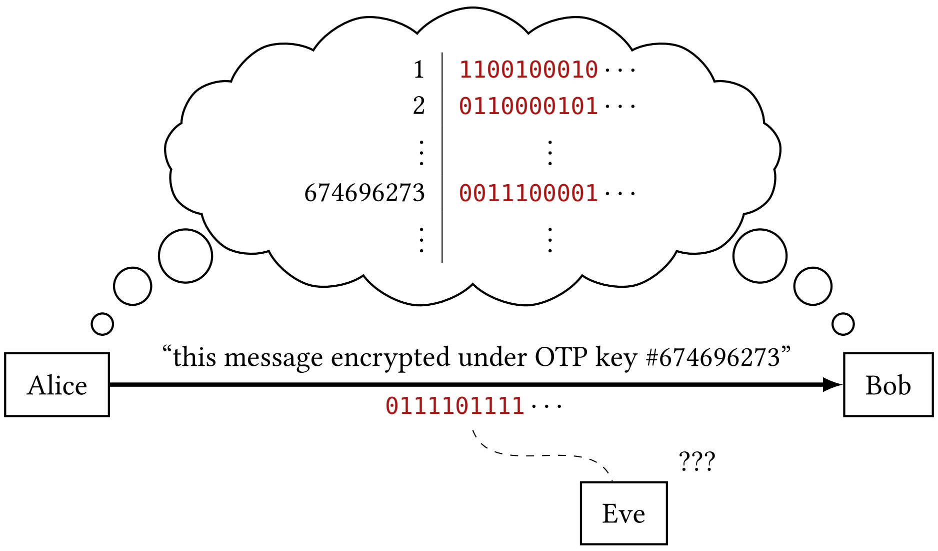
6 Pseudorandom Functions & Block Ciphers

Imagine if Alice & Bob had an *infinite* amount of shared randomness — not just a short key. They could split it up into λ-bit chunks and use each one as a one-time pad whenever they want to send an encrypted message of length λ.

Alice could encrypt by saying, "hey Bob, this message is encrypted with one-time pad using chunk #674696273 as key." Bob could decrypt by looking up location #674696273 in his copy of the shared randomness. As long as Alice doesn't repeat a key/chunk, an eavesdropper (who doesn't have the shared randomness) would learn nothing about the encrypted messages. Although Alice announces (publicly) *which* location/chunk was used as each one-time pad key, that information doesn't help the attacker know the *value* at that location.



It is silly to imagine an infinite amount of shared randomness. However, an exponential amount of something is often just as good as an infinite amount. A shared table containing "only" 2^λ one-time pad keys would be quite useful for encrypting as many messages as you could ever need.

A **pseudorandom function (PRF)** is a tool that allows Alice & Bob to achieve the effect of such an exponentially large table of shared randomness in practice. In this chapter we will explore PRFs and their properties. In a later chapter, after introducing new security definitions for encryption, we will see that PRFs can be used to securely encrypt *many* messages under the same key, following the main idea illustrated above.

6.1 Definition

Continuing our example, imagine a huge table of shared data stored as an array T, so the ith item is referenced as $T[i]$. Instead of thinking of i as an integer, we can also think of i as a binary string. If the array has 2^{in} items, then i will be an *in*-bit string. If the array contains strings of length "*out*", then the notation $T[i]$ is like a function that takes an input from $\{0,1\}^{in}$ and gives an output from $\{0,1\}^{out}$.

A pseudorandom function emulates the functionality of a huge array. It is a function F that takes an input from $\{0,1\}^{in}$ and gives an output from $\{0,1\}^{out}$. However, F also takes an additional argument called the **seed**, which acts as a kind of secret key.

The goal of a pseudorandom function is to "look like" a uniformly chosen array / lookup table. Such an array can be accessed through the LOOKUP subroutine of the following library:

```
for $x \in \{0,1\}^{in}$:
  $T[x] \leftarrow \{0,1\}^{out}$

LOOKUP($x \in \{0,1\}^{in}$):
  return $T[x]$
```

As you can see, this library initially fills up the array T with uniformly random data, and then allows the calling program to access any position in the array.

A pseudorandom function should produce indistinguishable behavior, when it is used with a uniformly chosen seed. More formally, the following library should be indistinguishable from the one above:

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

LOOKUP($x \in \{0,1\}^{in}$):
  return $F(k, x)$
```

Note that the first library samples $out \cdot 2^{in}$ bits uniformly at random (*out* bits for each of 2^{in} entries in the table), while the second library samples only λ bits (the same k is used for all invocations of F). Still, we are asking for the two libraries to be indistinguishable.

This is basically the definition of a PRF, with one technical caveat. We want to allow situations like $in \geqslant \lambda$, but in those cases the first library runs in exponential time. It is generally convenient to build our security definitions with libraries that run in polynomial time.[1] We fix this by taking advantage of the fact that, no matter how big the table T is meant to be, a polynomial-time calling program will only access a polynomial amount of it. In some sense it is "overkill" to actually populate the entire table T upfront. Instead, we can populate T in a lazy / on-demand way. T initially starts uninitialized, and its values are only assigned as the calling program requests them. This changes *when* each $T[x]$ is sampled (if at all), but does not change *how* it is sampled (*i.e.*, uniformly & independently). This also changes T from being a typical array to being an *associative array* ("hash table" or "dictionary" data structure), since it only maps a subset of $\{0,1\}^{in}$ to values in $\{0,1\}^{out}$.

[1]When we use a pseudorandom function as a component in other constructions, the libraries for PRF security will show up as *calling programs* of other libraries. The definition of indistinguishability requires all calling programs to run in polynomial time.

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:*

$\mathcal{L}^F_{\text{prf-real}}$

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

LOOKUP($x \in \{0,1\}^{in}$):
    return $F(k,x)$
```

$\mathcal{L}^F_{\text{prf-rand}}$

```
$T :=$ empty assoc. array

LOOKUP($x \in \{0,1\}^{in}$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{out}$
    return $T[x]$
```

Discussion, Pitfalls

The name "pseudorandom *function*" comes from the perspective of viewing T not as an (associative) array, but as a function $T : \{0,1\}^{in} \to \{0,1\}^{out}$. There are $2^{out \cdot 2^{in}}$ possible functions for T (an incredibly large number), and $\mathcal{L}_{\text{prf-rand}}$ chooses a "random function" by uniformly sampling its truth table as needed.

For each possible seed k, the residual function $F(k,\cdot)$ is also a function from $\{0,1\}^{in} \to \{0,1\}^{out}$. There are "only" 2^λ possible functions of this kind (one for each choice of k), and $\mathcal{L}_{\text{prf-real}}$ chooses one of these functions randomly. In both cases, the libraries give the calling program input/output access to the function that was chosen. You can think of this in terms of the picture from Section 5.1, but instead of strings, the objects are functions.

