Definition 1.1 *A pseudo-random function (PRF) F is a deterministic algorithm that has two inputs: a key k and an input data block x. Its output $y := F(k, x)$ is called an output data block. The associated finite spaces are: the key space $\mathcal{K}$, the input space $\mathcal{X}$, and the output space $\mathcal{Y}$. We say that F is defined over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$. [1]*

If we could unconditionally prove that a PRG is secure, then this implies that P $\neq$ NP. If P $\neq$ NP then there are tasks that are difficult to do with a poly-time algorithm. With a short witness (poly-size advice), then we can solve NP problems. In particular, if you have the seed, then the PRG is completely deterministic and you've solved P = NP.

1.0.1 How do we compare hardness in symmetric cryptography?

We often prove that a PRG is secure under the assumption that P $\neq$ NP. This is the minimal assumption. If a PRG exists and is secure, then P cannot be equal to NP. If this assumption is not enough, we have to make even stronger hardness assumptions (we have to assume that problems are hard not only in the worst case, but also in the average case; in other words, "**average-case hardness**").

Another standard assumption in symmetric cryptography is that one-way functions exist and are difficult to invert in the average case (this is a stronger assumption than P $\neq$ NP).

Definition 1.2 *One-way function. A function $f : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ is one-way if for all efficient adversaries $\mathcal{A}$:*

$$\Pr[x \leftarrow \mathcal{X}_\lambda : \mathcal{A}(1^\lambda, f(x)) \in f^{-1}(x)]$$

Desired properties:

1. Efficiently computable
2. Hard to invert (on average). Every probabilistic poly-time algorithm can only invert the function with some negligible probability.

As there are multiple possible values that can map to $f(x)$, the notation indicates that the adversary only needs to output *one* of them (i.e., any value in the pre-image of $f(x)$). The 1^λ indicates that the adversary's input is at least as long as the length of the security parameter so taht it can run in at least time that is polynomial in the security parameter.

Definition 1.3 *Pseudorandom function (PRF). A PRF $f : \mathcal{K}_\lambda \times \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ is secure if for all efficient adversaries $\mathcal{A}$:*

$$\mathsf{PRFAdv}[\mathcal{A}, F] = |\Pr[W_0] - \Pr[W_1| < \mathit{negl}(\lambda)$$

For a **security game** wherein $b \in \{0, 1\}$, W_b is the output of $\mathcal{A}$ in this experiment:

1. The challenger chooses some value for b and selects k, f and computes y based on the chosen value for b.

2. The adversary can send queries $x_i \in \mathcal{X}$ to which the challenger responds with y_i. The adversary may respond adaptively to the values returned by the challenger.

3. The adversary outputs W_b (or $\hat{b}$) based on information gained in the game.

Definition 1.4 *Pseudorandom permutation (PRP). A pseudorandom permutation is the same as a PRF, but $F(k, \cdot)$ is a permutation (a bijection).*

In practice, we can consider a PRP as a block cipher; we directly assume that AES is a secure PRP.

1.0.2 Connections between basic primitives

Transformations between each of these primitives are reductions of the P = NP problems.

$$\text{OWFs} \leftrightarrow \text{PRGs} \leftrightarrow \text{PRFs} \leftrightarrow \text{PRPs}$$

The takeaway is that one-way functions are necessary and sufficient for symmetric cryptography. This is why we view OWFs as the axioms of everthing in symmetric cryptography. Because all assumptions are equivalent, we work with the simplest ones (OWFs).

Recall: A central tool in symmetric cryptography is **authenticated encryption**. In CS255, we've seen ways to construct a secure authenticated encryption scheme. In particular, we've seen that we can get secure authenticated encryption schemes with simpler primitives like PRFs and PRPs.

Unfortunately, we don't really know how to construct symmetric cryptographic primitives that are *unconditionally* secure. An unconditional proof of security of such primitives would solve P versus NP.

We can still get some provable security despite this limitation by reducing security of our primitives. We reduce it to simpler and better-understood problems. We can construct provably secure PRFs and PRPs from simpler primitives, namely OWFs.

Today we still use heuristic constructions like AES and HMAC to construct authentication systems since our transformations from OWFs are not yet as efficient as we'd like.

2.0.3 Example candidates for OWFs

We call these candidates since we cannot prove them to be OWFs.

