = y$ or x comes before y in standard dictionary ordering.

Suppose a function $H : \{0,1\}^* \to \{0,1\}^n$ has the following property. For all strings x and y of the same length, if $x \sqsubseteq y$ then $H(x) \sqsubseteq H(y)$. Show that H is **not** collision resistant (describe how to efficiently find a collision in such a function).

Hint: Binary search, always recursing on a range that is *guaranteed* to contain a collision.

★ 12.6. Suppose a function $H : \{0,1\}^* \to \{0,1\}^n$ has the following property. For all strings x and y of the same length, $H(x \oplus y) = H(x) \oplus H(y)$. Show that H is **not** collision resistant (describe how to efficiently find a collision in such a function).

12.7. Generalize the Merkle-Damgård construction so that it works for arbitrary input lengths (and arbitrary values of t in the compression function).

12.8. Let F be a secure PRF with n-bit inputs, and let $\mathcal{H}$ be a collision-resistant hash function family with n-bit outputs. Define the new function $F'((k,H),x) = F(k,H(x))$, where we interpret (k,H) to be its key ($H \in \mathcal{H}$). Prove that F' is a secure PRF with arbitrary-length inputs.

12.9. More exotic issues with the Merkle-Damgård construction:

(a) Let H be a hash function with n-bit output, based on the Merkle-Damgård construction. Show how to compute (with high probability) 4 messages that all hash to the same value under H, using only $\sim 2 \cdot 2^{n/2}$ calls to H.

Hint: The 4 messages that collide will have the form $x\|y$, $x\|y'$, $x'\|y$ and $x'\|y'$. Use a length-extension idea and perform 2 birthday attacks.

(b) Show how to construct 2^d messages that all hash to the same value under H, using only $O(d \cdot 2^{n/2})$ evaluations of H.

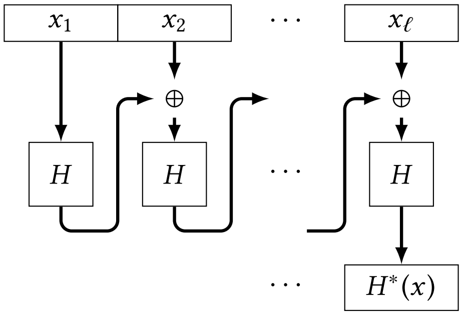
(c) Suppose H_1 and H_2 are (different) hash functions, both with n-bit output. Consider the function $H^*(x) = H_1(x)\|H_2(x)$. Since H^* has $2n$-bit output, it is tempting to think that finding a collision in H^* will take $2^{(2n)/2} = 2^n$ effort.

However, this intuition is not true when H_1 is a Merkle-Damgård hash. Show that when H_1 is Merkle-Damgård, then it is possible to find collisions in H^* with only $O(n2^{n/2})$ effort. The attack should assume nothing about H_2 (i.e., H_2 need not be Merkle-Damgård).

Hint: Applying part (b), first find a set of $2^{n/2}$ messages that all have the same hash under H_1. Among them, find 2 that also collide under H_2.

12.10. Let H be a collision-resistant hash function with output length n. Let H^* denote iterating H in a manner similar to CBC-MAC:

```
$\underline{H^*(x_1 \cdots x_\ell)}$:
  // each $x_i$ is $n$ bits
  $y_0 := 0^n$
  for $i = 1$ to $\ell$:
    $y_i := H(x_i \oplus y_{i-1})$
  return $y_i$
```



Show that H^* is **not** collision-resistant. Describe a successful attack.

13 The RSA Function

RSA was among the first public-key cryptography developed. It was first described in 1978, and is named after its creators, Ron Rivest, Adi Shamir, and Len Adleman.[1] Although "textbook" RSA by itself is not a secure encryption scheme, it is a fundamental ingredient for public-key cryptography.

13.1 Modular Arithmetic & Number Theory

In general, public-key cryptography relies on computational problems from abstract algebra. Of the techniques currently known for public-key crypto, RSA uses some of the simplest mathematical ideas, so it's an ideal place to start.

We will be working with modular arithmetic, so please review the section on modular arithmetic from the first lecture! We need to understand the behavior of the four basic arithmetic operations in the set $\mathbb{Z}_n = \{0, \ldots, n-1\}$.

Every element $x \in \mathbb{Z}_n$ has an inverse with respect to addition mod n: namely $-x \mathbin{\%} n$. For example, the additive inverse of 11 mod 14 is $-11 \equiv_{14} 3$. However, multiplicative inverses are not so straight-forward.

Greatest Common Divisors

If $d \mid x$ and $d \mid y$, then d is a **common divisor** of x and y. The largest possible such d is called the **greatest common divisor (GCD),** denoted $\gcd(x, y)$. If $\gcd(x, y) = 1$, then we say that x and y are **relatively prime**. The oldest "algorithm" ever documented is the one Euclid described for computing GCDs (ca. 300 BCE):

```
GCD($x, y$): // Euclid's algorithm
  if $y = 0$ then return $x$
  else return GCD($y, x \% y$)
```

Multiplicative Inverses

We let $\mathbb{Z}_n^*$ denote the set $\{x \in \mathbb{Z}_n \mid \gcd(x, n) = 1\}$, the **multiplicative group modulo** n. This group is *closed under multiplication mod n*, which just means that if $x, y \in \mathbb{Z}_n^*$ then $xy \in \mathbb{Z}_n^*$, where xy denotes multiplication mod n. Indeed, if $\gcd(x, n) = \gcd(y, n) = 1$, then $\gcd(xy, n) = 1$ and thus $\gcd(xy \mathbin{\%} n, n) = 1$ by Euclid's algorithm.

In abstract algebra, a *group* is a set that is closed under its operation (in this case multiplication mod n), and is also closed under inverses. So if $\mathbb{Z}_n^*$ is really a group under

[1] Clifford Cocks developed an equivalent scheme in 1973, but it was classified since he was working for British intelligence.

multiplication mod n, then for every $x \in \mathbb{Z}_n^*$ there must be a $y \in \mathbb{Z}_n^*$ so that $xy \equiv_n 1$. In other words, y is the **multiplicative inverse** of x (and we would give it the name x^{-1}.

The fact that we can always find a multiplicative inverse for elements of $\mathbb{Z}_n^*$ is due to the following theorem:

Theorem 13.1 (Bezout's Theorem) *For all integers x and y, there exist integers a and b such that $ax + by = \gcd(x,y)$. In fact, $\gcd(x,y)$ is the smallest positive integer that can be written as an integral linear combination of x and y.*

What does this have to do with multiplicative inverses? Take any $x \in \mathbb{Z}_n^*$; we will show how to find its multiplicative inverse. Since $x \in \mathbb{Z}_n^*$, we have $\gcd(x,n) = 1$. From Bezout's theorem, there exist integers a, b satisfying $ax + bn = 1$. By reducing both sides of this equation modulo n, we have

$$1 \equiv_n ax + bn \equiv_n ax + 0$$

(since $bn \equiv_n 0$). Thus the integer a guaranteed by Bezout's theorem is the multiplicative inverse of x modulo n.

We have shown that every $x \in \mathbb{Z}_n^*$ has a multiplicative inverse mod n. That is, if $\gcd(x,n) = 1$, then x has a multiplicative inverse. But might it be possible for x to have a multiplicative inverse mod n even if $\gcd(x,n) \neq 1$?

Suppose that we have an element x with a multiplicative inverse; that is, $xx^{-1} \equiv_n 1$. Then n divides $xx^{-1} - 1$, so we can write $xx^{-1} - 1 = kn$ (as an expression over the integers) for some integer k. Rearranging, we have that $xx^{-1} - kn = 1$. That is to say, we have a way to write 1 as an integral linear combination of x and n. From Bezout's theorem, this must mean that $\gcd(x,n) = 1$. Hence, $x \in \mathbb{Z}_n^*$. We conclude that:

$$\mathbb{Z}_n^* \overset{\text{def}}{=} \{x \in \mathbb{Z}_n \mid \gcd(x,n) = 1\} = \{x \in \mathbb{Z}_n \mid \exists y \in \mathbb{Z}_n : xy \equiv_n 1\}.$$

The elements of $\mathbb{Z}_n^*$ are *exactly* those elements with a multiplicative inverse mod n.

Furthermore, multiplicative inverses can be computed efficiently using an extended version of Euclid's GCD algorithm. While we are computing the GCD, we can also keep track of integers a and b from Bezout's theorem at every step of the recursion; see below:

```
EXTGCD($x, y$):
  // returns $(d, a, b)$ such that $\gcd(x,y) = d = ax + by$
  if $y = 0$:
    return $(x, 1, 0)$
  else:
    $(d, a, b)$ := EXTGCD($y, x \% y$)
    return $(d, b, a - b\lfloor x/y \rfloor)$
```

Example *Below is a table showing the computation of* EXTGCD$(35, 144)$. *Note that the columns x, y are computed from the top down (as recursive calls to* EXTGCD *are made), while the columns d, a, and b are computed from bottom up (as recursive calls return). Also note that in each row, we indeed have $d = ax + by$.*

x	y	$\lfloor \frac{x}{y} \rfloor$	$x \% y$	d	a	b
35	144	0	35	1	**-37**	9
144	35	4	4	1	9	-37
35	4	8	3	1	-1	9
4	3	1	1	1	1	-1
3	1	3	0	1	0	1
1	0	-	-	1	1	0

The final result demonstrates that $35^{-1} \equiv_{144} -37 \equiv_{144} 107$.

The Totient Function

Euler's **totient** function is defined as $\phi(n) \overset{\text{def}}{=} |\mathbb{Z}_n^*|$, in other words, the number of elements of Z_n which are relatively prime to n.

As an example, if p is a prime, then $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$ because every integer in $\mathbb{Z}_n$ apart from zero is relatively prime to p. Therefore, $\phi(p) = p - 1$.

We will frequently work modulo n where n is the product of two distinct primes $n = pq$. In that case, $\phi(n) = (p-1)(q-1)$. To see why, let's count how many elements in $\mathbb{Z}_{pq}$ share a common divisor with pq (i.e., are *not* in $\mathbb{Z}_{pq}^*$).

- The multiples of p share a common divisor with pq. These include $0, p, 2p, 3p, \ldots, (q-1)p$. There are q elements in this list.
- The multiples of q share a common divisor with pq. These include $0, q, 2q, 3q, \ldots, (p-1)q$. There are p elements in this list.

We have clearly double-counted element 0 in these lists. But no other element is double counted. Any item that occurs in both lists would be a common multiple of both p and q, but the least common multiple of p and q is pq since p and q are relatively prime. But pq is larger than any item in these lists.

We count $p+q-1$ elements in $\mathbb{Z}_{pq}$ which share a common divisor with pq. That leaves the rest to reside in $\mathbb{Z}_{pq}^*$, and there are $pq - (p+q-1) = (p-1)(q-1)$ of them. Hence $\phi(pq) = (p-1)(q-1)$.

General formulas for $\phi(n)$ exist, but they typically rely on knowing the prime factorization of n. We will see more connections between the difficulty of computing $\phi(n)$ and the difficulty of factoring n later in this part of the course.

Here's an important theorem from abstract algebra:

Theorem 13.2 (Euler's Theorem) *If* $x \in \mathbb{Z}_n^*$ *then* $x^{\phi(n)} \equiv_n 1$.