Note that even in the case of a "random function" ($\mathcal{L}_{\text{prf-rand}}$), the function T itself is still **deterministic**! To be precise, this library chooses a deterministic function, uniformly, from the set of all possible deterministic functions. But once it makes this choice, the input/output behavior of T is fixed. If the calling program calls LOOKUP twice with the same x, it receives the same result. The same is true in $\mathcal{L}_{\text{prf-real}}$, since F is a deterministic function and k is fixed throughout the entire execution. To avoid this very natural confusion, it is perhaps better to think in terms of "randomly initialized lookup tables" rather than "random functions."

How NOT to Build a PRF

We can appreciate the challenges involved in building a PRF by looking at a natural approach that doesn't quite work.

Example *Suppose we have a length-doubling PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ and try to use it to construct a PRF F as follows:*

```
$F(k,x)$:
    return $G(k) \oplus x$
```

You might notice that all we have done is rename the encryption algorithm of "pseudo-OTP" (Construction 5.2). We have previously argued that this algorithm is a secure method for one-time encryption, and that the resulting ciphertexts are pseudorandom. Is this enough for a secure PRF? No, we can attack the security of this PRF.

Attacking F means designing distinguisher that behaves as differently as possible in the presence of the two $\mathcal{L}^F_{\text{prf-}\star}$ libraries. We want to show that F is insecure even if *G is an* excellent PRG. *We should not try to base our attack on distinguishing outputs of G from random. Instead, we must try to* ***break the inappropriate way that G is used*** *to construct a PRF.*

The distinguisher must use the interface of the $\mathcal{L}_{\text{prf-}\star}$ libraries — i.e., *make some calls to the LOOKUP subroutine and output 0 or 1 based on the answers it gets. The LOOKUP subroutine takes an argument, so the distinguisher has to choose 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 5.3. So we should be looking for a calling program that makes more than one call to LOOKUP.*

If we make two calls to LOOKUP — say, on inputs x_1 and x_2 — the responses from $\mathcal{L}_{\text{prf-real}}$ 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? They actually have a pattern that the calling program can notice: their XOR is always $x_1 \oplus x_2$, a value that is already known to the calling program.*

We can condense all of our observations into the following distinguisher:

```
𝒜
pick $x_1, x_2 \in \{0,1\}^{2\lambda}$ arbitrarily so that $x_1 \neq x_2$
$z_1$ := LOOKUP($x_1$)
$z_2$ := LOOKUP($x_2$)
return $z_1 \oplus z_2 \overset{?}{=} x_1 \oplus x_2$
```

Let's compute its advantage in distinguishing $\mathcal{L}^F_{\text{prf-real}}$ from $\mathcal{L}^F_{\text{prf-rand}}$ by considering $\mathcal{A}$'s behavior when linked to these two libraries:

```
𝒜
pick $x_1 \neq x_2 \in \{0,1\}^{2\lambda}$
$z_1$ := LOOKUP($x_1$)
$z_2$ := LOOKUP($x_2$)
return $z_1 \oplus z_2 \overset{?}{=} x_1 \oplus x_2$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$
$k \leftarrow \{0,1\}^\lambda$

LOOKUP($x$):
    return $G(k) \oplus x$ // $F(k, x)$
```

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 $\mathcal{A}$ always outputs 1. That is,*

$$\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-real}} \Rightarrow 1] = 1.$$

```
𝒜
pick $x_1 \neq x_2 \in \{0,1\}^{2\lambda}$
$z_1$ := LOOKUP($x_1$)
$z_2$ := LOOKUP($x_2$)
return $z_1 \oplus z_2 \overset{?}{=} x_1 \oplus x_2$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$
$T$ := empty assoc. array

LOOKUP($x$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{2\lambda}$
    return $T[x]$
```

When $\mathcal{A}$ is linked to $\mathcal{L}^F_{\text{prf-rand}}$, the responses of the two calls to LOOKUP *will be chosen uniformly and independently because* LOOKUP *is being called on distinct inputs. Consider the moment in time when the second call to* LOOKUP *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 exactly the 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 certainly non-negligible since it doesn't even approach 0. This shows that F is not a secure PRF.

At a more philosophical level, we wanted to identify exactly how G is being used in an inappropriate way. The PRG security libraries guarantee security when G's seed is chosen freshly for each call to G. This construction of F violates that rule and allows the same seed to be used twice in different calls to G, where the results are supposed to look independent.

This example shows the challenge of building a PRF. Even though we know how to make any *individual* output pseudorandom, it is difficult to make all outputs collectively appear *independent*, when in reality they are derived from a single short seed.

6.2 PRFs vs PRGs; Variable-Hybrid Proofs

In this section we show that a PRG can be used to construct a PRF, **and vice-versa.** The construction of a PRG from PRF is practical, and is one of the more common ways to obtain a PRG in practice. The construction of a PRF from PRG is more of theoretical interest and does not reflect how PRFs are designed in practice.

Constructing a PRG from a PRF

As promised, a PRF can be used to construct a PRG. The construction is quite natural. For simplicity, suppose we have a PRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ (*i.e.*, *in* = *out* = λ). We can build a length-doubling PRG in the following way:

Construction 6.2 (Counter PRG)

```
$\underline{G(s)}$:
  $x := F(s, 0\cdots00)$
  $y := F(s, 0\cdots01)$
  return $x \| y$
```

There is nothing particularly special about the inputs $0\cdots00$ and $0\cdots01$ to F. All that matters is that they are distinct. The construction can be extended to easily give more than 2 blocks of output, by treating the input to F as a simple counter (hence the name of this construction).

The guarantee of a PRF is that when its seed is chosen uniformly and it is invoked on distinct inputs, its outputs look independently uniform. In particular, its output on inputs $0\cdots00$ and $0\cdots01$ are indistinguishable from uniform. Hence, concatenating them gives a string which is indistinguishable from a uniform 2λ-bit string.

That really is all there is to the security of this construction, but unfortunately there is a slight technical issue which makes the security proof more complicated than you might guess. We will have to introduce a new technique of **variable hybrids** to cope with it.