Number theoretic candidates:

- Factoring. $f(x, y) = x \cdot y$ for equal length primes x, y
- Discrete-log. $f_{p,g}(x) = g^x \mod p$ (d-log is a permutation on the numbers 1 to $p-1$)

Combinatorial candidates:

- based on the subset-sum. $f(x_1, \ldots, x_n, S) := (\sum_{i \in S} x_i, \ldots, x_n)$
- Levin's OWF. f_L, $\exists$ OWF $\Rightarrow f_L$ is one-way

Note: Dlog is a permutation on the numbers 1 to $p-1$. If g is a generator and the group is cyclic then when you raise it to the power of x, you get all the possible x's from 1 to $p-1$. This is the permutation on these numbers.

Definition 2.5 *One way permutation. A function f is a one-way permutation if it satisfies the following properties:*

1. *f is a one way function (OWF),*
2. *It is length-preserving ($|f(x)| = |x|$) and one-to-one (injective).*

Why do we call it a one way permutation? For every particular input length, the function f is just a permutation on all possible strings of that length.

Recall: Our overall goal is to be able to construct all symmetric primitives from the mild assumption that OWFs exist.

The general path that one takes to construct symmetric key primitives:

OWF $\rightarrow$ weak-PRG (or a stretch-1 PRG) $\rightarrow$ poly-stretch PRG (of arbitrary polynomial length) $\rightarrow$ PRF.

2.1 From OWPs to stretch-1 PRGs

Given some seed drawn uniformly at random, s,

Why can't we just take s_1? You can imagine that $f(s)$ could be one-way but leak the first bit of s. Then, the first bit of s will not look random given $f(s)$. Any fixed-bit isn't really hidden by $f(s)$. To claim that $f(s)$ is uniformly random we do not need the one-way property, we only need the permutation. To explain this, let's look at $f(s)$ when $s :\leftarrow^R \{0,1\}^n$.

What's the probability that $f(s)$ is equal to some constant string a? Formally, what is $\Pr[f(s) = a : s \leftarrow^R \{0,1\}^n]$? We claim that this is the same as the probability that s is equal to f inverse of a (when s is chosen at random from $\{0,1\}^n$), $\Pr[s = f^{-1}(a) : s \leftarrow^R \{0,1\}^n]$. Because f is a permutation we can always find an inverse for a and this is the only case in which $f(s) = a$.

$$\Pr[f(s) = a : s \leftarrow^R \{0,1\}^n] = \Pr[s = f^{-1}(a) : s \leftarrow^R \{0,1\}^n]$$

Now, we can use the fact that s is a uniformly random variable so that the probability that s is equal to some fixed value $f^{-1}(a)$ is $1/2^n$ as s is chosen uniformly at random. Thus, since s is uniformly random, then $f(s)$ is uniformly random. Intuitively, if I take all possoble strings of length n with uniform probability and permute them around, I still get the uniform probability distribution.

Note: $f(s)$ doesn't look random if I also see s. That is, if one sees them together. If we only see $f(s)$, then s could be anything since we permuted the values around.

Definition 2.6 *Hardcore bit.* $b : \{0,1\}^* \to \{0,1\}$ *is a harcore bit of a function* $f : \{0,1\}^* \to \{0,1\}^*$ *if:*

1. $b(x)$ *is efficiently computable from* x,
2. $b(x)$ *is hard to predict from* $f(x)$

Formally, $\forall$ PPT alg. $\mathcal{A}$:

$$\Pr[\mathcal{A}(1^n, f(x)) = b(x) : x \leftarrow^R \{0,1\}^n] \leq \frac{1}{2} + \text{negl.} \tag{2.1}$$

A hardcore bit is defined for a specific function. The hardcore bit is a predicate of the functions in some sense. In the security game of a hardcore bit, it's hard to compute $b(x)$ given $f(x)$ as input to the algorithm. The definition of every hardcore bit is coupled to a particular function. Intuitively, $b(x)$ is partial information about the input that the OWF hides.

In words, it is hard for A to predict the value of $b(x)$ better than random, so we upper-bound it at negligibly better than $1/2$ (a random guess for a single bit in $\{0,1\}$ is $1/2$, so there is a "naive algorithm" that always succeeds with probability $1/2$).

Theorem 2.1 *Goldreich-Levin. Every function has a hardcore bit.*

Idea: If we have a hardcore bit, then the PRG $G(s)$ is defined as $g(s) = f(s)\|b(s)$. Up until now the condition on $b(s)$ is that it's hard to predict the value of $b(s)$ from $f(s)$. We need something stronger. We need to say that $b(s)$ actually looks random (so that $g(s)$ looks random. Thankfully, if $b(s)$ is hard to predict, it follows that it has to be random looking (see official lecture notes for more).