As a final corollary, we can deduce Fermat's "little theorem," that $x^p \equiv_p x$ for all x, when p is prime.[2]

[2] You have to handle the case of $x \equiv_p 0$ separately, since $0 \notin \mathbb{Z}_p^*$ so Euler's theorem doesn't apply to it.

13.2 The RSA Function

The RSA function is defined as follows:

- Let p and q be distinct primes (later we will say more about how they are chosen), and let $N = pq$. N is called the **RSA modulus**.
- Let e and d be integers such that $ed \equiv_{\phi(N)} 1$. That is, e and d are multiplicative inverses mod $\phi(N)$ — not mod N! e is called the **encryption exponent**, and d is called the **decryption exponent**. These names are historical, but not entirely precise since RSA by itself does not achieve CPA security.
- The RSA function is: $m \mapsto m^e \,\%\, N$, where $m \in \mathbb{Z}_N$.
- The inverse RSA function is: $c \mapsto c^d \,\%\, N$, where $c \in \mathbb{Z}_N$.

Essentially, the RSA function (and its inverse) is a simple modular exponentiation. The most confusing thing to remember about RSA is that e and d "live" in $\mathbb{Z}^*_{\phi(N)}$, while m and c "live" in $\mathbb{Z}_N$.

Let's make sure the function we called the "inverse RSA function" is actually in inverse of the RSA function. The RSA function raises its input to the e power, and the inverse RSA function raises its input to the d power. So it suffices to show that raising to the ed power has no effect modulo N.

Since $ed \equiv_{\phi(N)} 1$, we can write $ed = t\phi(N) + 1$ for some integer t. Then:

$$(m^e)^d = m^{ed} = m^{t\phi(N)+1} = (m^{\phi(N)})^t m \equiv_N 1^t m = m$$

Note that we have used the fact that $m^{\phi(N)} \equiv_N 1$ from Euler's theorem.

to-do *Discuss computational aspects of modular exponentiation, and in general remind readers that efficiency of numerical algorithms is measured in terms of the* number of bits *needed to write the input.*

Security Properties

In these notes we will not formally define a desired security property for RSA. Roughly speaking, the idea is that even when N and e can be made public, it should be hard to compute the operation $c \mapsto c^d \,\%\, N$. In other words, the RSA function $m \mapsto m^e \,\%\, N$ is:

- easy to compute given N and e
- hard to invert given N and e but not d
- easy to invert given d

to-do *more details*

13.3 Chinese Remainder Theorem

The multiplicative group $\mathbb{Z}_N^*$ has some interesting structure when N is the product of distinct primes. We can use this structure to optimize some algorithms related to RSA.

History. Some time around the 4th century CE, Chinese mathematician Sun Tzu in his book *Sun Tze Suan Ching* discussed problems relating to simultaneous equations of modular arithmetic:

> *"We have a number of things, but we do not know exactly how many. If we count them by threes we have two left over. If we count them by fives we have three left over. If we count them by sevens we have two left over. How many things are there?"*[3]

In our notation, he is asking for a solution x to the following system of equations:

$$\begin{aligned} x &\equiv_3 2 \\ x &\equiv_5 3 \\ x &\equiv_7 2 \end{aligned}$$

A generalized way to solve equations of this kind was later given by mathematician Qin Jiushao in 1247 CE. For our eventual application to RSA, we will only need to consider the case of two simultaneous equations.

Theorem 13.3 (CRT) Suppose $\gcd(r,s) = 1$. Then for all integers u,v, there is a solution for x in the following system of equations:

$$\begin{aligned} x &\equiv_r u \\ x &\equiv_s v \end{aligned}$$

Furthermore, this solution is *unique* modulo rs.

Proof Since $\gcd(r,s) = 1$, we have by Bezout's theorem that $1 = ar + bs$ for some integers a and b. Furthermore, b and s are multiplicative inverses modulo r. Now choose $x = var + ubs$. Then,

$$x = var + ubs \equiv_r (va)0 + u(s^{-1}s) = u$$

So $x \equiv_r u$, as desired. By a symmetric argument, we can see that $x \equiv_s v$, so x is a solution to the system of equations.

Now we argue why the solution is *unique* modulo rs. Suppose x and x' are two solutions to the system of equations, so we have:

$$\begin{aligned} x &\equiv_r x' \equiv_r u \\ x &\equiv_s x' \equiv_s v \end{aligned}$$

[3] Translation due to Joseph Needham, *Science and Civilisation in China, vol. 3: Mathematics and Sciences of the Heavens and Earth*, 1959.

Since $x \equiv_r x'$ and $x \equiv_s x'$, it must be that $x - x'$ is a multiple of r and a multiple of s. Since r and s are relatively prime, their least common multiple is rs, so $x - x'$ must be a multiple of rs. Hence, $x \equiv_{rs} x'$. So any two solutions to this system of equations are congruent mod rs. ■

We can associate every pair $(u,v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ with its corresponding system of equations of the above form (with u and v as the right-hand-sides). The CRT suggests a relationship between these pairs $(u,v) \in \mathbb{Z}_r \times \mathbb{Z}_s$ and elements of $\mathbb{Z}_{rs}$.

For $x \in \mathbb{Z}_{rs}$, and $(u,v) \in \mathbb{Z}_r \times \mathbb{Z}_s$, let us write

$$x \overset{\text{crt}}{\longleftrightarrow} (u,v)$$

to mean that x is a solution to $x \equiv_r u$ and $x \equiv_s v$. The CRT says that the $\overset{\text{crt}}{\longleftrightarrow}$ relation is a *bijection* (1-to-1 correspondence) between elements of $\mathbb{Z}_{rs}$ and elements of $\mathbb{Z}_r \times \mathbb{Z}_s$.

In fact, the relationship is even deeper than that. Consider the following observations:

1. If $x \overset{\text{crt}}{\longleftrightarrow} (u,v)$ and $x' \overset{\text{crt}}{\longleftrightarrow} (u',v')$, then $x + x' \overset{\text{crt}}{\longleftrightarrow} (u+u', v+v')$. You can see this by adding relevant equations together from the system of equations. Note here that the addition $x + x'$ is done mod rs; the addition $u + u'$ is done mod r; and the addition $v + v'$ is done mod s.
2. If $x \overset{\text{crt}}{\longleftrightarrow} (u,v)$ and $x' \overset{\text{crt}}{\longleftrightarrow} (u',v')$, then $xx' \overset{\text{crt}}{\longleftrightarrow} (uu', vv')$. You can see this by multiplying relevant equations together from the system of equations. As above, the multiplication xx' is mod rs; uu' is done mod r; vv' is done mod s.
3. Suppose $x \overset{\text{crt}}{\longleftrightarrow} (u,v)$. Then $\gcd(x, rs) = 1$ if and only if $\gcd(u,r) = \gcd(v,s) = 1$. In other words, the $\overset{\text{crt}}{\longleftrightarrow}$ relation is a 1-to-1 correspondence between elements of $\mathbb{Z}^*_{rs}$ and elements of $\mathbb{Z}^*_r \times \mathbb{Z}^*_s$.[4]

The bottom line is that the CRT demonstrates that $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ **are essentially the same mathematical object.** In the terminology of abstract algebra, the two structures are *isomorphic.*

Think of $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ being two different kinds of *names* or *encodings* for the same set of items. If we know the "$\mathbb{Z}_{rs}$-names" of two items, we can add them (mod rs) to get the $\mathbb{Z}_{rs}$-name of the result. If we know the "$\mathbb{Z}_r \times \mathbb{Z}_s$-names" of two items, we can add them (first components mod r and second components mod s) to get the $\mathbb{Z}_r \times \mathbb{Z}_s$-name of the result. The CRT says that both of these ways of adding give the same results.

Additionally, the proof of the CRT shows us how to convert between these styles of names for a given object. So given $x \in \mathbb{Z}_{rs}$, we can compute $(x \,\%\, r, x \,\%\, s)$, which is the corresponding element/name in $\mathbb{Z}_r \times \mathbb{Z}_s$. Given $(u,v) \in \mathbb{Z}_r \times \mathbb{Z}_s$, we can compute $x = var + ubs \,\%\, rs$ (where a and b are computed from the extended Euclidean algorithm) to obtain the corresponding element/name $x \in \mathbb{Z}_{rs}$.

From a **mathematical** perspective, $\mathbb{Z}_{rs}$ and $\mathbb{Z}_r \times \mathbb{Z}_s$ are the same object. However, from a **computational** perspective, there might be reason to favor one over the other. In fact, it turns out that doing computations in the $\mathbb{Z}_r \times \mathbb{Z}_s$ realm is significantly cheaper.

[4] Fun fact: this yields an alternative proof that $\phi(pq) = (p-1)(q-1)$ when p and q are prime. That is, $\phi(pq) = |\mathbb{Z}^*_{pq}| = |\mathbb{Z}^*_p \times \mathbb{Z}^*_q| = (p-1)(q-1)$.

Application to RSA

In the context of RSA decryption, we are interested in taking $c \in \mathbb{Z}_{pq}$ and computing $c^d \in \mathbb{Z}_{pq}$. Since p and q are distinct primes, $\gcd(p,q) = 1$ and the CRT is in effect.

Thinking in terms of $\mathbb{Z}_{pq}$-arithmetic, raising c to the d power is rather straightforward. However, the CRT suggests that another approach is possible: We could convert c into its $\mathbb{Z}_p \times \mathbb{Z}_q$ representation, do the exponentiation under that representation, and then convert back into the $\mathbb{Z}_{pq}$ representation. This approach corresponds to the bold arrows in Figure 13.1, and the CRT guarantees that the result will be the same either way.

operations in $\mathbb{Z}_{pq}$: operations in $\mathbb{Z}_p \times \mathbb{Z}_q$:

$$\begin{tikzcd}
c \arrow[r, "\text{convert representation}"] \arrow[d, "\text{exponentiate in } \mathbb{Z}_{pq}"'] & (u,v) \arrow[d, "\text{exponentiate in } \mathbb{Z}_p \times \mathbb{Z}_q"] \\
c^d & (u^d, v^d) \arrow[l, "\text{convert representation}"]
\end{tikzcd}$$

Figure 13.1: *Two ways to compute $c \mapsto c^d$ in $\mathbb{Z}_{pq}$.*

Now why would we ever want to compute things this way? Performing an exponentiation modulo an n-bit number requires about n^3 steps. Let's suppose that p and q are each n bits long, so that the RSA modulus N is $2n$ bits long. Performing $c \mapsto c^d$ modulo N therefore costs about $(2n)^3 = 8n^3$ total.

The CRT approach involves two modular exponentiations — one mod p and one mod q. Each of these moduli are only n bits long, so the total cost is $n^3 + n^3 = 2n^3$. **The CRT approach is 4 times faster!** Of course, we are neglecting the cost of converting between representations, but that cost is very small in comparison to the cost of exponentiation.

It's worth pointing out that this speedup can only be done for the RSA *inverse* function. One must know p and q in order to exploit the Chinese Remainder Theorem, and only the party performing the RSA inverse function typically knows this.

13.4 The Hardness of Factoring N

Clearly the hardness of RSA is related to the hardness of factoring the modulus N. Indeed, if you can factor N, then you can compute $\phi(N)$, solve for d, and easily invert RSA. So factoring must be *at least as hard as* inverting RSA.

Factoring integers (or, more specifically, factoring RSA moduli) is believed to be a hard problem for classical computers.[5] In this section we show that some other problems related to RSA are "as hard as factoring." What does it mean for a computational problem to be "as hard as factoring?" More formally, in this section we will show the following:

Theorem 13.4 *Either **all** of the following problems can be solved in polynomial-time, or **none** of them can:*

[5] A polynomial-time algorithm for factoring is known for quantum computers.