Claim 6.3 *If F is a secure PRF, then the counter PRG construction G above is a secure PRG.*

Proof In order to prove that G is a secure PRG, we must prove that the following libraries are indistinguishable:

```
$\mathcal{L}^G_{\text{prg-real}}$
QUERY():
    $s \leftarrow \{0,1\}^\lambda$
    $x := F(s, 0\cdots00)$        }
    $y := F(s, 0\cdots01)$        } // $G(s)$
    return $x\|y$                  }
```

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

During the proof, we are allowed to use the fact that F is a secure PRF. That is, we can use the fact that the following two libraries are indistinguishable:

```
$\mathcal{L}^F_{\text{prf-real}}$
$k \leftarrow \{0,1\}^\lambda$

LOOKUP($x \in \{0,1\}^{in}$):
    return $F(k, x)$
```

```
$\mathcal{L}^F_{\text{prf-rand}}$
$T :=$ empty assoc. array

LOOKUP($x \in \{0,1\}^{in}$):
    if $T[x]$ undefined:
        $T[x] \leftarrow \{0,1\}^{out}$
    return $T[x]$
```

The inconvenience in the proof stems from a mismatch of the s variable in $\mathcal{L}_{\text{prg-real}}$ and the k variable in $\mathcal{L}_{\text{prf-real}}$. In $\mathcal{L}_{\text{prg-real}}$, s is local to the QUERY subroutine. Over the course of an execution, s will take on many values. Since s is used as the PRF seed, we must write the calls to F in terms of the LOOKUP subroutine of $\mathcal{L}_{\text{prf-real}}$. But in $\mathcal{L}_{\text{prf-real}}$ the PRF seed is fixed for the entire execution. In other words, we can only use $\mathcal{L}_{\text{prf-real}}$ to deal with a single PRF seed at a time, but $\mathcal{L}_{\text{prg-real}}$ deals with many PRG seeds at a time.

To address this, we will have to apply the security of F (*i.e.*, replace $\mathcal{L}_{\text{prf-real}}$ with $\mathcal{L}_{\text{prf-rand}}$) *many times* during the proof — in fact, once for every call to QUERY made by the calling program. Previous security proofs had a fixed number of hybrid steps (*e.g.*, the proof of Claim 5.5 used 7 hybrid libraries to show $\mathcal{L}_{\text{prg-real}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-7}} \approx \mathcal{L}_{\text{prg-rand}}$). This proof will have a **variable number of hybrids that depends on the calling program.** Specifically, we will prove

$$\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}q} \approx \mathcal{L}^G_{\text{prg-rand}},$$

where q is the number of times the calling program calls QUERY.

Don't be overwhelmed by all these hybrids. They all follow a simple pattern. In fact, the ith hybrid looks like this:

```
$\mathcal{L}_{\text{hyb-}i}$:
$count := 0$

QUERY():
  $count := count + 1$
  if $count \leqslant \boxed{i}$:
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0\cdots00)$
    $y := F(s, 0\cdots01)$
    return $x\|y$
```

In other words, the hybrid libraries all differ in the value $\boxed{i}$ that is inserted into the code above. If you're familiar with C compilers, think of this as adding "`#define i 427`" to the top of the code above, to obtain $\mathcal{L}_{\text{hyb-427}}$.

First note what happens for extreme choices of $\boxed{i}$:

- In $\mathcal{L}_{\text{hyb-0}}$, the if-branch is never taken ($count \leqslant 0$ is never true). This library behaves exactly like $\mathcal{L}^{G}_{\text{prg-real}}$ by giving PRG outputs on every call to QUERY.

- If q is the total number of times that the calling program calls QUERY, then in $\mathcal{L}_{\text{hyb-}q}$, the if-branch is always taken ($count \leqslant q$ is always true). This library behaves exactly like $\mathcal{L}^{G}_{\text{prg-rand}}$ by giving truly uniform output on every call to QUERY.

In general, $\mathcal{L}_{\text{hyb-}i}$ will respond to the first i calls to QUERY by giving truly random output. It will respond to all further calls by giving outputs of our PRG construction.

We have argued that $\mathcal{L}^{G}_{\text{prg-real}} \equiv \mathcal{L}_{\text{hyb-0}}$ and $\mathcal{L}^{G}_{\text{prg-rand}} \equiv \mathcal{L}_{\text{hyb-}q}$. To complete the proof, we must show that $\mathcal{L}_{\text{hyb-}(i-1)} \approx \mathcal{L}_{\text{hyb-}i}$ for all i. The main reason for going to all this trouble of defining so many hybrid libraries is that $\mathcal{L}_{\text{hyb-}(i-1)}$ and $\mathcal{L}_{\text{hyb-}i}$ are completely identical except in how they respond to the ith call to QUERY. This difference involves a single call to the PRG (and hence a single PRF seed), which allows us to apply the security of the PRF.

In more detail, let i be arbitrary, and consider the following sequence of steps starting with $\mathcal{L}_{\text{hyb-}(i-1)}$:

```
$count := 0$

QUERY():
  $count := count + 1$
  if $count < i$:
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  elsif $count = i$:
    $s^* \leftarrow \{0,1\}^{\lambda}$
    $x := F(s^*, 0\cdots00)$
    $y := F(s^*, 0\cdots01)$
    return $x \| y$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0\cdots00)$
    $y := F(s, 0\cdots01)$
    return $x \| y$
```

We have taken $\mathcal{L}_{\text{hyb-}(i-1)}$ and simply expanded the else-branch ($count \geqslant i$) into two subcases ($count = i$ and $count > i$). However, both cases lead to the same block of code (apart from a change to a local variable's name), so the change has no effect on the calling program.

```
$count := 0$

QUERY():
  $count := count + 1$
  if $count < i$:
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  elsif $count = i$:
    $x :=$ LOOKUP($0\cdots00$)
    $y :=$ LOOKUP($0\cdots01$)
    return $x \| y$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0\cdots00)$
    $y := F(s, 0\cdots01)$
    return $x \| y$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$
$k \leftarrow \{0,1\}^{\lambda}$

LOOKUP($x$):
  return $F(k, x)$
```

We have factored out the calls to F that use seed s^* (corresponding to the $count = i$ case) in terms of $\mathcal{L}_{\text{prf-real}}$. This change no effect on the calling program.

```
$count := 0$

QUERY():
  $count := count + 1$
  if $count < i$ :
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  elsif $count = i$ :
    $x :=$ LOOKUP($0 \cdots 00$)
    $y :=$ LOOKUP($0 \cdots 01$)
    return $x \| y$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0 \cdots 00)$
    $y := F(s, 0 \cdots 01)$
    return $x \| y$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$

$T :=$ empty assoc. array

LOOKUP($x$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{\lambda}$
  return $T[x]$
```

From the fact that F is a secure PRF, we can replace $\mathcal{L}^F_{\text{prf-real}}$ with $\mathcal{L}^F_{\text{prf-rand}}$, and the overall change is indistinguishable.

```
$count := 0$

QUERY():
  $count := count + 1$
  if $count < i$ :
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  elsif $count = i$ :
    $x :=$ LOOKUP($0 \cdots 00$)
    $y :=$ LOOKUP($0 \cdots 01$)
    return $x \| y$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0 \cdots 00)$
    $y := F(s, 0 \cdots 01)$
    return $x \| y$
```

$\diamond$

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

Since $count = i$ happens only once, only two calls to LOOKUP will be made across the entire lifetime of the library, and they are on distinct inputs. Therefore, the if-branch in LOOKUP will always be taken, and T is never needed (it is only needed to "remember" values and give the same answer when the same x is used twice as argument to LOOKUP). Simplifying the library therefore has no effect on the calling program:

```
$count := 0$

QUERY():
  $count := count + 1$
  if $count < i$:
    $r \leftarrow \{0,1\}^{2\lambda}$
    return $r$
  elsif $count = i$:
    $x \leftarrow \{0,1\}^{\lambda}$
    $y \leftarrow \{0,1\}^{\lambda}$
    return $x \| y$
  else:
    $s \leftarrow \{0,1\}^{\lambda}$
    $x := F(s, 0 \cdots 00)$
    $y := F(s, 0 \cdots 01)$
    return $x \| y$
```

Inlining the subroutine has no effect on the calling program. The resulting library responds with uniformly random output to the first i calls to QUERY, and responds with outputs of our PRG G to the others. Hence, this library has identical behavior to $\mathcal{L}_{\text{hyb-}i}$.