A random linear combination of the bits of the input is hard-core. Given OWP f,

- We extend it to get another OWP $g(x\|r) := (f(x)\|r)$ where $|x| = |r| = n$
- $b(x\|r) = \sum_{i=1}^{n} x_i \cdot r_i \mod 2$

If f is one-way, then g is one-way. It's easy to guess the bits of r given the output (since they're copied as is) but from $f(x)$ it should be hard to guess x (so g is one-way).

We claim that b is hardcore for g. Given $g(x\|r)$ it should be hard to find the inner product of x_i and r_i . Even if $\mathcal{A}$ leaks partial information about x, the random linear combination of the bits of x should still be hard to predict. HW1 Q3 focuses on this topic.

$$f \Rightarrow g : G(x\|r := f(x)\|r\|b(x,r) \tag{2.2}$$

So we took as input a string of length $2n$ bits and output a string of length $2n+1$. b is not only hard to predict but it also looks random.

2.2 The Blum-Micali Construction

How do we increase the stretch of our PRG? The Blum-Micali construction. You can start with a 1-stretch PRG and then use G' to get a PRG that increases the length of the input.

We start with a weak PRG G that only increases the length of the input by 1.

$$\text{Let } G : \{0,1\}^n \to \{0,1\}^{n+1} \text{ be a PRG}$$

We want to construct a different PRG G' that will be much better. Takes n bits as input and outputs $\ell(n)$ bits where $\ell(n)$ is poly in n.

$$\text{Construct } G' : \{0,1\}^n \to \{0,1\}^{\ell(n)}$$

To construct a larger output, we repeatedly apply G. Since the output of G is $n+1$ bits, we truncate the output $G(s)$ to be n bits so that we can continue to feed subsequent outputs to G. $G(s_0) = s_1\|b_1$. We repeatedly apply G $\ell(n)$ times. Each $n+1$th bit is saved $\{b_1, \ldots, b_\ell\}$. We can discard s_ℓ since it's not used as input for another computation of G.

Now we must prove that this is secure.

Theorem 2.2 *G is a secure PRG $\Rightarrow$ G' is a secure PRG.*

Claim 2.3 *G' is efficient (running time).*

The running time of G' is: $t_{G'}(n) = \ell(n) \times t_G(n) + O(\ell(n))$. Since ℓ is poly and G' applies the basic PRG G a linear number of times, it's quite efficient.

Claim 2.4 *G' is secure.*

$$\left\{G'(s) : s \leftarrow^R \{0,1\}^n\right\} \approx_c \left\{y : y \leftarrow^R \{0,1\}^n\right\} \tag{2.3}$$

That is, $\forall PPT$ algorithm $\mathcal{A}$

PRG distinguishing advantage of adversary $\mathcal{A}$ on PRG G':

$$\mathsf{PRGAdv}[\mathcal{A}, G'] = |\Pr[\mathcal{A}(G'(s)) = 1 : s \leftarrow^R \{0,1\}^n] - \Pr[\mathcal{A}(y) = 1 : y \leftarrow^R \{0,1\}^{\ell(n)}]| \leq \text{negl.}(n) \tag{2.4}$$

We will prove that these two distributions D_0 and D_1 using a **hybrid argument**.

2.3 Hybrid Argument

D_0 is the pseudorandom distribution from the Blum-Micali construction.

Instead of running the PRG for *ell* steps, we'll just run it for $\ell - 1$ steps. For example, for D_1 we feed s into G so that the input that produces b_2 (recall that s will always be a truly random input). D_i has i truly random bits and $\ell - i$ bits obtained from the PRG.

For example, D_0 has 0 truly random output bits and all ℓ bits $\{b_1, \ldots, b_\ell\}$ are obtained from the PRG. D_1 has 1 truly random bit, D_2 has 2 truly random bits, and D_ℓ has ℓ truly random bits (we never run G in this case).

Claim 2.5 *The indistinguishability of every neighboring pair D_i, D_{i+1} follows from that the basic PRG G is secure.*

Fix $\mathcal{A}$ to be some particular alg that tries to break G'. Let $p_i = \Pr[\mathcal{A}(y) = 1 : y \leftarrow D_i]$. We want to show that $|p_0 - p_\ell| = \text{negl}$.

Telescopic Sequence and Triangle Inequality

$$|p_0 - p_\ell| = |p_0 - p_1 + p_1 - p_2 + p_2 \ldots p_{\ell-1} - p_\ell| \leq \sum_{i=1}^{\ell} |p_{i+1} - p_i| \tag{2.5}$$

Claim 2.6 $\forall i = 1, \ldots, \ell, |p_{i+1} - p_i| \leq \mathit{negl}(n)$.