1. *Given an RSA modulus $N = pq$, compute its factors p and q.*
2. *Given an RSA modulus $N = pq$ compute $\phi(N) = (p-1)(q-1)$.*
3. *Given an RSA modulus $N = pq$ and value e, compute its inverse d, where $ed \equiv_{\phi(N)} 1$.*
4. *Given an RSA modulus $N = pq$, find any $x \not\equiv_N \pm 1$ such that $x^2 \equiv_N 1$.*

To prove the theorem, we will show:

- *if* there is an efficient algorithm for (1), *then* we can use it as a subroutine to construct an efficient algorithm for (2). This is straight-forward: if you have a subroutine factoring N into p and q, then you can call the subroutine and then compute $(p-1)(q-1)$.
- *if* there is an efficient algorithm for (2), *then* we can use it as a subroutine to construct an efficient algorithm for (3). This is also straight-forward: if you have a subroutine computing $\phi(N)$ given N, then you can compute the multiplicative inverse of e using the extended Euclidean algorithm.
- *if* there is an efficient algorithm for (3), *then* we can use it as a subroutine to construct an efficient algorithm for (4).
- *if* there is an efficient algorithm for (4), *then* we can use it as a subroutine to construct an efficient algorithm for (1).

Below we focus on the final two implications.

Using square roots of unity to factor N

Problem (4) of Theorem 13.4 concerns a new concept known as square roots of unity:

Definition 13.5 (Sqrt of unity) *x is a **square root of unity modulo** N if $x^2 \equiv_N 1$. If $x \not\equiv_N 1$ and $x \not\equiv_N -1$, then we say that x is a **non-trivial** square root of unity.*

Note that ± 1 are always square roots of unity modulo N, for any N ($(\pm 1)^2 = 1$ over the integers, so it is also true mod N). But if N is the product of distinct odd primes, then N has 4 square roots of unity: two trivial and two non-trivial ones (see the exercises in this chapter).

Claim 13.6 *Suppose there is an efficient algorithm for computing nontrivial square roots of unity modulo N. Then there is an efficient algorithm for factoring N. (This is the (4) ⇒ (1) step in Theorem 13.4.)*

Proof The reduction is rather simple. Suppose NTSRU is an algorithm that on input N returns a non-trivial square root of unity modulo N. Then we can factor N with the following algorithm:

```
FACTOR($N$):
  $x$ := NTSRU($N$)
  return gcd($N, x+1$) and gcd($N, x-1$)
```

The algorithm is simple, but we must argue that it is correct. When x is a nontrivial square root of unity modulo N, we have the following:

$$\begin{array}{rll} x^2 \equiv_{pq} 1 & \Rightarrow pq \mid x^2 - 1 & \Rightarrow pq \mid (x+1)(x-1) \\ x \not\equiv_{pq} 1 & & \Rightarrow pq \nmid (x-1) \\ x \not\equiv_{pq} -1 & & \Rightarrow pq \nmid (x+1) \end{array}$$

So the prime factorization of $(x+1)(x-1)$ contains a factor of p and a factor of q. But neither $x+1$ nor $x-1$ contain factors of *both* p and q. Hence $x+1$ and $x-1$ must each contain factors of exactly one of $\{p,q\}$, and $\{\gcd(pq, x-1), \gcd(pq, x+1)\} = \{p,q\}$. ■

Finding square roots of unity

Claim 13.7 *If there is an efficient algorithm for computing $d \equiv_{\phi(N)} e^{-1}$ given N and e, then there is an efficient algorithm for computing nontrivial square roots of unity modulo N. (This is the (3) ⇒ (4) step in Theorem 13.4.)*

Proof Suppose we have an algorithm FIND_D that on input (N,e) returns the corresponding exponent d. Then consider the following algorithm which uses FIND_D as a subroutine:

```
SRU($N$):
  choose $e$ as a random $n$-bit prime
  $d$ := FIND_D($N, e$)
  write $ed - 1 = 2^s r$, with $r$ odd
  // i.e., factor out as many 2s as possible
  $w \leftarrow \mathbb{Z}_N$
  if $\gcd(w, N) \neq 1$: // $w \notin \mathbb{Z}_N^*$
    use $\gcd(w, N)$ to factor $N = pq$
    compute a nontrivial square root of unity using $p$ & $q$
  $x := w^r \% N$
  if $x \equiv_N 1$ then return 1
  for $i = 0$ to $s$:
    if $x^2 \equiv_N 1$ then return $x$
    $x := x^2 \% N$
```

There are several return statements in this algorithm, and it should be clear that all of them indeed return a square root of unity. Furthermore, the algorithm does eventually return within the main for-loop, because x takes on the sequence of values:

$$w^r, w^{2r}, w^{4r}, w^{8r}, \ldots, w^{2^s r}$$

and the final value of that sequence satisfies

$$w^{2^s r} = w^{ed-1} \equiv_N w^{(ed-1)\%\phi(N)} = w^{1-1} = 1.$$

Conditioned on $w \in \mathbb{Z}_N^*$, it is possible to show that the algorithm returns a square root of unity *chosen uniformly at random* from among the four possible square roots of unity.

So with probability 1/2, the output is a nontrivial square root. We can repeat this basic process n times, and eventually encounter a nontrivial square root of unity with probability $1 - 2^{-n}$. ■

to-do *more complete analysis*

13.5 Malleability of RSA, and Applications

We now discuss several surprising problems that turn out to be equivalent to the problem of inverting RSA. The results in this section rely on the following *malleability* property of RSA: Suppose you are given $c = m^e$ for an unknown message m. Assuming e is public, you can easily compute $c \cdot x^e = (mx)^e$. In other words, given the RSA function applied to m, it is possible to obtain the RSA function applied to a related message mx.

Inverting RSA on a small subset

Suppose you had a subroutine $\textsc{invert}(N, e, c)$ that inverted RSA (*i.e.*, returned $c^d \bmod N$) but only for, say, 1% of all possible c's. That is, there exists some subset $G \subseteq \mathbb{Z}_N$ with $|G| \geqslant N/100$, such that for all $m \in G$ we have $m = \textsc{invert}(N, e, m^e)$.

If you happen to have a value $c = m^e$ for $m \notin G$, then it's not so clear how useful such a subroutine INVERT could be to you. However, it turns out that the subroutine can be used to invert RSA on *any input whatsoever.* Informally, if inverting RSA is easy on 1% of inputs, then inverting RSA is easy *everywhere.*

Assuming that we have such an algorithm INVERT, then this is how we can use it to invert RSA on any input:

```
REALLYINVERT($(N, e, c)$):
  do:
    $r \leftarrow \mathbb{Z}_N$
    $c' := c \cdot r^e \% N$
    $m' :=$ INVERT$(N, e, c')$
    $m := m' \cdot r^{-1}$
  repeat if $m^e \not\equiv_N c$
  return $m$
```

Suppose the input to REALLYINVERT involves $c = (m^*)^e$ for some unknown m^*. The goal is to output m^*.

In the main loop, c' is constructed to be an RSA encoding of $m^* \cdot r$. Since r is uniformly distributed in $\mathbb{Z}_N$, so is $m^* \cdot r$. So the probability of $m^* \cdot r$ being in the "good set" G is 1%. Furthermore, when it is in the good set, INVERT correctly returns $m^* \cdot r$. And in that case, REALLYINVERT outputs the correct answer m^*.

Each time through the main loop incurs a 1% chance of successfully inverting the given c. Therefore the expected running time of REALLYINVERT is $1/0.01 = 100$ times through the main loop.

Determining high-order bits of m

Consider the following problem: Given $c = m^e \bmod N$ for an unknown m, determine whether $m > N/2$ or $m < N/2$. That is, does m live in the top half or bottom half of $\mathbb{Z}_N$?

We show a surprising result that even this limited amount of information is enough to completely invert RSA. Equivalently, if inverting RSA is hard, then it is not possible to tell whether m is in the top half or bottom half of $\mathbb{Z}_N$ given $m^e \,\%\, N$.

The main idea is that we can do a kind of binary search in $\mathbb{Z}_N$. Suppose $\textsc{tophalf}(N,e,c)$ is a subroutine that can tell whether $c^d \bmod N$ is in $\{0,\ldots,\frac{N-1}{2}\}$ or in $\{\frac{N+1}{2},\ldots,N-1\}$. Given a candidate c, we can call $\textsc{tophalf}$ to reduce the possible range of m from $\mathbb{Z}_N$ to either the top or bottom half. Consider the ciphertext $c' = c \cdot 2^e$, which encodes $2m$. We can use $\textsc{tophalf}$ to now determine whether $2m$ is in the top half of $\mathbb{Z}_N$. If $2m$ is in the top half of $\mathbb{Z}_N$, then m is in the top half of its current range. Using this approach, we can repeatedly query $\textsc{tophalf}$ to reduce the search space for m by half each time. In only $\log N$ queries we can uniquely identify m.

```
BSEARCH$(N,e,c)$:
  $lo := 0;\quad hi := N-1$
  for $i = 1$ to $\log N$:
    $mid := (hi + lo)/2$
    if TOPHALF$(N,e,c)$:
      $hi := mid$
    else:
      $lo := mid$
    $c := c \cdot 2^e$
  return $\lfloor hi \rfloor$
```

to-do *more complete analysis*

Exercises

13.1. Prove by induction the correctness of the $\textsc{extgcd}$ algorithm. That is, whenever $(d,a,b) = \textsc{extgcd}(x,y)$, we have $\gcd(x,y) = d = ax + by$. You may use the fact that the original Euclidean algorithm correctly computes the GCD.

13.2. Prove that if $g^a \equiv_n 1$ and $g^b \equiv_n 1$, then $g^{\gcd(a,b)} \equiv_n 1$.

13.3. Prove that $\gcd(2^a - 1, 2^b - 1) = 2^{\gcd(a,b)} - 1$.

13.4. Prove that $x^a \,\%\, n = x^{a \% \phi(n)} \,\%\, n$. In other words, when working modulo n, you can reduce exponents modulo $\phi(n)$.

13.5. In this problem we determine the efficiency of Euclid's GCD algorithm. Since its input is a pair of numbers (x,y), let's call $x + y$ the *size* of the input. Let F_k denote the kth Fibonacci number, using the indexing convention $F_0 = 1$; $F_1 = 2$. Prove that (F_k, F_{k-1})

is the smallest-*size* input on which Euclid's algorithm makes k recursive calls. *Hint:* Use induction on k.

Note that the *size* of input (F_k, F_{k-1}) is F_{k+1}, and recall that $F_{k+1} \approx \phi^{k+1}$, where $\phi \approx 1.618\ldots$ is the golden ratio. Thus, for any inputs of *size* $N \in [F_k, F_{k+1})$, Euclid's algorithm will make less than $k \leqslant \log_\phi N$ recursive calls. In other words, the worst-case number of recursive calls made by Euclid's algorithm on an input of *size* N is $O(\log N)$, which is linear in the number of bits needed to write such an input.[6]

13.6. Consider the following **symmetric-key** encryption scheme with plaintext space $\mathcal{M} = \{\texttt{0},\texttt{1}\}^\lambda$. To encrypt a message m, we "pad" m into a prime number by appending a zero and then random non-zero bytes. We then mulitply by the secret key. To decrypt, we divide off the key and then strip away the "padding."