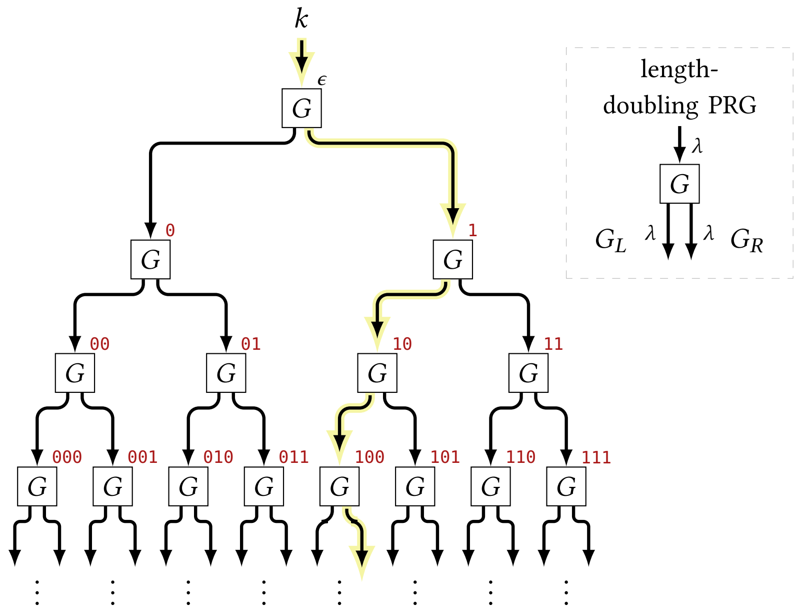
We showed that $\mathcal{L}_{\text{hyb-}(i-1)} \approx \mathcal{L}_{\text{hyb-}i}$, and therefore:

$$\mathcal{L}^G_{\text{prg-real}} \equiv \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}q} \equiv \mathcal{L}^G_{\text{prg-rand}}$$

This shows that $\mathcal{L}^G_{\text{prg-real}} \approx \mathcal{L}^G_{\text{prg-rand}}$, so G is a secure PRG. ■

★ A Theoretical Construction of a PRF from a PRG

We have already seen that it is possible to feed the output of a PRG back into the PRG again, to extend its stretch (Claim 5.7). This is done by making a long chain (like a linked list) of PRGs. The trick to constructing a PRF from a PRG is to chain PRGs together in a **binary tree** (similar to Exercise 5.8(a)). The leaves of the tree correspond to final outputs of the PRF. If we want a PRF with an exponentially large domain (*e.g.*, $in = \lambda$), the binary tree itself is exponentially large! However, it is still possible to compute any individual leaf efficiently by simply traversing the tree from root to leaf. This tree traversal itself is the PRF algorithm. This construction of a PRF is due to Goldreich, Goldwasser, and Micali, in the paper that defined the concept of a PRF.



Imagine a complete binary tree of height *in* (*in* will be the input length of the PRF). Every node in this tree has a *position* which can be written as a binary string. Think of a node's position as the directions to get there starting at the root, where a 0 means "go left" and 1 means "go right." For example, the root has position ϵ (the empty string), the right child of the root has position 1, etc.

The PRF construction works by assigning a *label* to every node in the tree, using the a length-doubling PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. For convenience, we will write $G_L(k)$ and $G_R(k)$ to denote the first λ bits and last λ bits of $G(k)$, respectively. Labels in the tree are λ-bit strings, computed according to the following two rules:

1. The root node's label is the PRF seed.
2. If the node at position p has label v, then its left child (at position $p\|0$) gets label $G_L(v)$, and its right child (at position $p\|1$) gets label $G_R(v)$.

In the picture above, a node's label is the string being sent on its incoming edge. The tree has 2^{in} leaves, whose positions are the strings $\{0,1\}^{in}$. We define $F(k,x)$ to be the label of node/leaf x. To compute this label, we can traverse the tree from root to leaf, taking left and right turns at each node according to the bits of x and computing the labels along that path according to the labeling rule. In the picture above, the highlighted path corresponds to the computation of $F(k, 1001\cdots)$.

It is important to remember that the binary tree is a useful conceptual tool, but it is exponentially large in general. Running the PRF on some input does not involve computing labels for the entire tree, only along a single path from root to leaf.

Construction 6.4 (GGM PRF)

```
in = arbitrary
out = λ

F(k, x ∈ {0,1}^in):
  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 6.5 *If G is a secure PRG, then Construction 6.4 is a secure PRF.*

Proof We prove the claim using a sequence of hybrids. The number of hybrids in this case depends on the input-length parameter *in*. The hybrids are defined as follows:

```
$\mathcal{L}_{\text{hyb-}d}$
T := empty assoc. array

QUERY(x):
  p := first d bits of x
  if T[p] undefined:
    T[p] ← {0,1}^λ
  v := T[p]
  for i = d + 1 to in:
    if x_i = 0 then v := G_L(v)
    if x_i = 1 then v := G_R(v)
  return v
```

The hybrids differ only in their hard-coded value of d. We will show that

$$\mathcal{L}^F_{\text{prf-real}} \equiv \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}in} \equiv \mathcal{L}^F_{\text{prf-rand}}.$$

We first start by understanding the behavior of $\mathcal{L}_{\text{hyb-}d}$ for extreme choices of d. Simplifications to the code are shown on the right.

```
$\mathcal{L}_{\text{hyb-0}}$
T := empty assoc. array                 k := undefined
                                        // k is alias for T[ϵ]
LOOKUP(x):
  p := first 0 bits of x                p = ϵ
  if T[p] undefined:                    if k undefined:
    T[p] ← {0,1}^λ                        k ← {0,1}^λ
  v := T[p]                           ⎫
  for i = 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)
```

In $\mathcal{L}_{\text{hyb-0}}$, we always have $p = \epsilon$, so the only entry of T that is accessed is $T[\epsilon]$. Then renaming $T[\epsilon]$ to k, we see that $\mathcal{L}_{\text{hyb-0}} \equiv \mathcal{L}^F_{\text{prf-real}}$. The only difference is when the PRF seed k ($T[\epsilon]$) is sampled: eagerly at initialization time in $\mathcal{L}^F_{\text{prf-real}}$ vs. at the last possible minute in $\mathcal{L}_{\text{hyb-0}}$.