The claim implies the theorem because if it is true that $|p_{i+1} - p_i| \leq \text{negl}(n)$ for all i, then $|p_0 - p_\ell|$ is equivalent to:

$$\ell(n) \times \text{negl}(n) = \text{negl}(n) \tag{2.6}$$

The $\ell(n)$ term comes from the number of terms in the sum (from $i = 1$ to ℓ). Thus, $|p_0 - p_\ell| \leq \ell(n) \times \text{negl}(n) = \text{negl}(n)$. The $\ell(n)$ term is poly in n by construction, so multiplying poly $\times$ negl is still negl.

Proof. Suppose we have an algorithm $\mathcal{A}$ that tries to break G' $|p_{i+1} - p_i| = \varepsilon(n)$. We construct $\mathcal{B}$ that is a distinguisher for the basic PRG G.

Claim 2.7 *The success probability of $\mathcal{B}$ is related to ε. Since G is a secure PRG, this implies that $\mathcal{B}$ cannot break it with non-negl probability.*

If $\mathcal{B}$'s goal is to break G, the distinguisher $\mathcal{B}$'s input is a string that is the same length as the output of the PRG, $n+1$. Let this length $n+1$ string be called z. $\mathcal{B}$ works as follows:

On input $z \in \{0,1\}^{n+1}$:

1. Parse z as $z = s_i \| b_i$ where $b_i \in \{0,1\}$, $s_i \in \{0,1\}^n$
2. Choose $b_1, \ldots, b_{i-1} \leftarrow^R \{0,1\}$
   (to generate the first $i-1$ bits, just draw them uniformly at random $b_1 \leftarrow^R \{0,1\}, \ldots, b_{i-1} \leftarrow^R \{0,1\}$).
3. Compute $b_{i+1}, s_{i+1} \leftarrow G(s_i), \ldots b_\ell, s_\ell \leftarrow G(s_{\ell-1})$
4. Set $y \leftarrow b_1, \ldots, b_\ell$
5. Run and output $\mathcal{A}(y)$

We have two cases:

1. $z = G(s)$ for $s \leftarrow^R \{0,1\}^n$. The output of $\mathcal{B}$ looks like $b_1 \leftarrow^R \{0,1\} \ldots b_{i-1} \leftarrow^R \{0,1\}$ and use G to get the remaining $\ell - (i-1)$ bits. This is exactly D_{i-1}.
2. $z \leftarrow^R \{0,1\}^{n+1}$. The output of $\mathcal{B}$ looks like $b_1 \leftarrow^R \{0,1\} \ldots b_i \leftarrow^R \{0,1\}$. We then take $b_i \leftarrow^R \{0,1\}$, $s_i \leftarrow^R \{0,1\}^n$ and run G on these so that the remaining construction (running G on s_i and b_i and so on) gets us b_{i+1} through b_ℓ. This is exactly D_i.

Once it generates these distributions, it runs $\mathcal{A}$ on $y \leftarrow b_1, \ldots, b_\ell$.

$$\begin{aligned}
\mathsf{PRGADv}[\mathcal{B}, G] &= |\Pr[\mathcal{B}(G(s)) = 1 : s \leftarrow^R \{0,1\}^n] - \Pr[\mathcal{B}(z) : z \leftarrow^R \{0,1\}^{n+1}]| && (2.7)\\
&= |\Pr[\mathcal{A}(y) = 1 : y \leftarrow D_{i-1}] - \Pr[\mathcal{A}(y) = 1 : y \leftarrow D_i]| && (2.8)\\
&= |p_{i-1} - p_i| && (2.9)\\
&= \varepsilon(n) && (2.10)
\end{aligned}$$

Since G is a secure PRG, $\varepsilon(n)$ must be negl. ■

Note: This is a common proof construction in cryptography: You take one algorithm $\mathcal{A}$ that breaks that primitive that you *want* to prove secure and you use it to construct $\mathcal{B}$ that breaks the primitive that you *assume* to be secure.

To recap, we haev thus transformed OWP → hardcore bit → 1-bit stretch → poly-stretch PRG.

Now, we'd like to go from poly-stretch PRG to PRF.

2.3.1 Goldreich-Goldwasser-Micali (GGM) Construction

Recall: PRF takes two parameters, a key and an input $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^\lambda$

Q. Doesn't the length of the output need to be length n? Actually, we can arbitrarily increase the output length of the PRF from λ to n (the output length doesn't matter very much). For example, we can run the output of length λ through a PRG and get n output bits.