The idea is that decrypting a ciphertext without knowledge of the secret key requires factoring the product of two large primes, which is a hard problem.

```
KeyGen:
    choose random $\lambda$-bit prime $k$
    return $k$

Dec($k, c$):
    $m' := c/k$
    while $m'$ not a multiple of 10:
        $m' := \lfloor m'/10 \rfloor$
    return $m'/10$

Enc($k, m \in \{\texttt{0},\texttt{1}\}^\lambda$):
    $m' := 10 \cdot m$
    while $m'$ not prime:
        $d \leftarrow \{1, \ldots, 9\}$
        $m' := 10 \cdot m' + d$
    return $k \cdot m'$
```

Show an attack breaking CPA-security of the scheme. That is, describe a distinguisher and compute its bias. *Hint:* ask for any two ciphertexts.

13.7. Explain why the RSA encryption exponent e must always be an odd number.

13.8. The Chinese Remainder Theorem states that there is always a solution for x in the following system of equations, when $\gcd(r,s) = 1$:

$$x \equiv_r u$$
$$x \equiv_s v$$

Give an example u, v, r, s, with $\gcd(r,s) \neq 1$ for which the equations have no solution. Explain why there is no solution.

13.9. Bob chooses an RSA plaintext $m \in \mathbb{Z}_N$ and encrypts it under Alice's public key as $c \equiv_N m^e$. To decrypt, Alice first computes $m_p \equiv_p c^d$ and $m_q \equiv_q c^d$, then uses the CRT conversion to obtain $m \in \mathbb{Z}_N$, just as expected. But suppose Alice is using faulty hardware, so that she computes a **wrong value** for m_q. The rest of the computation happens correctly, and Alice computes the (wrong) result $\hat{m}$. Show that, no matter what m is, and no matter what Alice's computational error was, Bob can factor N if he learns $\hat{m}$.

[6] A more involved calculation that incorporates the cost of each division (modulus) operation shows the worst-case overall efficiency of the algorithm to be $O(\log^2 N)$ — quadratic in the number of bits needed to write the input.

Hint: Bob knows m and $\hat{m}$ satisfying the following:

$$\begin{aligned} m &\equiv_p \hat{m} \\ m &\not\equiv_q \hat{m} \end{aligned}$$

13.10. (a) Show that given an RSA modulus N and $\phi(N)$, it is possible to factor N easily. *Hint:* you have two equations (involving $\phi(N)$ and N) and two unknowns (p and q).

(b) Write a `pari` function that takes as input an RSA modulus N and $\phi(N)$ and factors N. Use it to factor the following 2048-bit RSA modulus. *Note:* take care that there are no precision issues in how you solve the problem; double-check your factorization!

```
  N = 133140272889335192922108409260662174476303831652383671688547009484
      253235940586917140482669182256368285260992829447207980183170174867
      620358952230969986447559330583492429636627298640338596531894556546
      013113154346823212271748927859647994534586133553218022983848108421
      465442089919090610542344768294481725103757222421917115971063026806
      587141287587037265150653669094323116686574536558866591647361053311
      046516013069669036866734126558017744393751161611219195769578488559
      882902397248309033911661475005854696820021069072502248533328754832
      698616238405221381252145137439919090800085955274389382721844956661
      1138745095472005761807
phi = 133140272889335192922108409260662174476303831652383671688547009484
      253235940586917140482669182256368285260992829447207980183170174867
      620358952230969986447559330583492429636627298640338596531894556546
      013113154346823212271748927859647994534586133553218022983848108421
      465442089919090610542344768294481725103757214932292046538867218497
      635256772227370109066785312096589779622355495419006049974567895189
      687318110498058692315630856693672069320529062399681563590382015177
      322909744749330702607931428154183726552004527201956226396835500346
      779062494259638983191117891502783513452775160701785906451173152044O
      2981816860178885028680
```

13.11. True or false: if $x^2 \equiv_N 1$ then $x \in \mathbb{Z}_N^*$. Prove or give a counterexample.

13.12. Discuss the computational difficulty of the following problem:

Given an integer N, find an element of $\mathbb{Z}_N \setminus \mathbb{Z}_N^*$.

If you can, relate its difficulty to that of other problems we've discussed (factoring N or inverting RSA).

13.13. (a) Show that it is possible to efficiently compute all four square roots of unity modulo pq, given p and q. *Hint:* CRT!

(b) Implement a `pari` function that takes distinct primes p and q as input and returns the four square roots of unity modulo pq. Use it to compute the four square roots of unity modulo

$$105295498644227198587577819266 3 \times 611174539744122090068393470777.$$

★ 13.14. Show that, conditioned on $w \in \mathbb{Z}_N^*$, the SqrtUnity subroutine outputs a square root of unity chosen uniformly at random from the 4 possible square roots of unity. *Hint:* use the Chinese Remainder Theorem.

13.15. Suppose N is an RSA modulus, and $x^2 \equiv_N y^2$, but $x \not\equiv_N \pm y$. Show that N can be efficiently factored if such a pair x and y are known.

13.16. Why are ±1 the only square roots of unity modulo p, when p is an odd prime?

13.17. When N is an RSA modulus, why is squaring modulo N a 4-to-1 function, but raising to the e^{th} power modulo N is 1-to-1?

13.18. Implement a `pari` function that efficiently factors an RSA modulus N, given only N, e, and d. Use your function to factor the following 2048-bit RSA modulus. *Note:* `pari` function `valuation(n,p)` returns the largest number d such that $p^d \mid n$.

```
N = 157713892705550064909750632475691896977526767652833932128735618711
    213662561319634033137058267272367265499003291937716454788882499492
    311117065951077245304317542978715216577264400048278064574204140564
    709253009840166821302184014310192765595015483588878761062406993721
    851190041888790873152584082212461847511180066690936944585390792304
    663763886417861546718283897613617078370412411019301687497005038294
    389148932398661048471814117247898148030982257697888167001010511378
    647288478239379740416388270380035364271593609513220655573614212415
    962670795230819103845127007912428958291134064942068225836213242131
    15022256956985205924967
e = 327598866483920224268285375349315001772252982661926675504591773242
    501030864502336359508677092544631083799700755236766113095163469666
    905258066495934057774395712118774014408282455244138409433389314036
    198045263991986560198273156037233588691392913730537367184867549274
    682884119866630822924707702796323546327425328705958528315517584489
    590815901470874024949798420173098581333151755836650797037848765578
    433873141626191257009250151327378074817106208930064676608134109788
    601067077103742326030259629322458620311949453584045538305945217564
    027461013225009980998673160144967719374426764116721861138496780008
    6366258360757218165973
d = 138476999734263775498100443567132759182144573474474014195021091272
    755207803162019484487127866675422608401990888942659393419384528257
    462434633738686176601555755842189986431725335031620097854962295968
    391161090826380458969236418585963384717406704714837349503808786086
    701573765714825783042297344050528898259745757741233099297952332012
    749897281090378398001337057869189488734951853748327631883502135139
    523664990296334020327713900408683264232664645438899178442633342438
    198329983121207315436447041915897544445402505558420138506655106015
    215450140256129977382476062366519087386576874886938585789874186326
    69265500594424847344765
```

13.19. In this problem we'll see that it's bad to choose RSA prime factors p and q too close together.

(a) Let $s = (p-q)/2$ and $t = (p+q)/2$. Then t is an integer greater than $\sqrt{N}$ such that $t^2 - N$ is a perfect square. When p and q are close, t is not much larger than $\sqrt{N}$, so by testing successive integers, it is possible to find t and s, and hence p and q. Describe the details of this attack and how it works.

(b) Implement a `pari` function that factors RSA moduli using this approach. Use it to factor the following 2048-bit number (whose two prime factors are guaranteed to be close enough for the factoring approach to work in a reasonable amount of time, but far enough apart that you can't do the trial-and-error part by hand). What qualifies as "close prime factors" in this problem? How close was t to $\sqrt{N}$?

Hint: `pari` has an `issquare` function. Also, be sure to do exact square roots over the integers, not the reals.

```
N = 514202868664266501986736340226343880193216864011643244558701956114
    553317880043289827487456460284103951463512024249329243228109624011
    915392411888724026403127686707255825056081890692595715828380690811
    131686383180282330775572385822102181209569411961125753242467971879
    131305986986525600110340790595987975345573842266766492356686762134
    653833064511337433089249621257629107825681429573934949101301135200
    918606211394413498735486599678541369375887840013842439026159037108
    043724221865116794034194812236381299786395457277559879575752254116
    612726596118528071785474551058540599198869986780286733916614335663
    3723003246569630373323
```

14 Diffie-Hellman Key Agreement

14.1 Cyclic Groups

Definition 14.1 *Let $g \in \mathbb{Z}_n^*$. Define $\langle g \rangle_n = \{g^i \,\%\, n \mid i \in \mathbb{Z}\}$, the set of all powers of g reduced mod n. Then g is called a* ***generator*** *of $\langle g \rangle_n$, and $\langle g \rangle_n$ is called the* ***cyclic group generated by*** *g* ***mod*** *n.*

If $\langle g \rangle_n = \mathbb{Z}_n^*$, then we say that g is a ***primitive root mod n.***

The definition allows the generator g to be raised to a *negative* integer. Since $g \in \mathbb{Z}_n^*$, it is guaranteed that g has a multiplicative inverse mod n, which we can call g^{-1}. Then g^{-i} can be defined as $g^{-i} \overset{\text{def}}{=} (g^{-1})^i$. All of the usual laws of exponents hold with respect to this definition of negative exponents.

Example *Taking $n = 13$, we have:*

$$\begin{aligned}
\langle 1 \rangle_{13} &= \{1\} \\
\langle 2 \rangle_{13} &= \{1,2,4,8,3,6,12,11,9,5,10,7\} = \mathbb{Z}_{13}^* \\
\langle 3 \rangle_{13} &= \{1,3,9\}
\end{aligned}$$

Thus 2 is a primitive root modulo 13. Each of the groups $\{1\}$, $\mathbb{Z}_{13}^*$, $\{1,3,9\}$ is a cyclic group under multiplication mod 13.

A cyclic group may have more than one generator, for example:

$$\langle 3 \rangle_{13} = \langle 9 \rangle_{13} = \{1,3,9\}$$

Similarly, there are four primitive roots modulo 13 (equivalently, $\mathbb{Z}_{13}^*$ has four different generators); they are 2, 6, 7, and 11.

Not every integer has a primitive root. For example, there is no primitive root modulo 15. However, when p is a prime, there is always a primitive root modulo p (and so $\mathbb{Z}_p^*$ is a cyclic group).

Let us write $\mathbb{G} = \langle g \rangle = \{g^i \mid i \in \mathbb{Z}\}$ to denote an unspecified cyclic group generated by g. The defining property of $\mathbb{G}$ is that each of its elements can be written as a power of g. From this we can conclude that:

- Any cyclic group is closed under multiplication. That is, take any $X, Y \in \mathbb{G}$; then it must be possible to write $X = g^x$ and $Y = g^y$ for some integers x, y. Using the multiplication operation of $\mathbb{G}$, the product is $XY = g^{x+y}$, which is also in $\mathbb{G}$.
- Any cyclic group is closed under inverses. Take any $X \in \mathbb{G}$; then it must be possible to write $X = g^x$ for some integer x. We can then see that $g^{-x} \in \mathbb{G}$ by definition, and $g^{-x}X = g^{-x+x} = g^0$ is the identity element. So X has a multiplicative inverse (g^{-x}) in $\mathbb{G}$.