```
$\mathcal{L}_{\textsf{hyb-}in}$

$T$ := empty assoc. array

LOOKUP($x$):
    $p$ := 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 $in$:                 } // unreachable
        if $x_i = 0$ then $v := G_L(v)$
        if $x_i = 1$ then $v := G_R(v)$
    return $v$                                return $T[x]$
```

In $\mathcal{L}_{\textsf{hyb-}in}$, we always have $p = x$ and the body of the for-loop is always unreachable. In that case, it is easy to see that $\mathcal{L}_{\textsf{hyb-}in}$ has identical behavior to $\mathcal{L}^F_{\textsf{prf-rand}}$.

The general pattern is that $\mathcal{L}_{\textsf{hyb-}d}$ "chops off" the top d levels of the conceptual binary tree. When computing the output for some string x, we don't start traversing the tree from the root but rather d levels down the tree, at the node whose position is the d-bit prefix of x (called p in the library). We initialize the label of this node as a uniform value (unless it has already been defined), and then continue the traversal to the leaf x.

To finish the proof, we show that $\mathcal{L}_{\textsf{hyb-}(d-1)}$ and $\mathcal{L}_{\textsf{hyb-}d}$ are indistinguishable:

```
$T$ := empty assoc. array

LOOKUP($x$):
    $p$ := first $d-1$ 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 $d$ bits of $x$
    $v := T[p']$
    for $i = d + 1$ to $in$:
        if $x_i = 0$ then $v := G_L(v)$
        if $x_i = 1$ then $v := G_R(v)$
    return $v$
```

The library that is shown here is different from $\mathcal{L}_{\textsf{hyb-}(d-1)}$ in the highlighted parts. However, these differences have no effect on the calling program. The library here advances $d-1$ levels down the tree (to the node at location p), initializes that node's label as a uniform value, then computes the labels for *both* its children, and finally continues computing labels toward the leaf. The only significant difference from $\mathcal{L}_{\textsf{hyb-}(d-1)}$ is that it computes the labels of *both* of p's children, even though only one is on the path to x. Since it computes the label correctly, though, it makes no difference when (or if) this extra label is computed.

```
$T$ := empty assoc. array

LOOKUP($x$):
  $p$ := first $d-1$ bits of $x$
  if $T[p]$ undefined:
    $T[p\|0] \| T[p\|1]$ := QUERY()
  $p'$ := first $d+1$ bits of $x$
  $v := T[p']$
  for $i = d+1$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$
```

$\diamond$

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

We have factored out the body of the if-statement in terms of $\mathcal{L}^G_{\text{prg-real}}$ since it involves an call to G on uniform input. Importantly, the seed to G (called $T[p]$ in the previous hybrid) was not used anywhere else — it was a string of length $d-1$ while the library only reads $T[p']$ for p' of length d. The change has no effect on the calling program.

```
$T$ := empty assoc. array

LOOKUP($x$):
  $p$ := first $d-1$ bits of $x$
  if $T[p]$ undefined:
    $T[p\|0] \| T[p\|1]$ := QUERY()
  $p'$ := first $d+1$ bits of $x$
  $v := T[p']$
  for $i = d+1$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$
```

$\diamond$

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

We have applied the security of G and replaced $\mathcal{L}_{\text{prg-real}}$ with $\mathcal{L}_{\text{prg-rand}}$. The change is indistinguishable.

```
$T$ := empty assoc. array

LOOKUP($x$):
  $p$ := first $d-1$ 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 $i = d+1$ to $in$:
    if $x_i = 0$ then $v := G_L(v)$
    if $x_i = 1$ then $v := G_R(v)$
  return $v$
```

We have inlined $\mathcal{L}_{\text{prg-rand}}$ and split the sampling of 2λ bits into two separate statements sampling λ bits each. In this library, we advance d levels down the tree, assign a uniform label to a node (and its sibling), and then proceed to the leaf applying G as usual. The only difference between this library and $\mathcal{L}_{\text{hyb-}d}$ is that we sample the label of a node that is not on our direct path. But since we sample it uniformly, it doesn't matter when (or if) that extra value is sampled. Hence, this library has identical behavior to $\mathcal{L}_{\text{hyb-}d}$.

We showed that $\mathcal{L}_{\text{hyb-}(d-1)} \approx \mathcal{L}_{\text{hyb-}d}$. Putting everything together, we have:

$$\mathcal{L}^F_{\text{prf-real}} \equiv \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}in} \equiv \mathcal{L}^F_{\text{prf-rand}}.$$

Hence, F is a secure PRF. ■

6.3 Block Ciphers (Pseudorandom Permutations)

After fixing the seed of a PRF, it computes a function from $\{0,1\}^{in}$ to $\{0,1\}^{out}$. Let's consider the case where $in = out$. Some functions from $\{0,1\}^{in}$ to $\{0,1\}^{out}$ are invertible, which leads to the question of whether a PRF might realize such a function and be invertible (with knowledge of the seed). In other words, what if it were possible to determine x when given k and $F(k,x)$? While this would be a convenient property, it is not guaranteed by the PRF security definition, even in the case of $in = out$. A function from $\{0,1\}^{in}$ to $\{0,1\}^{out}$ chosen at random is unlikely to have an inverse, therefore a PRF instantiated with a random key is unlikely to have an inverse.

A **pseudorandom permutation (PRP)** — also called a **block cipher** — is essentially a PRF that is guaranteed to be invertible for every choice of seed. We use both terms (PRP and block cipher) interchangeably. The term "permutation" refers to the fact that, for every k, the function $F(k,\cdot)$ should be a permutation of $\{0,1\}^{in}$. Instead of requiring a PRP to be indistinguishable from a randomly chosen function, we require it to be indistinguishable from a randomly chosen *invertible* function.[2] This means we must modify one of the libraries from the PRF definition. Instead of populating the associative array T with uniformly random values, it chooses uniformly random *but distinct* values. As long as T gives distinct outputs on distinct inputs, it is consistent with some invertible function. The library guarantees distinctness by only sampling values that it has not previously assigned. Thinking of an associative array T as a key-value store, we use the notation T.values to denote the set of values stored in T.

Definition 6.6 (PRP syntax) *Let $F : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ be a deterministic function. We refer to blen as the **blocklength** of F and any element of $\{0,1\}^{blen}$ as a **block**.*

*We call F a **secure pseudorandom permutation (PRP)** (**block cipher**) if the following two conditions hold:*

1. *(Invertible given k) There is a function $F^{-1} : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ satisfying*
$$F^{-1}(k, F(k,x)) = x,$$
for all $k \in \{0,1\}^\lambda$ and all $x \in \{0,1\}^{blen}$.

2. *(Security) $\mathcal{L}^F_{\text{prp-real}} \approx \mathcal{L}^F_{\text{prp-rand}}$, where:*

