1 Pseudorandom Functions (PRFs)

Pseudorandom functions (PRFs) are (keyed) functions whose behavior looks like a truly random function to an outsider (provided the outsider does not know the key). Such functions are also known as *ciphers* (The two terms are equivalent and some sub-communities of cryptography prefer one term over the other.) and they are instrumental in building symmetric encryption schemes. We'll return to this topic in Lecture 5 and shortly sketch the use of PRFs in symmetric encryption schemes in Section 3 and Section 4. For now, let us focus on the following questions

(1) How do we define security of PRFs?

(2) How can we build PRFs?

(3) Can we base PRFs on PRGs?

(4) Can we build PRGs from PRFs?

Jumping ahead, we note that the answer to (3) and (4) is yes. But before going there, what is the difference between a PRG and a PRF?

1.1 Syntax

Let's start by having a look at their syntax [1]

Syntax of a PRG	Syntax of a (λ, λ)-PRF
$g : \{0,1\}^* \to \{0,1\}^*$	$f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$
$x \mapsto g(x)$	$(k, x) \mapsto f(k, x)$
with $\lvert g(x)\rvert = \lvert x\rvert + s(\lvert x\rvert)$	with $\lvert k\rvert = \lvert x\rvert = \lvert f(k, x)\rvert$

I.e., a PRG takes a *single input* and expands it, whereas a PRF takes a key k and can then return a pseudorandom value $f(k, x)$ for each x. In a way, we can think of $2^{|x|}$ many pseudorandom values of length $|x|$ and indexed by x. One can generate many pseudorandom values also by a PRG when iterating it, but one will not iterate the PRG more than polynomial times, so the exponential indexing space is quite useful. For example, we can sample a uniformly random x, and with high probability, it will be different from any uniformly random x which I have sampled before. This will be used in the construction of symmetric-encryption schemes. Of course, I can also avoid repeating the same x by keeping state, and some symmetric encryption schemes indeed use state rather than generate values x at random, as good randomness is sometimes hard to get by. So, all in all, we *might* be able to study cryptography without defining PRFs, but they turn out a quite a convenient (in fact ubiquitous) tool and are standardized by the National Institute of Standards and Technology (NIST) as an independent primitive `https://csrc.nist.gov/projects/block-cipher-techniques`. In fact even PRGs are often built from PRFs, essentially by evaluating the PRF

[1] In this lecture, so far, we used the term PRF, but in later lectures, we will refer to the PRFs in this lecture as (λ, λ)-PRFs, where the first λ refers to the input length and the second λ refers to the output size.

several times and concatenating the outputs (see Exercise Sheet 3 or `https://en.wikipedia.org/wiki/NIST_SP_800-90A#CTR_DRBG`. We'll see the converse direction, how to build a PRF from a PRG in Section 2 which, jumping ahead, essentially shows us how to use a PRG to obtain exponentially many pseudorandom values, indexed by some x.

1.2 Security

In the previous lectures, we defined security via *security experiments* where we gave the adversary one out of two inputs. But we cannot easily give a function to an adversary, since the input-output table of a function would be exponentially large. Thus, we give the adversary *oracle* access to the function, i.e., we allow the adversary to choose inputs to the function and receive the function outputs[2].

Oracles How do we specify oracles? An oracle is specified by some pseudocode. For the security of PRFs, we will use a EVAL oracle which allows the adversary black-box access to the function. We will write oracles in all-capitals and in a special font. Next, how do we specify that the adversary calls the oracle EVAL? If we want to say that

> Adversary $\mathcal{A}$ sends z to the EVAL oracle and assigns the result to variable y, we write $y \leftarrow \mathsf{EVAL}(z)$ in the pseudo-code of $\mathcal{A}$.

I.e., when $\mathsf{EVAL}(z)$ appears in the pseudo-code of an adversary, then the adversary itself is not executing the code of $\mathsf{EVAL}(z)$, but rather, the adversary sends z to the EVAL oracle, the EVAL oracle performs operations and returns some answer. I.e., the adversary *only* sees the answer (and assigns it to y), but not the actual computations.