These facts demonstrate that $\mathbb{G}$ is indeed a *group* in the terminology of abstract algebra.

Discrete Logarithms

It is typically easy to compute the value of g^x in a cyclic group, given g and x. For example, when using a cyclic group of the form $\mathbb{Z}_n^*$, we can easily compute the modular exponentiation $g^x \% n$ using repeated squaring.

The inverse operation in a cyclic group is called the discrete logarithm problem:

Definition 14.2 (Discrete Log) *The **discrete logarithm problem** is: given $X \in \langle g \rangle$, determine a number x such that $g^x = X$. Here the exponentiation is with respect to the multiplication operation in $\mathbb{G} = \langle g \rangle$.*

The discrete logarithm problem is conjectured to hard (that is, no polynomial-time algorithm exists for the problem) in certain kinds of cyclic groups.

14.2 Diffie-Hellman Key Agreement

Key agreement refers to the problem of establishing a private channel using public communication. Suppose Alice & Bob have never spoken before and have no shared secrets. By exchanging *public* messages (*i.e.*, that can be seen by any external observer), they would like to establish a secret that is known only to the two of them.

The **Diffie-Hellman** protocol is such a key-agreement protocol, and it was the first published instance of public-key cryptography:

Construction 14.3 (Diffie-Hellman) *Both parties agree (publicly) on a cyclic group $\mathbb{G}$ with generator g. Let $n = |\mathbb{G}|$. All exponentiations are with respect to the group operation in $\mathbb{G}$.*

1. *Alice chooses $a \leftarrow \mathbb{Z}_n$. She sends $A = g^a$ to Bob.*
2. *Bob chooses $b \leftarrow \mathbb{Z}_n$. He sends $B = g^b$ to Alice.*
3. *Bob locally outputs $K := A^b$. Alice locally outputs $K := B^a$.*

$$\begin{tikzcd}
\underline{\text{Alice}} & & \underline{\text{Bob}} \\
a \leftarrow \mathbb{Z}_n \arrow[rr, "A = g^a"] & & {} \\
{} & & b \leftarrow \mathbb{Z}_n \arrow[ll, "B = g^b"'] \\
\text{return } B^a & & \text{return } A^b
\end{tikzcd}$$

By substituting and applying standard rules of exponents, we see that both parties output a common value, namely $K = g^{ab} \in \mathbb{G}$.

Defining Security for Key Agreement

Executing a key agreement protocol leaves two artifacts behind. First, we have the collection of messages that are exchanged between the two parties. We call this collection a **transcript.** We envision two parties executing a key agreement protocol in the presence of an *eavesdropper,* and hence we imagine that the transcript is public. Second, we have the **key** that is output by the parties, which is private.

To define security of key agreement, we would like to require that the transcript leaks no (useful) information to the eavesdropper about the key. There are a few ways to approach the definition:

- We could require that it is hard to compute the key given the transcript. However, this turns out to be a rather weak definition. For example, it does not rule out the possibility that an eavesdropper could guess the *first half* of the bits of the key.
- We could require that the key is *pseudorandom* given the transcript. This is a better definition, and the one we use. To formalize this idea, we define two libraries. In both libraries the adversary / calling program can obtain the transcript of an execution of the key agreement protocol. In one library the adversary obtains the key that resulted from the protocol execution, while in the other library the adversary obtains a totally unrelated key (chosen uniformly from the set $\Sigma.\mathcal{K}$ of possible keys).

Definition 14.4 (KA security) *Let Σ be a key-agreement protocol. We write $\Sigma.\mathcal{K}$ for the keyspace of the protocol (*i.e., *the set of possible keys it produces). We write $(t,K) \leftarrow \textsc{execprot}(\Sigma)$ to denote the process of executing the protocol between two honest parties, where t denotes the resulting transcript, and K is resulting key. Note that this process is randomized, and that K is presumably correlated to t.*

*We say that Σ is **secure** if $\mathcal{L}^{\Sigma}_{\text{ka-real}} \approx \mathcal{L}^{\Sigma}_{\text{ka-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{ka-real}}$
QUERY():
  $(t,K) \leftarrow \textsc{execprot}(\Sigma)$
  return $(t,K)$
```

```
$\mathcal{L}^{\Sigma}_{\text{ka-rand}}$
QUERY():
  $(t,K) \leftarrow \textsc{execprot}(\Sigma)$
  $K' \leftarrow \Sigma.\mathcal{K}$
  return $(t,K')$
```

14.3 Decisional Diffie-Hellman Problem

The Diffie Hellman protocol is parameterized by the choice of cyclic group $\mathbb{G}$ (and generator g). Transcripts in the protocol consist of (g^a, g^b), where a and b are chosen uniformly. The key corresponding to such a transcript is g^{ab}. The set of possible keys is the cyclic group $\mathbb{G}$.

Let us substitute the details of the Diffie-Hellman protocol into the KA security libraries. After simplifying, we see that the security of the Diffie Hellman protocol is equivalent to the following statement:

```
$\mathcal{L}^{\mathbb{G}}_{\text{dh-real}}$
QUERY():
  $a, b \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^{ab})$
```

$\approx$

```
$\mathcal{L}^{\mathbb{G}}_{\text{dh-rand}}$
QUERY():
  $a, b, c \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^c)$
```

We have renamed the libraries to $\mathcal{L}_{\text{dh-real}}$ and $\mathcal{L}_{\text{dh-rand}}$. In $\mathcal{L}_{\text{dh-real}}$ the response to QUERY corresponds to a DHKA transcript (g^a, g^b) along with the corresponding "correct" key

g^{ab}. The response in $\mathcal{L}_{\text{dh-rand}}$ corresponds to a DHKA transcript along with a completely independent random key g^c.

Definition 14.5 (DDH) *The* ***decisional Diffie-Hellman (DDH) assumption*** *in a cyclic group* $\mathbb{G}$ *is that* $\mathcal{L}^{\mathbb{G}}_{\text{dh-real}} \approx \mathcal{L}^{\mathbb{G}}_{\text{dh-rand}}$ *(libraries defined above).*

Since we have defined the DDH assumption by simply renaming the security definition for DHKA, we immediately have:

Claim 14.6 *The DHKA protocol is a secure KA protocol* ***if and only if*** *the DDH assumption is true for the choice of* $\mathbb{G}$ *used in the protocol.*

For Which Groups does the DDH Assumption Hold?

So far our only example of a cyclic group is $\mathbb{Z}_p^*$, where p is a prime. Although *many* textbooks describe DHKA in terms of this cyclic group, it is not a good choice because the DDH assumption is *demonstrably false* in $\mathbb{Z}_p^*$. To see why, we introduce a new concept:

Claim 14.7 (Euler criterion) *If* p *is a prime and* $X = g^x \in \mathbb{Z}_p^*$, *then* $X^{\frac{p-1}{2}} \equiv_p (-1)^x$.

Note that $(-1)^x$ is 1 if x is even and -1 if x is odd. So, while in general it is hard to determine x given g^x, Euler's criterion says that it is possible to determine the *parity of* x (*i.e.*, whether x is even or odd) given g^x.

To see how these observations lead to an attack against the Diffie-Hellman protocol, consider the following attack:

```
$\mathcal{A}$:
$(A, B, C) \leftarrow$ QUERY()
return $1 \overset{?}{\equiv}_p C^{\frac{p-1}{2}}$
```

Roughly speaking, the adversary returns true whenever C can be written as g raised to an *even* exponent. When linked to $\mathcal{L}_{\text{dh-real}}$, $C = g^{ab}$ where a and b are chosen uniformly. Hence ab will be even with probability 3/4. When linked to $\mathcal{L}_{\text{dh-rand}}$, $C = g^c$ for an independent random c. So c is even only with probability 1/2. Hence the adversary distinguishes the libraries with advantage 1/4.

Concretely, with this choice of group, the key g^{ab} will never be uniformly distributed. See the exercises for a slightly better attack which correlates the key to the transcript.

Quadratic Residues. Several better choices of cyclic groups have been proposed in the literature. Arguably the simplest one is based on the following definition:

Definition 14.8 *A number* $X \in \mathbb{Z}_n^*$ *is a* ***quadratic residue modulo*** n *if there exists some integer* Y *such that* $Y^2 \equiv_n X$. *That is, if* X *can be obtained by squaring a number mod* n. *Let* $\mathbb{QR}_n^* \subseteq \mathbb{Z}_n^*$ *denote the set of quadratic residues mod* n.

For our purposes it is enough to know that, when p is prime, $\mathbb{QR}_p^*$ is a cyclic group with $(p-1)/2$ elements (see the exercises). When both p and $(p-1)/2$ are prime, we call p a **safe prime** (and call $(p-1)/2$ a *Sophie Germain prime*). To the best of our knowledge the DDH assumption is true in $\mathbb{QR}_p^*$ when p is a safe prime.

Exercises

14.1. Let p be an odd prime, as usual. Recall that $\mathbb{QR}^*_p$ is the set of quadratic residues mod p — that is, $\mathbb{QR}^*_p = \{x \in \mathbb{Z}^*_p \mid \exists y : x \equiv_p y^2\}$. Show that if g is a primitive root of $\mathbb{Z}^*_p$ then $\langle g^2 \rangle = \mathbb{QR}^*_p$.

Note: This means that $g^a \in \mathbb{QR}^*_p$ if and only if a is even — and in particular, the choice of generator g doesn't matter.

14.2. Suppose $N = pq$ where p and q are distinct primes. Show that $|\mathbb{QR}^*_N| = |\mathbb{QR}^*_p| \cdot |\mathbb{QR}^*_q|$.

Hint: Chinese remainder theorem.

14.3. Suppose you are given $X \in \langle g \rangle$. You are allowed to choose any $X' \neq X$ and learn the discrete log of X' (with respect to base g). Show that you can use this ability to learn the discrete log of X.

14.4. Let $\langle g \rangle$ be a cyclic group with n elements and generator g. Show that for all integers a, it is true that $g^a = g^{a \% n}$.

Note: As a result, $\langle g \rangle$ is isomorphic to the additive group $\mathbb{Z}_n$.

14.5. Let g be a primitive root of $\mathbb{Z}^*_n$. Recall that $\mathbb{Z}^*_n$ has $\phi(n)$ elements. Show that g^a is a primitive root of $\mathbb{Z}^*_n$ if and only if $\gcd(a, \phi(n)) = 1$.

Note: It follows that, for every n, there are either 0 or $\phi(\phi(n))$ primitive roots mod n.

14.6. Let $\langle g \rangle$ be a cyclic group with n elements. Show that for all $x, y \in \langle g \rangle$, it is true that $x^n = y^n$.

Hint: every $x \in \langle g \rangle$ can be written as $x = g^a$ for some appropriate a. What is $(g^a)^n$?

14.7. (a) Prove the following variant of Lemma 4.9: Suppose you fix a value $x \in \mathbb{Z}_N$. Then when sampling $q = \sqrt{2N}$ values $r_1, \ldots, r_q$ uniformly from $\mathbb{Z}_N$, with probability at least 0.6 there exist $i \neq j$ with $r_i \equiv_N r_j + x$.

Hint: This is extremely similar to Exercise **??**.

(b) Let g be a primitive root of $\mathbb{Z}^*_p$ (for some prime p). Consider the problem of computing the discrete log of $X \in \mathbb{Z}^*_p$ with respect to g — that is, finding x such that $X \equiv_p g^x$. Argue that if one can find integers r and s such that $g^r \equiv_p X \cdot g^s$ then one can compute the discrete log of X.

(c) Combine the above two observations to describe a $O(\sqrt{p})$-time algorithm for the discrete logarithm problem in $\mathbb{Z}^*_p$.

14.8. In an execution of DHKA, the eavesdropper observes the following values:

$$p = 461733370363 \qquad A = 114088419126$$
$$g = 2 \qquad B = 276312808197$$

What will be Alice & Bob's shared key?

14.9. Explain what is wrong in the following argument:

In Diffie-Hellman key agreement, Alice sends $A = g^a$ and Bob sends $B = g^b$. Their shared key is g^{ab}. To break the scheme, the eavesdropper can simply compute $A \cdot B = (g^a)(g^b) = g^{ab}$.

14.10. Let $\mathbb{G}$ be a cyclic group with n elements and generator g. Consider the following algorithm:

```
RAND($A,B,C$):
    $r,s,t \leftarrow \mathbb{Z}_n$
    $A' := A^t g^r$
    $B' := Bg^s$
    $C' := C^t B^r A^{st} g^{rs}$
    return $(A',B',C')$
```

Let $DH = \{(g^a, g^b, g^{ab}) \in \mathbb{G}^3 \mid a,b, \in \mathbb{Z}_n\}$.

(a) Suppose $(A,B,C) \in DH$. Show that the output distribution of RAND(A,B,C) is the uniform distribution over DH

(b) Suppose $(A,B,C) \notin DH$. Show that the output distribution of RAND(A,B,C) is the uniform distribution over $\mathbb{G}^3$.

★ (c) Consider the problem of determining whether a given triple (A,B,C) is in the set DH. Suppose you have an algorithm $\mathcal{A}$ that solves this problem on average slightly better than chance. That is:

$$\Pr[\mathcal{A}(A,B,C) = 1] > 0.51 \text{ when } (A,B,C) \text{ chosen uniformly in } DH$$
$$\Pr[\mathcal{A}(A,B,C) = 0] > 0.51 \text{ when } (A,B,C) \text{ chosen uniformly in } \mathbb{G}^3$$

The algorithm $\mathcal{A}$ does not seem very useful if you have a *particular* triple (A,B,C) and you really want to know whether it is in DH. You might have one of the triples for which $\mathcal{A}$ gives the wrong answer, and there's no real way to know.

Show how to construct a randomized algorithm $\mathcal{A}'$ such that: for every $(A,B,C) \in \mathbb{G}^3$:

$$\Pr\left[\mathcal{A}'(A,B,C) = [(A,B,C) \overset{?}{\in} DH]\right] > 0.99$$

Here the input A,B,C is fixed and the probability is over the internal randomness in $\mathcal{A}'$. So on *every* possible input, $\mathcal{A}'$ gives a very reliable answer.

to-do *better attack against $\mathbb{Z}_p^*$ instantiation of DHKA*

15 Public-Key Encryption

So far, the encryption schemes that we've seen are **symmetric-key** schemes. The same key is used to encrypt and decrypt. In this chapter we introduce **public-key** (sometimes called *asymmetric*) encryption schemes, which use different keys for encryption and decryption. The idea is that the encryption key can be made *public*, so that anyone can send an encryption to the owner of that key, even if the two users have never spoken before and have no shared secrets. The decryption key is private, so that only the designated owner can decrypt.

We modify the syntax of an encryption scheme in the following way. A public-key encryption scheme consists of the following three algorithms:

KeyGen: Outputs a *pair* (pk, sk) where pk is a public key and sk is a private/secret key.

Enc: Takes the public key pk and a plaintext m as input, and outputs a ciphertext c.

Dec: Takes the secret key sk and a ciphertext c as input, and outputs a plaintext m.

We modify the correctness condition similarly. A public-key encryption scheme satisfies *correctness* if, for all $m \in \mathcal{M}$ and all $(pk, sk) \leftarrow \text{KeyGen}$, we have $\text{Dec}(sk, \text{Enc}(pk, m)) = m$ (with probability 1 over the randomness of Enc).

15.1 Security Definitions

We now modify the definition of CPA security to fit the setting of public-key encryption. As before, the adversary calls a challenge subroutine with two plaintexts — the difference between the two libraries is which plaintext is actually encrypted. Of course, the encryption operation now takes the public key.

Then the biggest change is that we would like to make the public key *public.* In other words, the calling program should have a way to learn the public key (otherwise the library cannot model a situation where the public key is known to the adversary). To do this, we simply add another subroutine that returns the public key.

Definition 15.1 *Let Σ be a public-key encryption scheme. Then Σ is* ***secure against chosen-plaintext at-***

tacks (CPA secure) *if* $\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$, *where:*

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m_L)$
```

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m_R)$
```

to-do *Re-iterate how deterministic encryption still can't be CPA-secure in the public-key setting.*

Pseudorandom Ciphertexts

We can modify/adapt the definition of pseudorandom ciphertexts to public-key encryption in a similar way:

Definition 15.2 *Let Σ be a public-key encryption scheme. Then Σ has* ***pseudorandom ciphertexts in the presence of chosen-plaintext attacks (CPA\$ security)*** *if* $\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa\$-rand}}$, *where:*

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-real}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  return $\Sigma.\text{Enc}(pk, m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-rand}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
  return $pk$

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

As in the symmetric-key setting, CPA\$ security (for public-key encryption) implies CPA security:

Claim 15.3 *Let Σ be a public-key encryption scheme. If Σ has CPA\$ security, then Σ has CPA security.*

The proof is extremely similar to the proof of the analogous statement for symmetric-key encryption (Theorem 8.3), and is left as an exercise.

15.2 One-Time Security Implies Many-Time Security

So far, everything about public-key encryption has been directly analogous to what we've seen about symmetric-key encryption. We now discuss a peculiar property that is different between the two settings.

In symmetric-key encryption, we saw examples of encryption schemes that are secure when the adversary sees only one ciphertext, but insecure when the adversary sees more ciphertexts. One-time pad is the standard example of such an encryption scheme.

Surprisingly, if a *public-key* encryption scheme is secure when the adversary sees just one ciphertext, then it is also secure for many ciphertexts! In short, there is no public-key one-time pad that is weaker than full-fledged public-key encryption — there is public-key encryption or nothing.

To show this property formally, we first adapt the definition of one-time secrecy (Definition 2.6) to the public-key setting. There is one small but important technical subtlety: in Definition 2.6 the encryption key is chosen at the last possible moment in the body of CHALLENGE. This ensures that the key is local to this scope, and therefore each value of the key is only used to encrypt one plaintext.

In the public-key setting, however, it turns out to be important to allow the adversary to see the public key before deciding which plaintexts to encrypt. (This concern is not present in the symmetric-key setting precisely because there is nothing public upon which the adversary's choice of plaintexts can depend.) For that reason, in the public-key setting we must sample the keys at initialization time so that the adversary can obtain the public key via GETPK. To ensure that the key is used to encrypt only one plaintext, we add a counter and a guard condition to CHALLENGE, so that it only responds once with a ciphertext.

Definition 15.4 *Let Σ be a public-key encryption scheme. Then Σ has* ***one-time secrecy*** *if $\mathcal{L}^{\Sigma}_{\text{pk-ots-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{pk-ots-L}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $count := count + 1$
  if $count > 1$: return null
  return $\Sigma.\text{Enc}(pk, m_L)$
```

```
$\mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $count := count + 1$
  if $count > 1$: return null
  return $\Sigma.\text{Enc}(pk, m_R)$
```

Claim 15.5 *Let Σ be a public-key encryption scheme. If Σ has one-time secrecy, then Σ is CPA-secure.*

Proof Suppose $\mathcal{L}^{\Sigma}_{\text{pk-ots-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-R}}$. Our goal is to show that $\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$. The proof centers around the following hybrid library $\mathcal{L}_{\text{hyb-}h}$, which is designed to be linked to either

$\mathcal{L}_{\text{pk-ots-L}}$ or $\mathcal{L}_{\text{pk-ots-R}}$:

```
$\mathcal{L}_{\text{hyb-}h}$

$count = 0$
$pk :=$ GETPK()

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  $count := count + 1$
  if $count < h$:
    return $\Sigma.\text{Enc}(pk, m_R)$
  elsif $count = h$:
    return CHALLENGE′($m_L, m_R$)
  else:
    return $\Sigma.\text{Enc}(pk, m_L)$
```

Here the value h is an unspecified value that will be a hard-coded constant, and CHALLENGE′ (called by the "elsif" branch) and GETPK refer to the subroutine in $\mathcal{L}_{\text{pk-ots-}\star}$. Note that $\mathcal{L}_{\text{hyb-}h}$ is designed so that it only makes one call to CHALLENGE′ — in particular, only when its own CHALLENGE subroutine is called for the h^{th} time.

We now make a few observations:

$\mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-L}} \equiv \mathcal{L}_{\text{pk-cpa-L}}$: In both libraries, every call to CHALLENGE encrypts the left plaintext. In particular, the first call to CHALLENGE in $\mathcal{L}_{\text{hyb-1}}$ triggers the "elsif" branch, so the challenge is routed to $\mathcal{L}_{\text{pk-ots-L}}$, which encrypts the left plaintext. In all other calls to CHALLENGE, the "else" branch is triggered and the left plaintext is encrypted explicitly.

$\mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-L}} \equiv \mathcal{L}_{\text{hyb-}(h+1)} \diamond \mathcal{L}_{\text{pks-ots-R}}$, for all h. In both of these libraries, the first h calls to CHALLENGE encrypt the left plaintext, and all subsequent calls encrypt the right plaintext.

$\mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-L}} \approx \mathcal{L}_{\text{hyb-}h} \diamond \mathcal{L}_{\text{pks-ots-R}}$, for all h. This simply follows from the fact that $\mathcal{L}_{\text{pks-ots-L}} \approx \mathcal{L}_{\text{pks-ots-R}}$.

$\mathcal{L}_{\text{hyb-}q} \diamond \mathcal{L}_{\text{pks-ots-R}} \equiv \mathcal{L}_{\text{pk-cpa-R}}$, where q is the number of times the calling program calls CHALLENGE. In particular, every call to CHALLENGE encrypts the right plaintext.

Putting everything together, we have that:

$$\begin{aligned}\mathcal{L}_{\text{pk-cpa-L}} &\equiv \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-L}} \approx \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-R}} \\ &\equiv \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{\text{pk-ots-L}} \approx \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{\text{pk-ots-R}} \\ &\;\vdots\end{aligned}$$

$$\begin{aligned} &\equiv \mathcal{L}_{\textsf{hyb-}q} \diamond \mathcal{L}_{\textsf{pk-ots-L}} \approx \mathcal{L}_{\textsf{hyb-}q} \diamond \mathcal{L}_{\textsf{pk-ots-R}} \\ &\equiv \mathcal{L}_{\textsf{pk-cpa-R}}, \end{aligned}$$

and so $\mathcal{L}_{\textsf{pk-cpa-L}} \approx \mathcal{L}_{\textsf{pk-cpa-R}}$. ■

The reason this proof goes through for public-key encryption but not symmetric-key encryption is that *anyone can encrypt* in a public-key scheme. In a symmetric-key scheme, it is not possible to generate encryptions without the key. But in a public-key scheme, the encryption key is public.