```
$\mathcal{L}^F_{\text{prp-real}}$
$k \leftarrow \{0,1\}^\lambda$

LOOKUP($x \in \{0,1\}^{blen}$):
  return $F(k,x)$
```

```
$\mathcal{L}^F_{\text{prp-rand}}$
$T$ := empty assoc. array

LOOKUP($x \in \{0,1\}^{blen}$):
  if $T[x]$ undefined:
    $T[x] \leftarrow \{0,1\}^{blen} \setminus T.\text{values}$
  return $T[x]$
```

[2] As we will see later, the distinction between randomly chosen function and randomly chosen *invertible* function is not as significant as it might seem.

"T.values" refers to the set $\{v \mid \exists x : T[x] = v\}$.

The changes from the PRF definition are highlighted in yellow. In particular, the $\mathcal{L}_{\text{prp-real}}$ and $\mathcal{L}_{\text{prf-real}}$ libraries are identical.

Discussion, Pitfalls

In the definition, both the functions F and F^{-1} take the seed k as input. Therefore, only someone with k can invert the block cipher. Think back to the definition of a PRF — without the seed k, it is hard to compute $F(k, x)$. A block cipher has a forward and reverse direction, and computing *either* of them is hard without k!

6.4 Relating PRFs and Block Ciphers

In this section we discuss how to obtain PRFs from PRPs/block ciphers, and vice-versa.

Switching Lemma (PRPs are PRFs, Too!)

Imagine you can query a PRP on chosen inputs (as in the $\mathcal{L}_{\text{prp-real}}$ library), and suppose the blocklength of the PRP is $blen = \lambda$. You would only be able to query that PRP on a *negligible fraction* of its exponentially large input domain. It seems unlikely that you would even be able to tell that it was a PRP (*i.e.*, an invertible function) rather than a PRF (an unrestricted function).

This idea can be formalized as follows.

Lemma 6.7 (PRP switching) *Let $\mathcal{L}_{\text{prf-rand}}$ and $\mathcal{L}_{\text{prp-rand}}$ be defined as in Definitions 6.1 & 6.6, with parameters $in = out = blen = \lambda$ (so that the interfaces match up). Then $\mathcal{L}_{\text{prf-rand}} \approx \mathcal{L}_{\text{prp-rand}}$.*

Proof Recall the replacement-sampling lemma, Lemma 4.11, which showed that the following libraries are indistinguishable:

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

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 $r$
```

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

LOOKUP($x \in \{0,1\}^\lambda$):
  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 (consider R playing the role of T.values in $\mathcal{L}_{\text{prp-rand}}$). Then from Lemma 4.11, 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. ■

Using the switching lemma, we can conclude that every PRP (with *blen* = λ) is also a PRF:

Corollary 6.8 *Let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a secure PRP (with blen = λ). Then F is also a secure PRF.*

Proof As we have observed above, $\mathcal{L}^F_{\text{prf-real}}$ and $\mathcal{L}^F_{\text{prp-real}}$ are literally the same library. Since F is a secure PRP, $\mathcal{L}^F_{\text{prp-real}} \approx \mathcal{L}^F_{\text{prp-rand}}$. Finally, by the switching lemma, $\mathcal{L}^F_{\text{prp-rand}} \approx \mathcal{L}^F_{\text{prf-rand}}$. Putting everything together:

$$\mathcal{L}^F_{\text{prf-real}} \equiv \mathcal{L}^F_{\text{prp-real}} \approx \mathcal{L}^F_{\text{prp-rand}} \approx \mathcal{L}^F_{\text{prf-rand}},$$

hence F is a secure PRF. ■

Keep in mind that the switching lemma applies only when the blocklength is sufficiently large (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 sampling with long (length-λ) strings (look at the proof of Lemma 4.11 to recall why). Exercise 6.14 asks you to show that a random permutation over a *small domain* can be distinguished from a random (unconstrained) function; so, a PRP with a small blocklength is **not** a PRF.

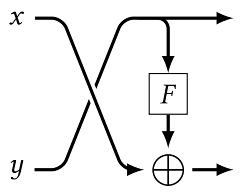
Constructing a PRP from a PRF: The Feistel Construction

How can you build an *invertible* block cipher out of a PRF that is not necessarily invertible? In this section, we show a simple technique called the **Feistel construction** (named after IBM cryptographer Horst Feistel).

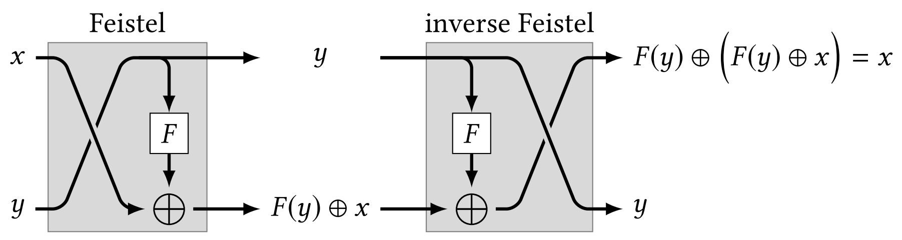
The main idea in the Feistel construction is to convert a not-necessarily-invertible function $F : \{0,1\}^n \to \{0,1\}^n$ into an invertible function $F^* : \{0,1\}^{2n} \to \{0,1\}^{2n}$. The function F^* is called the **Feistel round with round function** F and is defined as follows:

Construction 6.9 (Feistel round)

```
$F^*(x \| y)$:
    // each of $x, y$ are $n$ bits
    return $y \| (F(y) \oplus x)$
```



No matter what F is, its Feistel round F^* is invertible. Not only that, but its inverse is a kind of "mirror image" of F^*:



Note how both the forward and inverse Feistel rounds use F in the forward direction!

Example *Let's see what happens in the Feistel construction with a trivial round function. Consider the constant function $F(y) = 0^n$, which is the "least invertible" function imaginable. The Feistel construction gives:*

$$\begin{aligned} F^*(x\|y) &= y\|(F(y) \oplus x) \\ &= y\|(0^n \oplus x) \\ &= y\|x \end{aligned}$$

The result is a function that simply switches the order of its halves — clearly invertible.