Given a PRG $G : \{0,1\}^{\lambda} \to \{0,1\}^{2\lambda}$. How do we use a length-doubling PRG to construct a PRF?

Take as input our seed s (s_0) and run it through G to get 2λ bits (s_1). We can interpret these 2λ bits as a function that takes a seed of length λ and an input of length 1-bit. $s_0 = F(s, 0)$ and $s_1 = F(s, 1)$. To increase the input size of the PRF we repeat the process and feed $F(s, 0)$ into G and feed $F(s, 1)$ into G to get s_{00}, s_{01}, s_{10}, and s_{11}. After $\log n$ steps in depth, we can construct a PRF that will take n bits as input. We don't need to explicitly construct the entire tree to compute the value of the PRF.

Where does the hybrid argument come in? In order to apply the PRG again for the security proof we need for each output (and intermediate input) to be random.

Recall: GGM construction is a way that goes from a PRG G that doubles its input ($\{0,1\}^\lambda \to \{0,1\}^{2\lambda}$) to a PRF $F : \{0,1\}^\lambda \times \{0,1\}^M \to \{0,1\}^\lambda$.

Idea: You start with some λ bit input, apply the PRG and get 2λ and iteratively build a tree n levels deep to get 2^n leaves. If you want to have a PRF that has an n bit input; you can pick any of these leaves as output of the PRF.

Why is the hybrid argument necessary? Each leaf is going to look random because of the preceding layer. We need a hybrid argument because the outputs are not random but pseudorandom. Input λ may be random but each output of G is pseudorandom. In order to apply the PRG again for the security proof we need for each output (and intermediate input) to be random, we use the hybrid argument to swap each intermediate input with something that is truly random. We then apply the PRG again, and so on all the way down.

3.4 Commitments

Commitments (e.g., how to play rock, paper, scissors over the phone)

Definition 3.7 *Commitment scheme. A commitment scheme allows you to commit to a message without revealing it (like a locked box).*

$$\mathsf{commit} : \mathcal{M} \times \mathcal{R} \to C \qquad \mathsf{commit}(m, r) \to c \tag{3.11}$$

We "open" a commitment by sending m, r. Commitments should have two properties:

1. Hiding: Seeing c says nothing about m. The notion for hiding is that for $\forall m_0, m_1 \in \mathcal{M}$

$$\forall m_0, m_1 \in \mathcal{M} \left\{\mathsf{commit}(m_0, r) : r \leftarrow^R \mathcal{R}\right\} \approx \left\{\mathsf{commit}(m_1, r) : r \leftarrow^R \mathcal{R}\right\}, \tag{3.12}$$

2. Binding: After seeing c, we cannot change our mind about m. In other words, no PPT adversary $\mathcal{A}$ can produce $m_0, m_1 \in \mathcal{M}, m_0 \neq m_1, r_0, r_1 \in \mathcal{R}$ s.t. $\mathsf{commit}(m_0, r_0) = \mathsf{commit}(m_1, r_1)$.

Q. Is $\mathsf{commit}(m, r) = \mathsf{AES}(r, m)$ a commitment? No. AES behaves like a PRF and it is hiding (if you take m and r is unknown to the person who gets the commitment it should be difficult to find m). AES is not necessarily binding as you could pick a different r' and open it up to a different message.

3.4.1 Pederson Commitments

- Public params: group G of prime order p where $g, h \in G$.
- Commitment: $\mathsf{commit}(m, r) = g^m h^r$ where $m, r \in \mathbb{Z}_p$

We want to ask whether it is hiding and biding:

1. Is it hiding? Yes. $\mathsf{commit}(m, r)$ is uniform in G. h^r is a random element of the group and masks the message that you're trying to commit to.
2. Is it binding? Yes, assuming the hardness of dlog in G.

Recall: Discrete log assumption from CS255. The discrete log problem states that given $h \in G$, it is hard to find x s.t. $h = g^x$.

We can play a brief security game to show this.

Discrete log security game. Take an adversary $\mathcal{A}$ and a challenger. The challenger picks some random x and raises g^x to get h, sends (g, h) to $\mathcal{A}$. Then the adversary attempts to find x. $\mathcal{A}$ sends $\hat{x}$ back to the challenger and wins if $\hat{x} = x$ and it correctly retrieves x. The dlog assumption is that there's no PPT $\mathcal{A}$ that can win this security game and retrieve x with greater than negl probability.

3.4.2 Life advice regarding commitments on breaking dlog

Whenever you want to break dlog, you should try to get 2 representations of the same group element. That is, if our goal is to take h and find the discrete log, find two different ways of finding h and solve the discrete log.