In more detail, the $\mathcal{L}_{\textsf{hyb-}h}$ library can indeed obtain pk from $\mathcal{L}_{\textsf{pk-ots-}\star}$. It therefore has enough information to perform the encryptions for all calls to CHALLENGE. Indeed, you can think of $\mathcal{L}_{\textsf{hyb-0}}$ as doing everything that $\mathcal{L}_{\textsf{pk-cpa-L}}$ does, even though it doesn't know the secret key. We let $\mathcal{L}_{\textsf{hyb-}h}$ designate the h^{th} call to CHALLENGE as a special one to be handled by $\mathcal{L}_{\textsf{pk-ots-}\star}$. This allows us to change the h^{th} encryption from using m_L to m_R.

15.3 ElGamal Encryption

ElGamal encryption is a public-key encryption scheme that is based on DHKA.

Construction 15.6 (ElGamal) *The public parameters are a choice of cyclic group $\mathbb{G}$ with n elements and generator g.*

```
$\mathcal{M} = \mathbb{G}$
$\mathcal{C} = \mathbb{G}^2$

KeyGen:
    $sk := a \leftarrow \mathbb{Z}_n$
    $pk := A := g^a$
    return $(pk, sk)$

Enc$(A, M \in \mathbb{G})$:
    $b \leftarrow \mathbb{Z}_n$
    $B := g^b$
    return $(B, M \cdot A^b)$

Dec$(a, (B, X))$:
    return $X(B^a)^{-1}$
```

The scheme satisfies correctness, since for all M:

$$\begin{aligned} \textsf{Dec}(sk, \textsf{Enc}(pk, M)) &= \textsf{Dec}(sk, (g^b, M \cdot A^b)) \\ &= (M \cdot A^b)((g^b)^a)^{-1} \\ &= M \cdot (g^{ab})(g^{ab})^{-1} = M. \end{aligned}$$

Security

Imagine an adversary who is interested in attacking an ElGamal scheme. This adversary sees the public key $A = g^a$ and a ciphertext $(g^b, M g^{ab})$ go by. Intuitively, the Decisional Diffie-Hellman assumption says that the value g^{ab} looks random, even to someone who has seen g^a and g^b. Thus, the message M is masked with a pseudorandom group element — as we've seen before, this is a lot like masking the message with a random pad as in one-time pad. The only change here is that instead of the XOR operation, we are using the group operation in $\mathbb{G}$.

More formally, we can prove the security of ElGamal under the DDH assumption:

Claim 15.7 *If the DDH assumption in group $\mathbb{G}$ is true, then ElGamal in group $\mathbb{G}$ is CPA\$-secure.*

Proof It suffices to show that ElGamal has pseudorandom ciphertexts when the calling program sees only a single ciphertext. In other words, we will show that $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$, where these libraries are the $\mathcal{L}_{\text{pk-cpa\$-}\star}$ libraries from Definition 15.2 but with the single-ciphertext restriction used in Definition 15.4. It is left as an exercise to show that $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$ implies CPA\$ security (which in turn implies CPA security); the proof is very similar to that of Claim 15.5.

The sequence of hybrid libraries is given below:

```
$\mathcal{L}_{\text{pk-ots\$-real}}$
$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b \leftarrow \mathbb{Z}_n$
  $B := g^b$
  $X := M \cdot A^b$
  return $(B,X)$
```

The starting point is the $\mathcal{L}_{\text{pk-ots\$-real}}$ library, shown here with the details of ElGamal filled in.

```
$a \leftarrow \mathbb{Z}_n$; $b \leftarrow \mathbb{Z}_n$
$A := g^a$; $B := g^b$; $C := A^b$
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return $(B,X)$
```

The main body of QUERY computes some intermediate values B and A^b. But since those lines are only reachable one time, it does not change anything to precompute them at initialization time.

```
$(A,B,C) \leftarrow$ DHQUERY()
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return $(B,X)$
```

◇

```
$\mathcal{L}_{\text{dh-real}}$

DHQUERY():
  $a,b \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^{ab})$
```

We can factor out the generation of A,B,C in terms of the $\mathcal{L}_{\text{dh-real}}$ library from the Decisional Diffie-Hellman security definition (Definition 14.5).

```
$(A,B,C) \leftarrow$ DHQUERY()
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return $(B,X)$
```

◇

```
$\mathcal{L}_{\text{dh-rand}}$

DHQUERY():
  $a, b, c \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^c)$
```

Applying the security of DDH, we can replace $\mathcal{L}_{\text{dh-real}}$ with $\mathcal{L}_{\text{dh-rand}}$.

```
$a,b,c \leftarrow \mathbb{Z}_n$
$A := g^a$; $B := g^b$; $C := g^c$
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $X := M \cdot C$
  return $(B,X)$
```

The call to DHQUERY has been inlined.

```
$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b, c \leftarrow \mathbb{Z}_n$
  $B := g^b$;  $C := g^c$
  $X := M \cdot C$
  return $(B, X)$
```

As before, since the main body of QUERY is only reachable once, we can move the choice of B and C into that subroutine instead of at initialization time.

```
$\mathcal{L}_{\text{pk-ots\$-rand}}$
$a \leftarrow \mathbb{Z}_n$
$A := g^a$
$count = 0$

GETPK():
  return $A$

QUERY($M \in \mathbb{G}$):
  $count : count + 1$
  if $count > 1$: return null
  $b, x \leftarrow \mathbb{Z}_n$
  $B := g^b$;  $X := g^x$
  return $(B, X)$
```

When b is sampled uniformly from $\mathbb{Z}_n$, the expression $B = g^b$ is a uniformly distributed element of $\mathbb{G}$. Also recall that when C is a uniformly distributed element of $\mathbb{G}$, then $M \cdot C$ is uniformly distributed — this is analogous to the one-time pad property (see Exercise 2.3). Applying this change gives the library to the left.

In the final hybrid, the response to QUERY is a pair of uniformly distributed group elements (B, X). Hence that library is exactly $\mathcal{L}_{\text{pk-ots\$-rand}}$, as desired. ■

15.4 Hybrid Encryption

As a rule, public-key encryption schemes are much more computationally expensive than symmetric-key schemes. Taking ElGamal as a representative example, computing g^b in a cryptographically secure cyclic group is considerably more expensive than one evaluation of AES. As the plaintext data increases in length, the difference in cost between public-key and symmetric-key techniques only gets worse.

A clever way to minimize the cost of public-key cryptography is to use a method called **hybrid encryption.** The idea is to use the expensive public-key scheme to encrypt a *temporary key* for a symmetric-key scheme. Then use the temporary key to (cheaply) encrypt the large plaintext data.

To decrypt, one can use the decryption key of the public-key scheme to obtain the temporary key. Then the temporary key can be used to decrypt the main payload.

Construction 15.8 (Hybrid Enc) *Let Σ_{pub} be a public-key encryption scheme, and let Σ_{sym} be a symmetric-key encryption scheme, where $\Sigma_{sym}.\mathcal{K} \subseteq \Sigma_{pub}.\mathcal{M}$ — that is, the public-key scheme is capable of encrypting keys of the symmetric-key scheme.*

Then we define Σ_{hyb} to be the following construction:

```
$\mathcal{M} = \Sigma_{sym}.\mathcal{M}$
$\mathcal{C} = \Sigma_{pub}.\mathcal{C} \times \Sigma_{sym}.\mathcal{C}$

KeyGen:
    $(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$
    return $(pk, sk)$

Enc$(pk, m)$:
    $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
    $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk)$
    $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m)$
    return $(c_{\text{pub}}, c_{\text{sym}})$

Dec$(sk, (c_{\text{pub}}, c_{\text{sym}}))$:
    $tk := \Sigma_{\text{pub}}.\text{Dec}(sk, c_{\text{pub}})$
    return $\Sigma_{\text{sym}}.\text{Dec}(tk, c_{\text{sym}})$
```

Importantly, the message space of the hybrid encryption scheme is the message space of the symmetric-key scheme (think of this as involving very long plaintexts), but encryption and decryption involves expensive public-key operations only on a small temporary key (think of this as a very short string).

The correctness of the scheme can be verified via:

$$\begin{aligned}
\text{Dec}(sk, \text{Enc}(pk, m)) &= \text{Dec}\Big(sk, \big(\Sigma_{\text{pub}}.\text{Enc}(pk, tk),\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\big)\Big) \\
&= \Sigma_{\text{sym}}.\text{Dec}\Big(\Sigma_{\text{pub}}.\text{Dec}\big(sk,\ \Sigma_{\text{pub}}.\text{Enc}(pk, tk)\big),\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\Big) \\
&= \Sigma_{\text{sym}}.\text{Dec}\Big(tk,\ \Sigma_{\text{sym}}.\text{Enc}(tk, m)\Big) \\
&= m.
\end{aligned}$$

To show that hybrid encryption is a valid way to encrypt data, we prove that it provides CPA security, when its two components have appropriate security properties:

Claim 15.9 *If Σ_{sym} is a one-time-secret symmetric-key encryption scheme and Σ_{pub} is a CPA-secure public-key encryption scheme, then the hybrid scheme Σ_{hyb} (Construction 15.8) is also a CPA-secure public-key encryption scheme.*

Note that Σ_{sym} does not even need to be CPA-secure. Intuitively, one-time secrecy suffices because each temporary key tk is used only once to encrypt just a single plaintext.

Proof As usual, our goal is to show that $\mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-L}} \approx \mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-R}}$, which we do in a standard sequence of hybrids:

```
$\mathcal{L}^{\Sigma_{\text{hyb}}}_{\text{pk-cpa-L}}$

$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R$):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk)$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$
```

The starting point is $\mathcal{L}_{\text{pk-cpa-L}}$, shown here with the details of Σ_{hyb} filled in.

Our only goal is to somehow replace m_L with m_R. Since m_L is only used as a plaintext for Σ_{sym}, it is tempting to simply apply the one-time-secrecy property of Σ_{sym} to argue that m_L can be replaced with m_R. Unfortunately, this cannot work because the *key* used for that ciphertext is tk, which is used elsewhere. In particular, it is used as an argument to $\Sigma_{\text{pub}}.\text{Enc}$.

However, using tk as the plaintext argument to $\Sigma_{\text{pub}}.\text{Enc}$ should *hide* tk to the calling program, if Σ_{pub} is CPA-secure. That is, the Σ_{pub}-encryption of tk should look like a Σ_{pub}-encryption of some unrelated dummy value. More formally, we can factor out the call to $\Sigma_{\text{pub}}.\text{Enc}$ in terms of the $\mathcal{L}_{\text{pk-cpa-L}}$ library, as follows:

```
CHALLENGE($m_L, m_R$):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $tk' \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow$ CHALLENGE'$(tk, tk')$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$
```

$\diamond$

```
$\mathcal{L}^{\Sigma_{\text{pub}}}_{\text{pk-cpa-L}}$

$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE'($tk_L, tk_R$):
  return $\Sigma_{\text{pub}}.\text{Enc}(pk, tk_L)$
```

Here we have changed the variable names of the arguments of CHALLENGE′ to avoid unnecessary confusion. Note also that CHALLENGE now chooses *two* temporary keys — one which is actually used to encrypt m_L and one which is not used anywhere. This is because syntactically we must have two arguments to pass into CHALLENGE′.