Example *Let's try another simple round function, this time the identity function $F(y) = y$. The Feistel construction gives:*

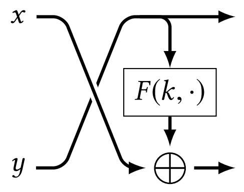
$$\begin{aligned} F^*(x\|y) &= y\|(F(y) \oplus x) \\ &= y\|(y \oplus x) \end{aligned}$$

This function is invertible because given y and $y \oplus x$ we can solve for x as $y \oplus (y \oplus x)$. You can verify that this is what happens when you plug F into the inverse Feistel construction.

We can also consider using a round function F that has a key/seed. The result will be an F^* that also takes a seed. For every seed k, $F^*(k, \cdot)$ will have an inverse (which looks like its mirror image).

Construction 6.10 (Keyed Feistel)

```
$F^*(k, x\|y)$:
    return $y\|(F(k,y) \oplus x)$
```



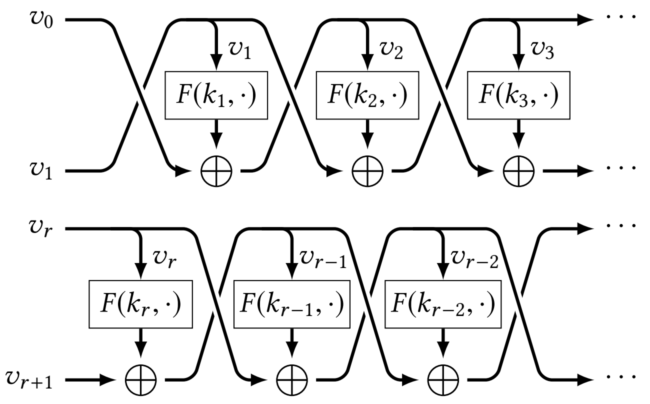
Now suppose F is a secure PRF and we use it as a Feistel round function, to obtain a keyed function F^*. Since $F^*(k, \cdot)$ is invertible for every k, and since F^* uses a secure PRF in some way, you might be tempted to claim that F^* is a secure PRP. Unfortunately, it is not! The output of F^* contains half of its input, making it quite trivial to break the PRP-security of F^*.

We can avoid this trivial attack by performing several Feistel rounds in succession, resulting in a construction called a **Feistel cipher**. At each round, we can even use a different key to the round function. If we use k_1 in the first round, k_2 in the second round, and so on, then $k_1, k_2, \ldots$ is called the **key schedule** of the Feistel cipher. The formal definition of an r-round Feistel cipher is given below:

Construction 6.11 (Feistel cipher)

```
$\mathbb{F}_r\big((k_1,\ldots,k_r), v_0\|v_1\big)$:
  for $i = 1$ to $r$:
    $v_{i+1} := F(k_i, v_i) \oplus v_{i-1}$
  return $v_r\|v_{r+1}$

$\mathbb{F}_r^{-1}\big((k_1,\ldots,k_r), v_r\|v_{r+1}\big)$:
  for $i = r$ downto 1:
    $v_{i-1} := F(k_i, v_i) \oplus v_{i+1}$
  return $v_0\|v_1$
```



Because each round is invertible (given the appropriate round key), the overall Feistel cipher is also invertible. Note that the inverse of the Feistel cipher uses inverse Feistel rounds and reverses the order of the key schedule.

Surprisingly, a 3-round Feistel cipher can actually be secure, although a 2-round Feistel cipher is never secure (see the exercises). More precisely: when F is a secure PRF with $in = out = \lambda$, then using F as the round function of a 3-round Feistel cipher results in a secure PRP. The Feistel cipher has blocklength 2λ, and it has a key of length 3λ (3 times longer than the key for F). Implicitly, this means that the three round keys are chosen independently.

Theorem 6.12 (Luby-Rackoff) *If $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ is a secure PRF, then the 3-round Feistel cipher $\mathbb{F}_3$ (Construction 6.11) is a secure PRP.*

Unfortunately, the proof of this theorem is beyond the scope of this book.

6.5 PRFs and Block Ciphers in Practice

Block ciphers are one of the cornerstones of cryptography in practice today. We have shown how (at least in principle) block ciphers can be constructed out of simpler primitives: PRGs and PRFs. However, in practice we use block ciphers that are designed "from scratch," and then use these block ciphers to construct simpler PRGs and PRFs when we need them.

We currently have **no proof** that any secure PRP exists. As we discussed in Section 5.2, such a proof would resolve the famous P vs NP problem. Without such proofs, what is our basis for confidence in the security of block ciphers being used today? The process that led to the Advanced Encryption Standard (AES) block cipher demonstrates the cryptographic community's best efforts at instilling such confidence.

The National Institute of Standards & Technology (NIST) sponsored a competition to design a block cipher to replace the DES standard from the 1970s. Many teams of cryptographers submitted their block cipher designs, all of which were then subject to years of intense public scrutiny by the cryptographic research community. The designs were evaluated on the basis of their performance and resistance to attacks against the PRP security definition (and other attacks). Some designs did offer proofs that they resist certain *classes of attacks*, and proofs that justify certain choices in building the block cipher from simpler components.

The Rijndael cipher, designed by Vincent Rijmen and Joan Daemen, was selected as the winner and became the AES standard in 2001. There may not be another cryptographic algorithm that has been the focus of more scrutiny and attempts at attack. So far no significant weaknesses in AES are known.[3]

The AES block cipher has a blocklength of 128 bits, and offers 3 different variants with 128-bit, 192-bit, and 256-bit keys. As a result of its standardization, AES is available in cryptographic libraries for any programming language. It is even implemented as hardware instructions in most modern processors, allowing millions of AES evaluations per second. As we have seen, once you have access to a good block cipher, it can be used directly also as a secure PRF (Corollary 6.8), and it can be used to construct a simple PRG (Construction 6.2). Even though AES itself is not a *provably secure* PRP, these constructions of PRFs and PRGs based on AES are secure. Or, more precisely, the PRF-security and PRG-security of these constructions is guaranteed to be as good as the PRP-security of AES.

⋆ 6.6 Strong Pseudorandom Permutations

Since a block cipher F has a corresponding inverse F^{-1}, it is natural to think of F and F^{-1} as interchangeable in some sense. However, the PRP security definition only guarantees a security property for F and not its inverse. In the exercises, you will see that it is possible to construct F which is a secure PRP, whose inverse F^{-1} is not a secure PRP!

It would be very natural to ask for a PRP whose F and F^{-1} are both secure. We will later see applications where this property would be convenient. An even stronger requirement would allow the distinguisher to query both F and F^{-1} in a *single* interaction (rather than one security definition where the distinguisher queries only F, and another definition where the distinguisher queries only F^{-1}). If a PRP is indistinguishable from a random permutation under that setting, then we say it is a **strong PRP** (SPRP).