Suppose $g^{m_0} h^{r_0} = c = g^{m_1} h^{r_1}$:

$$c = g^{m_0} h^{r_0} \tag{3.13}$$
$$g^{m_0} h^{r_0} = g^{m_1} h^{r_1} \tag{3.14}$$
$$g^{m_0} (g^x)^{r_0} = g^{m_1} (g^x)^{r_1} \tag{3.15}$$

And thus,

$$m_0 + x r_0 = m_1 + x r_1 \tag{3.16}$$

$$x = \frac{m_1 - m_0}{r_0 - r_1} \tag{3.17}$$

In words, if you can find two different commitments $(m_0, r_0), (m_1, r_1)$ that commit to the same commitment c (two different ways of writing the same group element), you can break the discrete log assumption. Note, $r_0 \neq r_1$ otherwise it must be true that $m_0 = m_1$ and the LHS and RHS are the same representation (the premise of this advice is that they are different representations).

3.4.3 Proving that Pederson Commitments are Binding

We want to show that if dlog problem is hard in $\mathbb{G} \to$ Pederson commitment is binding. Suppose Adv $\mathcal{A}$ breaks binding. We thus build an adversary $\mathcal{B}$ that uses $\mathcal{A}$ and wins the dlog game.

The intuition is that if you have an adversary that can break Pederson commitment binding, then you can build something that wins the dlog game. We know that the dlog game is hard to win, so surely we cannot have an adversary that breaks the binding property of the Pederson commitment.

Proof. Given $\mathcal{A}$, build $\mathcal{B}$ that wins the dlog game against the dlog challenger.

The dlog game. Dlog challenger samples $x \leftarrow^R \mathbb{Z}_q, h \leftarrow g^x$ and sends g, h to $\mathcal{B}$. $\mathcal{B}$ waits for input from $\mathcal{A}$ to do something with it before sending $\hat{x}$ back to the dlog challenger.

The Pedereson binding game. $\mathcal{B}$ sends g, h to $\mathcal{A}$. $\mathcal{A}$ sends (m_0, m_1, r_0, r_1) s.t. $m_0 \neq m_1$ and $\mathsf{commit}(m_0, r_0) \neq \mathsf{commit}(m_1, r_1)$ back to $\mathcal{B}$.

$\mathcal{B}$ passes on g, h as parameters to the Pederson commitment. Since $\mathcal{A}$ can win the Pederson commitment game and get two different representations of the same commitment c that it needs to apply the life advice from earlier, this will allow us to retrieve x.

Whenever $\mathcal{A}$ wins the binding game such that it breaks the binding of the Pederson commitment, $\mathcal{B}$ can produce x that is the discrete log of h. $\mathcal{B}$ wins the discrete log game with the same probability that $\mathcal{A}$ breaks the binding property of Pederson commitments. ■

Q. Isn't it an issue that g, h are fixed as public parameters? In our game, $\mathcal{B}$ chooses the parameters. Is this an issue? No, because the binding property is supposed to bind the person who produces the commitment (in this case, $\mathcal{A}$). There are other times when this can be a problem, but we'll punt that for later. Here, it suffices to say that because the hiding property is unconditional, it's okay for the person who's going to be receiving the commitments to pick the public parameters.

For the person sending the commitments, they could break hiding, so you have the person receiving the commitment se the parameters. For the person receiving the commitments, they cannot can't break the hiding property by picking bad public parameters.

Why are Pederson Commitments neat? They are additively homomorphic (i.e., you can add them up).

3.4.4 Additive Homomorphism

We claim that the following relationship holds: $\mathsf{commit}(m_0, r_0) \cdot \mathsf{commit}(m_1, r_1) = \mathsf{commit}(m_0 + m_1, r_0 + r_1)$

$$\begin{aligned} \mathsf{commit}(m_0, r_0) \cdot \mathsf{commit}(m_1, r_1) &= g^{m_0} h^{r_0} \cdot g^{m_1} h^{r_1} && (3.18)\\ &= g^{m_0 + m_1} h^{r_0 + r_1} && (3.19)\\ &= \mathsf{commit}(m_0 + m_1, r_0 + r_1) && (3.20) \end{aligned}$$

In words, you can do some computation on a value you know nothing about. By adding up two commitments, you get a new commitment that is a commitment to the sum of the constituent messages. Commitment schemes aren't the only scheme that has this property, there are encryption schemes that have it too (later in the course, we will cover fully homomorphic encryption).