Now imagine replacing $\mathcal{L}_{\text{pk-cpa-L}}$ with $\mathcal{L}_{\text{pk-cpa-R}}$ and then inlining subroutine calls. The result is:

```
$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
  return $pk$

CHALLENGE($m_L, m_R$):
  $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $tk' \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
  $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk')$
  $c_{\text{sym}} \leftarrow \Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
  return $(c_{\text{pub}}, c_{\text{sym}})$
```

At this point, it *does* now work to factor out the call to $\Sigma_{\text{sym}}.\text{Enc}$ in terms of the $\mathcal{L}_{\text{ots-L}}$ library. This is because the key tk is not used anywhere else in the library. The result of factoring out in this way is:

```
$(pk, sk) \leftarrow \Sigma_{\text{pub}}.\text{KeyGen}$

GETPK():
    return $pk$

CHALLENGE($m_L, m_R$):
    $tk' \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
    $c_{\text{pub}} \leftarrow \Sigma_{\text{pub}}.\text{Enc}(pk, tk')$
    $c_{\text{sym}} \leftarrow$ CHALLENGE'($m_L, m_R$)
    return $(c_{\text{pub}}, c_{\text{sym}})$
```

$\diamond$

```
$\mathcal{L}^{\Sigma_{\text{sym}}}_{\text{ots-L}}$

CHALLENGE'($m_L, m_R$):
    $tk \leftarrow \Sigma_{\text{sym}}.\text{KeyGen}$
    return $\Sigma_{\text{sym}}.\text{Enc}(tk, m_L)$
```

At this point, we can replace $\mathcal{L}_{\text{ots-L}}$ with $\mathcal{L}_{\text{ots-R}}$. After this change the Σ_{sym}-ciphertext encrypts m_R instead of m_L. This is the "half-way point" of the proof, and the rest of the steps are a mirror image of what has come before. In summary: we inline $\mathcal{L}_{\text{ots-R}}$, then we apply CPA security to replace the Σ_{pub}-encryption of tk' with tk. The result is exactly $\mathcal{L}_{\text{pk-cpa-R}}$, as desired. ■

Exercises

15.1. Prove Claim 15.3.

15.2. Show that a 2-message key-agreement protocol exists if and only if CPA-secure public-key encryption exists.

In other words, show how to construct a CPA-secure encryption scheme from any 2-message KA protocol, and vice-versa. Prove the security of your constructions.

15.3. (a) Suppose you are given an ElGamal encryption of an unknown plaintext $M \in \mathbb{G}$. Show how to construct a different ciphertext that also decrypts to the same M.

(b) Suppose you are given two ElGamal encryptions, of unknown plaintexts $M_1, M_2 \in \mathbb{G}$. Show how to construct a ciphertext that decrypts to their product $M_1 \cdot M_2$.

15.4. Suppose you obtain two ElGamal ciphertexts (B_1, C_1), (B_2, C_2) that encrypt unknown plaintexts M_1 and M_2. Suppose you also know the public key A and cyclic group generator g.

(a) What information can you infer about M_1 and M_2 if you observe that $B_1 = B_2$?

(b) What information can you infer about M_1 and M_2 if you observe that $B_1 = g \cdot B_2$?

★ (c) What information can you infer about M_1 and M_2 if you observe that $B_1 = (B_2)^2$?

Index of Security Definitions

One-time secrecy for symmetric-key encryption (Definition 2.6):

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

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

t-out-of-n secret sharing (Definition 3.3):

```
$\mathcal{L}^{\Sigma}_{\text{tsss-L}}$
QUERY($m_L, m_R \in \Sigma.\mathcal{M}, U$):
    if $|U| \geqslant \Sigma.t$: return err
    $s \leftarrow \Sigma.\text{Share}(m_L)$
    return $\{s_i \mid i \in U\}$
```

```
$\mathcal{L}^{\Sigma}_{\text{tsss-R}}$
QUERY($m_L, m_R \in \Sigma.\mathcal{M}, U$):
    if $|U| \geqslant \Sigma.t$: return err
    $s \leftarrow \Sigma.\text{Share}(m_R)$
    return $\{s_i \mid i \in U\}$
```

Pseudorandom generator (Definition 5.1):

```
$\mathcal{L}^{G}_{\text{prg-real}}$
QUERY():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

```
$\mathcal{L}^{G}_{\text{prg-rand}}$
QUERY():
    $z \leftarrow \{0,1\}^{\lambda+\ell}$
    return $z$
```

Pseudorandom function (Definition 6.1):

```
$\mathcal{L}^{F}_{\text{prf-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{in}$):
    return $F(k, x)$
```

```
$\mathcal{L}^{F}_{\text{prf-rand}}$
$T$ := empty assoc. array

QUERY($x \in \{0,1\}^{in}$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{out}$
    return $T[x]$
```

Pseudorandom permutation (Definition 7.2):

```
$\mathcal{L}^F_{\text{prp-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{blen}$):
  return $F(k,x)$
```

```
$\mathcal{L}^F_{\text{prp-rand}}$
$T$ := empty assoc. array

QUERY($x \in \{0,1\}^{blen}$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
  return $T[x]$
```

Strong pseudorandom permutation (Definition 7.6):

```
$\mathcal{L}^F_{\text{sprp-real}}$
$k \leftarrow \{0,1\}^\lambda$

QUERY($x \in \{0,1\}^{blen}$):
  return $F(k,x)$

INVQUERY($y \in \{0,1\}^{blen}$):
  return $F^{-1}(k,y)$
```

```
$\mathcal{L}^F_{\text{sprp-real}}$
$T, T^{-1}$ := empty assoc. arrays

QUERY($x \in \{0,1\}^{blen}$):
  if $T[x]$ undefined:
    $y \leftarrow \{0,1\}^{blen} \setminus \text{range}(T)$
    $T[x] := y$; $T^{-1}[y] := x$
  return $T[x]$

INVQUERY($y \in \{0,1\}^{blen}$):
  if $T^{-1}[y]$ undefined:
    $x \leftarrow \{0,1\}^{blen} \setminus \text{range}(T^{-1})$
    $T^{-1}[y] := x$; $T[x] := y$
  return $T^{-1}[y]$
```

CPA security for symmetric-key encryption (Definition 8.1, Section 9.2):

```
$\mathcal{L}^\Sigma_{\text{cpa-L}}$
$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_L)$
  return $c$
```

```
$\mathcal{L}^\Sigma_{\text{cpa-R}}$
$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
  if $|m_L| \neq |m_R|$ return null
  $c := \Sigma.\text{Enc}(k, m_R)$
  return $c$
```

CPA$ security for symmetric-key encryption (Definition 8.2, Section 9.2):

```
$\mathcal{L}^\Sigma_{\text{cpa\$-real}}$
$k \leftarrow \Sigma.\text{KeyGen}$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, m)$
  return $c$
```

```
$\mathcal{L}^\Sigma_{\text{cpa\$-rand}}$
CHALLENGE($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.\mathcal{C}(|m|)$
  return $c$
```

CCA security for symmetric-key encryption (Definition 10.1):

```
$\mathcal{L}^{\Sigma}_{\text{cca-L}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
    if $|m_L| \neq |m_R|$ return null
    $c := \Sigma.\mathsf{Enc}(k, m_L)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

DEC($c \in \Sigma.\mathcal{C}$):
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\mathsf{Dec}(k, c)$
```

```
$\mathcal{L}^{\Sigma}_{\text{cca-R}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
    if $|m_L| \neq |m_R|$ return null
    $c := \Sigma.\mathsf{Enc}(k, m_R)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

DEC($c \in \Sigma.\mathcal{C}$):
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\mathsf{Dec}(k, c)$
```

CCA$ security for symmetric-key encryption (Definition 10.2):

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-real}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
    $c := \Sigma.\mathsf{Enc}(k, m)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

DEC($c \in \Sigma.\mathcal{C}$):
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\mathsf{Dec}(k, c)$
```

```
$\mathcal{L}^{\Sigma}_{\text{cca\$-rand}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{S} := \emptyset$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
    $c \leftarrow \Sigma.\mathcal{C}(|m|)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

DEC($c \in \Sigma.\mathcal{C}$):
    if $c \in \mathcal{S}$ return null
    return $\Sigma.\mathsf{Dec}(k, c)$
```

MAC (Definition 11.2):

```
$\mathcal{L}^{\Sigma}_{\text{mac-real}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$

GETMAC($m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\mathsf{MAC}(k, m)$

VER($m \in \Sigma.\mathcal{M}, t$):
    return $t \overset{?}{=} \Sigma.\mathsf{MAC}(k, m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{mac-fake}}$
$k \leftarrow \Sigma.\mathsf{KeyGen}$
$\mathcal{T} := \emptyset$

GETMAC($m \in \Sigma.\mathcal{M}$):
    $t := \Sigma.\mathsf{MAC}(k, m)$
    $\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
    return $t$

VER($m \in \Sigma.\mathcal{M}, t$):
    return $(m, t) \overset{?}{\in} \mathcal{T}$
```

Collision resistance (Definition 12.1):

```
$\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$
$H \leftarrow \mathcal{H}$

$\textsc{getH}()$:
  return $H$

$\textsc{hash}(x \in \{0,1\}^*)$:
  $y := H(x)$
  return $y$
```

```
$\mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$
$H \leftarrow \mathcal{H}$
$H^{-1}$ := empty assoc. array

$\textsc{getH}()$:
  return $H$

$\textsc{hash}(x \in \{0,1\}^*)$:
  $y := H(x)$
  if $H^{-1}[y]$ defined and $H^{-1}[y] \neq x$:
    self destruct
  $H^{-1}[y] := x$
  return $y$
```

Key agreement (Definition 14.4):

```
$\mathcal{L}^{\Sigma}_{\text{ka-real}}$
$\textsc{query}()$:
  $(t,K) \leftarrow \textsc{execprot}(\Sigma)$
  return $(t,K)$
```

```
$\mathcal{L}^{\Sigma}_{\text{ka-rand}}$
$\textsc{query}()$:
  $(t,K) \leftarrow \textsc{execprot}(\Sigma)$
  $K' \leftarrow \Sigma.\mathcal{K}$
  return $(t,K')$
```

Decisional Diffie-Hellman assumption (Definition 14.5):

```
$\mathcal{L}^{\mathbb{G}}_{\text{dh-real}}$
$\textsc{query}()$:
  $a,b \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^{ab})$
```

```
$\mathcal{L}^{\mathbb{G}}_{\text{dh-rand}}$
$\textsc{query}()$:
  $a,b,c \leftarrow \mathbb{Z}_n$
  return $(g^a, g^b, g^c)$
```

CPA security for public-key encryption (Definition 15.1):

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa-L}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

$\textsc{getpk}()$:
  return $pk$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  return $\Sigma.\text{Enc}(pk, m_L)$
```

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa-R}}$
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

$\textsc{getpk}()$:
  return $pk$

$\textsc{challenge}(m_L, m_R \in \Sigma.\mathcal{M})$:
  return $\Sigma.\text{Enc}(pk, m_R)$
```

CPA$ security for public-key encryption (Definition 15.2):

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-real}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
    return $pk$

CHALLENGE($m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\text{Enc}(pk, m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{pk-cpa\$-rand}}$

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$

GETPK():
    return $pk$

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

One-time secrecy for public-key encryption (Definition 15.4):

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

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
    return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
    $count := count + 1$
    if $count > 1$: return null
    return $\Sigma.\text{Enc}(pk, m_L)$
```

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

$(pk, sk) \leftarrow \Sigma.\text{KeyGen}$
$count := 0$

GETPK():
    return $pk$

CHALLENGE($m_L, m_R \in \Sigma.\mathcal{M}$):
    $count := count + 1$
    if $count > 1$: return null
    return $\Sigma.\text{Enc}(pk, m_R)$
```