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 $s \in \{0,1\}^\lambda$ 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:

$$F(k,0\cdots00)\|F(k,0\cdots01)\|F(k,0\cdots10)\|\cdots\|F(k,1\cdots10)\|F(k,1\cdots11).$$

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 $\{0,1\}^{in}$ to $\{0,1\}^{out}$. 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 $\mathcal{L}_{\text{prf-rand}}$ as follows:

```
QUERY(x ∈ {0,1}^in):
  z ← {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) ⊕ 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 $\mathcal{L}^F_{\text{prf-}\star}$ 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 $\mathcal{L}_{\text{prf-}\star}$ 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 $x_1 \neq x_2$
$z_1 \leftarrow \text{QUERY}(x_1)$
$z_2 \leftarrow \text{QUERY}(x_2)$
return $z_1 \oplus z_2 \stackrel{?}{=} x_1 \oplus x_2$

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, $\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-rand}} \Rightarrow 1] = 1/2^{2\lambda}$.

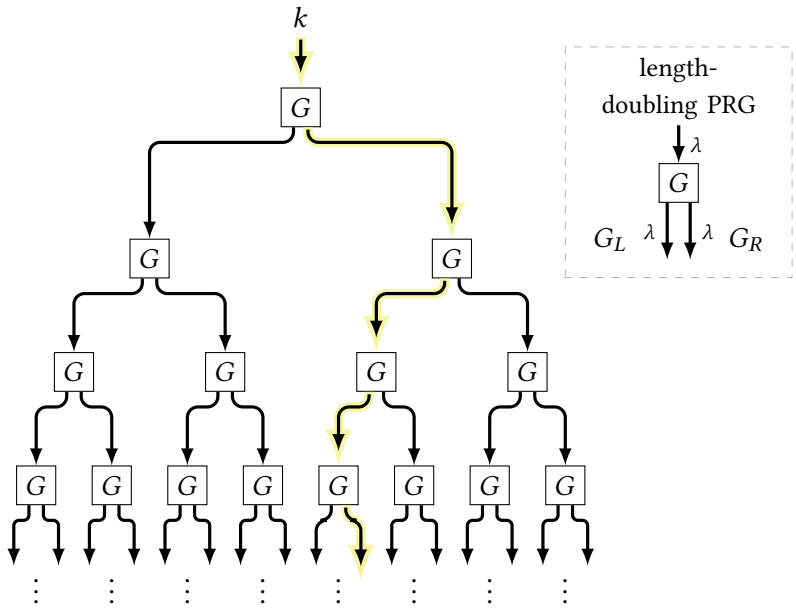
The advantage of $\mathcal{A}$ is therefore $|1 - 1/2^{2\lambda}|$ which is non-negligible (in fact, it is negligibly close to 1). This shows that F is not a secure PRF.

⋆ 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 PR**F**. 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)

$in = arbitrary$
$out = \lambda$

$F(k,x)$:
$v := k$
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 $x\mathtt{0}$ and $x\mathtt{1}$, 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 $T[x\mathtt{0}] = G_L(T[x])$ and $T[x\mathtt{1}] = G_R(T[x])$.

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 $T[\epsilon] = k$, $T[\mathtt{0}] = G_L(T[\epsilon])$, and $T[\mathtt{1}] = G_R(T[\epsilon])$. The security of the PRG G suggests that we can instead replace $T[\mathtt{0}]$ and $T[\mathtt{1}]$ 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 $T[\mathtt{0}]$ or $T[\mathtt{1}]$. 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 $T[v] \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$ 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:
    $T[p] \leftarrow \{\mathtt{0},\mathtt{1}\}^\lambda$
  $v := T[p]$
  for $i = D + 1$ to *in*:
    if $x_i = \mathtt{0}$ then $v := G_L(v)$
    if $x_i = \mathtt{1}$ 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:
$T[p] \leftarrow \{0,1\}^\lambda$	$k \leftarrow \{0,1\}^\lambda$
$v := T[p]$	$v := F(k,x)$
for $i =$ 0 + 1 to in:	
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:	// *unreachable*
if $x_i = 0$ then $v := G_L(v)$	
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:

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

```
T := empty assoc. array

QUERY(x):
  p := first D bits of x
  if T[p] undefined:
    T[p] ← {0,1}^λ
    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 $\mathcal{L}_{\text{hyb-1}}[d]$ and $\mathcal{L}_{\text{hyb-2}}[d]$ have identical behavior. The only difference is that whenever $T[p]$ is sampled at level d of the tree, $\mathcal{L}_{\text{hyb-2}}$ 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.

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

```
T := empty assoc. array

QUERY(x):
  p := first D bits of x
  if T[p] undefined:
    T[p‖0] ← {0,1}^λ
    T[p‖1] ← {0,1}^λ
  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 $\mathcal{L}_{\text{hyb-2}}[d] \approx \mathcal{L}_{\text{hyb-3}}[d]$. The difference between these libraries is in how $T[p\|0]$ and $T[p\|1]$ are computed. In $\mathcal{L}_{\text{hyb-3}}$, they are derived from applying a PRG to $T[p]$, a value that is never used elsewhere. In $\mathcal{L}_{\text{hyb-3}}$, 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 $\mathcal{L}_{\text{hyb-2}}$, we can apply the PRG security of G to obtain $\mathcal{L}_{\text{hyb-3}}$.

Finally, we claim that $\mathcal{L}_{\text{hyb-3}}[d] \equiv \mathcal{L}_{\text{hyb-1}}[d+1]$ 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 $\mathcal{L}_{\text{hyb-1}}$ library assigns these values lazily at the last moment, while $\mathcal{L}_{\text{hyb-3}}$ also assigns them whenever a "sibling" node is requested.

Putting all of these observations together, we have:

$$\begin{aligned}
\mathcal{L}^F_{\text{prf-real}} &\equiv \mathcal{L}_{\text{hyb-1}}[0] \equiv \mathcal{L}_{\text{hyb-2}}[0] \approx \mathcal{L}_{\text{hyb-3}}[0] \\
&\equiv \mathcal{L}_{\text{hyb-1}}[1] \equiv \mathcal{L}_{\text{hyb-2}}[1] \approx \mathcal{L}_{\text{hyb-3}}[1] \\
&\vdots \\
&\equiv \mathcal{L}_{\text{hyb-1}}[in-1] \equiv \mathcal{L}_{\text{hyb-2}}[in-1] \approx \mathcal{L}_{\text{hyb-3}}[in-1] \\
&\equiv \mathcal{L}_{\text{hyb-1}}[in] \equiv \mathcal{L}^F_{\text{prf-rand}}
\end{aligned}$$

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.