We don't need the homomorphism for rock, paper, scissors over the phone. Is there a simpler commitment scheme? Yes.

3.4.5 Random Oracles (for the uninitiated)

The *random oracle model* is a way to model hash functions. The random oracle model can be thought of as a stronger assumption about hash functions. The result is that we can think of hash functions as *random functions*:

$$H : \mathcal{X} \to \mathcal{Y} \text{ s.t. } H(x) \leftarrow^R \mathcal{Y} \tag{3.21}$$

In words, what this implies is that when we look at a hash function the output looks truly random. This is a much stronger assumption than collision resistance and in fact immediately implies collision resistance. This

is used all over the place in cryptography. Pretty much any signature scheme used in practice is based on the ROM. Examples:

- RSA full domain hash (we saw some of this in CS255)
- Most CCA secure public key encryption schemes

ex. Super easy commitments with hash functions in the ROM,

$$\mathsf{commit}(m, r) = H(m, r) \tag{3.22}$$

This is super easy because we don't have to think about groups or exponentiation.

1. Is it hiding? Yes. If you don't evaluate H on (m, r) then $H(m, r)$ looks uniformly random. If r is big enough, an adversary can guess r with $<$ negl probability. Thus, $H(m, r)$ is a PRF in the ROM too.
2. Is it binding? Yes. Breaking binding means m_0, m_1, r_0, r_1 s.t. $H(m_0, r_0) = H(m_1, r_1)$. H is a random function (that is collision resistant) by construction.

Q. Is $H(m)$ a commitment? No. It's not hiding. If you have a guess on m you can just check $H(m)$.

Q. Can we use hash functions as PRFs? Yes, but not really. AES is much faster than the ROM. There are other theoretical issues, but these are not the main reason.

It takes up a lot of space to write down a truly random function as every output of the function is unrelated to every other output (no structure). If H were a truly random function, its description would be very large. Hash functions in practice can be written down very concisely, so these are clearly not random functions.

Q. Why is it called a model when everything else we've talked about has been called an assumption?

3.4.6 Random Oracles (for the initiated)

Why do cryptography proofs say things about real systems even if we miss things like side-channel attacks. We can only build a model that we think reflects reality. We can prove statements within a model; the model acts as a framework for assumptions and proofs. Ultimately, we have to decide on our own whether our model matches reality. There's nothin tthat says that our proofs are connected to reality.

All proofs in previous lectures follow the *standard model*. The ROM augments this by adding an *oracle* that answers queries with evaluations of H. Everyone has access to the oracle.

The oracle allows us to sidestep the first stated problem of the ROM that truly random functions might have very large descriptions. Instead of having to worry about how to conretely write down the full hash function, we assume that an oracle exists and that we can ask the oracle certain questions.

We don't think about how the oracle implements the hash function. We also assume that the oracle generates answers to H and sends them back ot us.

As a *heuristic*, we replace the random oracle (RO) with a suitable* hash function. The random oracle has to be determinsitic (if we query it at the same point twice, we must get the same answer).

Q. What are suitable hash functions? Later we will discuss how to suitably discuss instantiation. SHA-256 is not a suitable hash function.

Q. Is the random oracle assumed to run in polynomial time? Each query to the ROM is considered to be constant time; it runs in unit time.

How do we construct proofs in the ROM? We will demonstrate this with an amazing PRF.j

3.4.7 An Amazing PRF

$$f(k, x) = H(x)^k \tag{3.23}$$

We will prove PRF security in ROM assuming the hardness of Decisional Diffie-Hellman (DDH).

Recall: DDH assumption. For a cyclic group G of order q with generator g,

$$\left\{g, g^x, g^y, g^{xy} : x, y \leftarrow^R \mathbb{Z}_q\right\} \approx_C \left\{g, g^x, g^y, g^z : x, y, z \leftarrow^R \mathbb{Z}_q\right\} \tag{3.24}$$

We want to prove that if there's an adversary $\mathcal{A}$ that breaks the amazing PRF, then we can build an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to break DDH.

Diffie-Hellman assumption security game (under the standard model). We have some adversary $\mathcal{A}$ that can break the PRF. We have $\mathcal{B}$ which we are going to build. $\mathcal{B}$ will interact with the DDH challenger. The goal of $\mathcal{B}$ is to win the DDH game.

The DDH game. The DDH challenger is going to take $x, y, z \leftarrow^R \mathbb{Z}_q$ and then if $b = 0$ send $X = g^x, Y = g^y, Z = g^{xy}$ to $\mathcal{B}$. If $b = 1$ the DDH challenger will do the same thing, except that $Z = g^z$ instead. $\mathcal{B}$sends $\hat{b}$ to the DDH challenger, who outputs 1 iff $\hat{b} = b$.