Games Since an adversary might interact with several oracles (e.g., in the case of block ciphers/pseudorandom permutation, we might want to allow the adversary to ask in the forward and backward direction and give one oracle for each direction), we might collect several oracles into a *Game*. I.e., a *Game* $\mathtt{G}$ consists of a set of of *oracles*, denoted $[\to \mathtt{G}]$, which operate on a joint *state* and might use the *parameters* specified by the game. We use the convention that a game only remembers the values of the state variables and "forgets" the value of any variable not explicitly listed as state.

Notation Finally, we need notation to denote an adversary which interacts with a game, for example $\mathtt{Gprf}^0_f$, we denote this as $\mathcal{A} \overset{\mathsf{EVAL}}{\to} \mathtt{Gprf}^0_f$. Thinking back to the PRG experiments we had, the PRG experiment returned an output 0 and 1, and we had statements of the form $\Pr\left[1 = \mathsf{Exp}^{\mathsf{PRG}}_{g,\mathcal{A}}\right]$. Similarly, we now

[2]One might wonder why is the adversary allowed to *choose* the input to the function. This seems very strong. Indeed, this is typically an over-estimation of the adversary's capabilities. This might seem strange, but it is actually a good thing to allow the adversary to do a lot of things. If our function is secure in a setting with a strong adversary, then, even more, it will be secure in the setting with a weak adversary. Additionally, as we will see in the *before the lecture part* of Lecture 4, sometimes, the adversary can partially determine the input to a cipher. A good system is designed such it does not break down just because an adversary happens to have a way to control the inputs to a function.

consider statements of the form $\Pr\left[1 = \mathcal{A}^{\mathsf{EVAL}} \Rightarrow \mathtt{Gprf}_f^0\right]$, where, essentially, the adversary $\mathcal{A}$ is the main procedure. Writing it this way will be very convenient when we write more proofs, as we'll see in lecture 4.

PRF security We are now ready to define the security of a PRF as indistinguishability between a *real* game $\mathtt{Gprf}_f^0$ and an *ideal* game $\mathtt{Gprf}^1$. The real game contains the pseudorandom function f, keyed with a key k (as we don't have an experiment "setup" anymore, we now draw the key k whenever EVAL is called for the first time. Namely, **if** $k = \bot$ is a check whether k has already been set or not.). In turn, the ideal game $\mathtt{Gprf}^1$ contains a random function. What is a random function? A random function is a function which maps each input to a random output. Importantly, a PRF is deterministic, i.e. when called twice on *the same* input, it yields *the same* output. This is important, since else it would be easy to distinguish a random function from a PRF, since PRFs always return the same output when an input is repeated (since they are deterministic functions of the key and the input). Below, in the code of the EVAL oracle of $\mathtt{Gprf}^1$, this will be modeled by maintaining a table T from which we retrieve a value $T[x]$ if x has been queried before (again, **if** $T[x] = \bot$ checks whether $T[x]$ has already been set or not). Let us now turn to the formal definition.

Definition 1.1 (Security of a (λ, λ)-PRF)**.** A (λ, λ)-PRF is secure if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the advantage $\mathsf{Adv}_{f,\mathcal{A}}^{\mathtt{Gprf}_f^0, \mathtt{Gprf}^1}(\lambda)$ defined as

$$\mathsf{Adv}_{f,\mathcal{A}}^{\mathtt{Gprf}_f^0, \mathtt{Gprf}^1}(\lambda) := \left|\Pr\left[1 = \mathcal{A}^{\mathsf{EVAL}} \Rightarrow \mathtt{Gprf}_f^0\right] - \Pr\left[1 = \mathcal{A}^{\mathsf{EVAL}} \Rightarrow \mathtt{Gprf}^1\right]\right|$$

is negligible in the security parameter λ, where $\mathtt{Gprf}_f^0$ and $\mathtt{Gprf}^1$ are defined below.

Note that all programs obtain the security parameter λ as *implicit* input now, i.e., it is given to them as input, but we omit to write it for brevity.