In the formal security definition, we provide the calling program *two* subroutines: one for forward queries and one for reverse queries. In $\mathcal{L}_{\textsf{sprp-real}}$, these subroutines are implemented by calling the PRP or its inverse accordingly. In $\mathcal{L}_{\textsf{sprp-rand}}$, we emulate the behavior of a randomly chosen permutation that can be queried in both directions. We maintain two associative arrays T and T_{inv} to hold the truth tables of these permutations, and sample their values on-demand. The only restriction is that T and T_{inv} maintain consistency ($T[x] = y$ if and only if $T_{inv}[y] = x$). This also ensures that they always represent an invertible function. We use the same technique as before to ensure invertibility.

[3]In all fairness, there is a possibility that government agencies like NSA know of weaknesses in many cryptographic algorithms, but keep them secret. I know of a rather famous cryptographer (whom I will not name here) who believes this is likely, based on the fact that NSA has hired more math & cryptography PhDs than have gone on to do public research.

Definition 6.13 (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:*

```
$\mathcal{L}^F_{\text{sprp-real}}$
$k \leftarrow \{0,1\}^\lambda$

LOOKUP($x \in \{0,1\}^{blen}$):
    return $F(k,x)$

INVLOOKUP($y \in \{0,1\}^{blen}$):
    return $F^{-1}(k,y)$
```

```
$\mathcal{L}^F_{\text{sprp-rand}}$
$T, T_{inv}$ := empty assoc. arrays

LOOKUP($x \in \{0,1\}^{blen}$):
    if $T[x]$ undefined:
        $y \leftarrow \{0,1\}^{blen} \setminus T$.values
        $T[x] := y$;   $T_{inv}[y] := x$
    return $T[x]$

INVLOOKUP($y \in \{0,1\}^{blen}$):
    if $T_{inv}[y]$ undefined:
        $x \leftarrow \{0,1\}^{blen} \setminus T_{inv}$.values
        $T_{inv}[y] := x$;   $T[x] := y$
    return $T_{inv}[y]$
```

Earlier we showed that using a PRF as the round function in a 3-round Feistel cipher results in a secure PRP. However, that PRP is **not** a *strong PRP.* Even more surprisingly, adding an extra round to the Feistel cipher does make it a strong PRP! We present the following theorem without proof:

Theorem 6.14 (Luby-Rackoff) *If $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ is a secure PRF, then the 4-round Feistel cipher $\mathbb{F}_4$ (Construction 6.11) is a secure SPRP.*

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.

(a) Let $m \in \{0,1\}^{out}$ be a fixed (public, hard-coded, known to the adversary) string. Define:

$$F_m(k,x) = F(k,x) \oplus m.$$

Prove that for every m, F_m is a secure PRF.

(b) Define

$$F'(k,x) = F(k,x) \oplus x.$$

Prove that F' 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 5.9(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. In the previous problem, what happens when m is secret and part of the PRF seed? Let F be a secure PRF, and define the new function: Define the new function

$$F'\Big((k,m),x\Big) = F(k,x) \oplus F(k,m).$$

The seed of F' is (k,m), which you can think of as a $\lambda + in$ bit string. Show that F' is indeed a secure PRF.

Hint: Rewrite the F' algorithm to include an "if $x = m$" clause and argue that the calling program can rarely satisfy this clause.

6.7. 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.

6.8. Suppose F is a secure PRF with input length in, but we want to use it to construct a PRF with longer input length. Below are some approaches that **don't** work. For each one, describe a successful distinguishing attack and compute its advantage:

(a) $F'(k, x\|x') = F(k,x)\|F(k,x')$, where x and x' are each *in* bits long.

(b) $F'(k, x\|x') = F(k,x) \oplus F(k,x')$, where x and x' are each *in* bits long.

(c) $F'(k, x\|x') = F(k,x) \oplus F(k,x \oplus x')$, where x and x' are each *in* bits long.

(d) $F'(k, x\|x') = F(k,\mathtt{0}\|x) \oplus F(k,\mathtt{1}\|x')$, where x and x' are each $in - 1$ bits long.

6.9. 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:

$$F(k,x) = \bigoplus_{i : x_i = 1} k_i.$$

Show that F is **not** a secure PRF. Describe a distinguisher and compute its advantage.

6.10. 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.

6.11. A function $\{\mathtt{0},\mathtt{1}\}^n \to \{\mathtt{0},\mathtt{1}\}^n$ is chosen uniformly at random. What is the probability that the function is invertible?

6.12. Let F be a secure PRP with blocklength $blen = 128$. Then for each k, the function $F(k,\cdot)$ is a permutation on $\{\mathtt{0},\mathtt{1}\}^{128}$. Suppose I choose a permutation on $\{\mathtt{0},\mathtt{1}\}^{128}$ 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?

6.13. Suppose $R : \{\mathtt{0},\mathtt{1}\}^n \to \{\mathtt{0},\mathtt{1}\}^n$ is chosen uniformly among all such functions. What is the probability that there exists an $x \in \{\mathtt{0},\mathtt{1}\}^n$ such that $R(x) = x$?

Hint: First find the probability that $R(x) \neq x$ for all x. Simplify your answer using the approximation $(1-y) \approx e^{-y}$.

6.14. 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 6.7. 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*.

6.15. Let $F : \{\mathtt{0},\mathtt{1}\}^{in} \to \{\mathtt{0},\mathtt{1}\}^{out}$ be a (not necessarily invertible) function. We showed how to use F as a round function in the Feistel construction ony when $in = out$.

Describe a modification of the Feistel construction 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! You are not being asked to show any security properties of the Feistel construction.

6.16. Show that a 1-round keyed Feistel cipher **cannot** be a secure PRP, no matter what its round functions are. That is, construct a distinguisher that successfully distinguishes $\mathcal{L}^F_{\text{prp-real}}$ and $\mathcal{L}^F_{\text{prp-rand}}$, knowing only that F is a 1-round Feistel cipher. In particular, the purpose is to attack the Feistel transform and not its round function, so your attack should work no matter what the round function is.

6.17. Show that a 2-round keyed Feistel cipher **cannot** be a secure PRP, no matter what its round functions are. Your attack should work without knowing the round keys, and it should work even with different (independent) round keys.

Hint: A successful attack requires two queries.

6.18. 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.

6.19. 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:

$$\Pr_{y \leftarrow \{0,1\}^{blen}} \left[\mathcal{A}(y) \diamond \mathcal{L}^F_{\text{prf-real}} \text{ outputs } F^{-1}(k, y)\right] \text{ is non-negligible.}$$

The notation means that $\mathcal{A}$ receives a random block y as an input (and is also linked to $\mathcal{L}_{\text{prf-real}}$). 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?

6.20. Let F be a secure PRP with blocklength $blen = \lambda$, and consider $\widehat{F}(k, x) = F(k, k) \oplus F(k, x)$.

(a) Show that $\widehat{F}$ is not a strong PRP (even if F is).

★ (b) Show that $\widehat{F}$ is a secure (normal) PRP.