$\mathtt{Gprf}_f^0$

Package Parameters	
λ:	security parameter
in:	input length λ or $*$
out:	output length λ
f:	function, (in, λ)-PRF

Package State	
k:	*key*

$\mathsf{EVAL}(x)$

assert $x \in \{0,1\}^{\text{in}}$
if $k = \bot$:
  $k \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow f(k, x)$
return y

$\mathtt{Gprf}^1$

Package Parameters	
λ:	security parameter
in:	input length λ or $*$
out:	output length λ

Package State	
T:	**table** [bitstring, integer $\rightarrow$ bitstring]

$\mathsf{EVAL}(x)$

assert $x \in \{0,1\}^{\text{in}}$
if $T[x] = \bot$
  $T[x] \leftarrow\!\!\$\ \{0,1\}^\lambda$
$y \leftarrow T[x]$
return y

1.3 Simple Example

In this section we give a simple example on how to use this newly introduced notation and definition. Namely, we prove that a constant function is not a PRF.

Theorem. The function $f(k, x) = 0^{|x|}$ is not a PRF.

Proof. Consider the following adversary

$\mathcal{A}()$

$y \leftarrow \mathsf{EVAL}(1^n)$
if $y = 0^n$
  return 1
else
  return 0

The above adversary is polynomial time, since EVAL query can be counted as one step and the rest of the computations that the adversary does are also efficient.
Now notice that in the ideal game EVAL will return a uniformly random bit-string, and the probability that a uniformly random string is 0^n is 2^{-n}, so in ideal game the adversary returns 1 with very small probability (probability 2^{-n}).
However, in the real game EVAL always computes the output of the PRF, so it always returns 0^n, so adversary always returns 1.
Hence, we can now compute the advantage:

$$\mathsf{Adv}^{\mathtt{Gprf}_f^0,\mathtt{Gprf}^1}_{f,\mathcal{A}}(\lambda) := \left|\Pr\left[1 = \mathcal{A}^{\overset{\mathsf{EVAL}}{\rightarrow}\ \mathtt{Gprf}_f^0}\right] - \Pr\left[1 = \mathcal{A}^{\overset{\mathsf{EVAL}}{\rightarrow}\ \mathtt{Gprf}^1}\right]\right| = 1 - 2^{-n}$$

which is non-negligible (very close to 1 even!), so the adversary succeeds in distinguishing f, so f is not a PRF. □

Note that in the above example the value 1^n is quite arbitrary, the adversary could also sample a random string as input to EVAL, or any string you prefer, and the analysis would not change (since the function f does not use the input, only the input length).

2 The Goldreich-Goldwasser-Micali (GGM) construction

The Goldreich-Goldwasser-Micali (GGM) construction constructs a pseudorandom function from a length-doubling pseudorandom generator as illustrated in Figure 1. The idea is to design a binary tree based on a length-doubling pseudorandom generator g by seeing (1) the key k of the PRF as an input to g and (2) then iterating the pseudorandom generator $|x|$ many times, where x is the second input to the PRF and, in the i-th iteration, we take the left output of g if $x_i = 0$ and the right input if $x_i = 1$. See the following example for $x = 10110$ on the left (the example is simplified by $|x| = |k| = \lambda$). The actual construction f_{GGM} is given on the right.

Simplified example for $x = 10110$

$f_{\text{GGM}}(k, x)$

$y_{[]} \leftarrow k$ // [] for empty string
$y_1 \leftarrow g_1(y_{[]})$
$y_{10} \leftarrow g_0(y_1)$
$y_{101} \leftarrow g_1(y_{10})$
$y_{1011} \leftarrow g_1(y_{101})$
$y_{10110} \leftarrow g_0(y_{1011})$
return y_{10110}

GGM Construction for arbitrary x

$f_{\text{GGM}}(k, x)$

$y_{[]} \leftarrow k$ // [] denotes empty string
for $i = 1..|x|$
  index $\leftarrow x_{1..i-1}$
  $y_{\text{index}||x_i} \leftarrow g_{x_i}(y_{\text{index}})$
return y_x

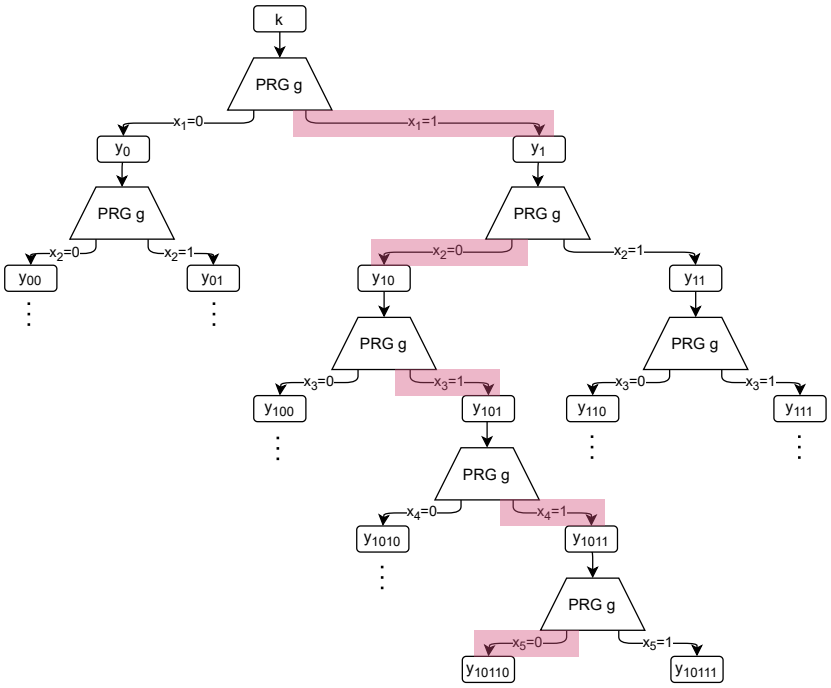

Figure 1: GGM Construction

Theorem (Goldreich-Goldwasser-Micali (GGM))**.** Let g be a pseudorandom generator (PRG) with $|g(x)| = 2|x|$. We write $g(x) = g_0(x)||g_1(x)$, where $|g_0(x)| = |g_1(x)| = |x|$, i.e., g_0 denotes the *left* half output of g and g_1 denotes the *right* half output of g. Then, f_{GGM} is a pseudorandom function.

3 Symmetric encryption from stream ciphers: counter mode

A pseudorandom function (also known as a *stream cipher* in some communities) can be easily evaluated in the *forward* direction, but it is not necessarily injective and does not necessarily admit an efficient inversion algorithm. E.g., f_{GGM} is most likely not an injective function.

How do we encrypt using a PRF? We use a PRF in *counter mode* to derive a padding and then xor the message and the padding to obtain the ciphertext. Below is the code for using a stream cipher f in counter mode. The notation $nonce + i$ refers to counting upwards in binary from $nonce$ by i modular 2^λ, i.e., the number after 1^λ is 0^λ.

CTR-f-$\mathsf{enc}(k, m)$	CTR-f-$\mathsf{dec}(k, (nonce, c))$
$nonce \leftarrow\!\!\$\ \{0,1\}^\lambda$	
$\ell \leftarrow \left\lceil \frac{\lvert m\rvert}{\lambda} \right\rceil$	$\ell \leftarrow \left\lceil \frac{\lvert c\rvert}{\lambda} \right\rceil$
for $i = 1..\ell$	**for** $i = 1..\ell$
$\quad pad_i \leftarrow f(k, nonce + i)$	$\quad pad_i \leftarrow f(k, nonce + i)$
$pad' \leftarrow pad_1 \lvert\rvert .. \lvert\rvert pad_\ell$	$pad' \leftarrow pad_1 \lvert\rvert .. \lvert\rvert pad_\ell$
$pad \leftarrow pad'_{1..\lvert m\rvert}$	$pad \leftarrow pad'_{1..\lvert c\rvert}$
$c \leftarrow m \oplus pad$	$m \leftarrow c \oplus pad$
return $(nonce, c)$	**return** m

4 Symmetric encryption from block ciphers

4.1 Pseudorandom permutations (PRPs)/block ciphers

A (λ, λ)-pseudorandom *permutation* f (known as a *block cipher* in some communities) has the same syntax as a (λ, λ)-PRF and satisfies the same security properties as a (λ, λ)-PRF (random functions and random permutations are hard to distinguish for polynomial-time adversaries). What distinguishes a (λ, λ)-pseudorandom *permutation* (PRP) from a (λ, λ)-PRF, is that a (λ, λ)-PRP f additionally admits a function f_{inv} such that for all $x, k \in \{0,1\}^\lambda$, it holds that $x = f_{\text{inv}}(k, f(k, x))$, i.e., first computing $y \leftarrow f(k, x)$ and then running $f_{\text{inv}}(k, y)$ will yield x again when using the same key k in forward and backward direction. AES, for example, is a PRP.
PRPs can be easily used as PRFs—one simply does not use f_{inv}. For the other direction, it is possible to turn a PRF into a PRP by the *Feistel construction*, first suggested by Luby and Rackoff, see `https://people.eecs.berkeley.edu/~luca/cs276/lecture15.pdf` for an introduction. There is an entire line of research discussing how many Feistel rounds are needed to ensure that the resulting permutation is indeed indistinguishable from random, see `https://eprint.iacr.org/2015/876` for a relatively recent work.

4.2 Cipher-block chaining (CBC) mode

In the lecture, we saw how to encrypt with a PRP. Namely, we used the *cipher-block chaining* (CBC) mode for encryption. The code is given below. We emphasize that CBC mode can *only* be used with a PRP, but not with a general PRF (unless the PRF is a PRP, that is). Below, for simplicity, we assume that the length of m is a multiple of block-length. If one wants to allow an arbitrary number of bytes, then one needs to use an injective padding, i.e., *each* message (even those which are a multiple of the block length) need to get at least one

byte of padding (since else, there are ambigious messages). The easiest way of padding to block size 128 bit (8 byte) is by adding whichever of the following makes the message a multiple of the block length:

- a 1-byte padding consisting of all zeroes, i.e., 00000000.
- a 2-byte padding consisting of twice an encoding of 1, i.e., 00000001 00000001.
- a 3-byte padding consisting of three times an encoding of 2, i.e., 00000010 00000010 00000010.
- a 4-byte padding consisting of 4 times an encoding of 3, i.e., 00000011 00000011 00000011 00000011.
- a 5-byte padding consisting of 5 times an encoding of 4, i.e., 00000100 00000100 00000100 00000100 00000100.
- a 6-byte padding consisting of 6 times an encoding of 5, i.e., 00000101 00000101 00000101 00000101 00000101 00000101.
- a 7-byte padding consisting of 7 times an encoding of 6, i.e., 00000110 00000110 00000110 00000110 00000110 00000110 00000110.
- a 8-byte padding consisting of 8 times an encoding of 7, i.e., 00000111 00000111 00000111 00000111 00000111 00000111 00000111 00000111.

Now, the following description of using a PRP in CBC-mode processes a message which has already been padded to the correct length.

CBC-f-$\mathsf{enc}(k, m)$

$nonce \leftarrow\!\!\$\ \{0,1\}^\lambda$
$c_0 \leftarrow nonce$
$\ell \leftarrow \dfrac{|m|}{\lambda}$
for $i = 1..\ell$
  $x_i \leftarrow m_{(i-1)\lambda+1..i\lambda}$
  $c_i \leftarrow f(k, x_i \oplus c_{i-1})$
$c \leftarrow (c_0, .., c_\ell)$
return c

CBC-f-$\mathsf{dec}(k, c)$

$\ell \leftarrow \dfrac{|c|}{\lambda} - 1$
Parse $c_0, .., c_\ell \leftarrow c$
for $i = 1..\ell$
  $x_i \leftarrow c_{i-1} \oplus f_{\text{inv}}(k, c_i)$
$m \leftarrow x_1||..||x_\ell$
return m