The PRF game. $\mathcal{B}$ is going to respond to questions from $\mathcal{A}$. $\mathcal{A}$ is allowed to ask queries of the form "evaluate the PRF at m". $\mathcal{B}$ responds with the evaluation $f(m)$. $\mathcal{A}$ will be able to evaluate the PRF at many points and will decide if it is interacting with a PRF or a random function. It will forward $\hat{b}$ to $\mathcal{B}$ who forwards $\hat{b}$ to the DDH challenger as described.

This proof works in the standard model. Now we introduce some components involved in the ROM.

In the ROM, everyone has access to the random oracle. $\mathcal{A}$ is an algorithm that is allowed to interact with the random oracle. $\mathcal{A}$ is allowed to make an additional query to the random oracle: "evaluate the random oracle at m". $\mathcal{B}$ must respond with the evaluation of the random oracle at m. $\mathcal{B}$ needs to use $\mathcal{A}$ to produce $\hat{b}$.

Central questions that $\mathcal{B}$ must address:

1. How will $\mathcal{B}$ respond to the different queries submitted to it by $\mathcal{A}$? (How to answer the queries to the random oracle)

2. How is $\mathcal{B}$ going to go from the $\hat{b}$ it gets from $\mathcal{A}$ to the $\hat{b}$ that it passes onto the DDH challenger? (How to answer the PRF questions)

3.4.8 Responding to queries to the random oracle

Definition 3.8 *Programming the random oracle. When an intermediate adversary impersonates the random oracle and responds to queries to the random oracle at certain points with some chosen response.*

Dirty trick (programming). $\mathcal{B}$ doesn't actually use the random oracle, it just pretends. Whenever $\mathcal{A}$ asks for a query to the random oracle, it will sample some value $\alpha \leftarrow^R \mathbb{Z}_q$ and set $H(m) \leftarrow X^\alpha$. X^α is distributed uniformly at random in $\mathbb{G}$. Because of this, $\mathcal{A}$ can't tell that this response is not from the random oracle. In other words, $\mathcal{A}$ is expecting to interact with a random oracle and a random oracle is something that responds with evaluations of a random function. Here, $\mathcal{B}$ is generating responses that look totally random so $\mathcal{A}$ doesn't know whether it is interacting with the random oracle or with $\mathcal{B}$ acting as the random oracle.

3.4.9 Responding to queries to the PRF

Proof.

- Sample $\alpha \leftarrow^R \mathbb{Z}_q$, set $H(m) \leftarrow X^\alpha$ (response to the random oracle)
- Send Z^α (response to queries to F)

Let's consider the two possible worlds:

- $b = 0$, $Z^\alpha = (g^{xy})^\alpha = (X^\alpha)^y$. Thus, the values that $\mathcal{B}$ is presenting to $\mathcal{A}$ is consistent with PRF evaluations with key y.
- $b = 1$, $Z^\alpha = (g^z)^\alpha =$ uniformly random. Thus, the values that $\mathcal{B}$ is presenting to $\mathcal{A}$ is what looks like evaluations of a random function.

Recalling that $\mathcal{A}$ is a distinguisher in the PRF game, it can give $\mathcal{B}$ a $\hat{b}$ that distinguishes whether $\mathcal{B}$ got a real Diffie-Hellman with xy or z. This means that $\mathcal{B}$ has the same advantage in the DDH game as $\mathcal{A}$ has in the PRF game. Because the DDH game is assumed to be hard, $\mathcal{A}$ should not exist. ■

Q. Why does $\mathcal{A}$ need to use $\mathcal{B}$ to make random oracle queries? Why can't it query the random oracle on its own (since everyone has access to the random oracle)? $\mathcal{B}$ uses $\mathcal{A}$ as a subroutine, so it fills in the role of the random oracle. We, in writing the proof, get to pick how $\mathcal{A}$ and $\mathcal{B}$ act in the proof.

3.4.10 Remaining open problems with the ROM

- The ROM is a heuristic, but it seems to work well in practice and gives us very fast schemes. In applications, we replace the random oracle with a hash function.
- There are some schemes that are (contrived) schemes that are secure in the ROM, but if we were to replace the random oracle with any hash function, they will *always* be insecure in the standard model (this is upsetting)! Thankfully, these schemes are largely contrived and thus far don't seem to have ramifications for real world crypto systems. In particular, as of now there is no known attack to real world crypto systems that utilize weaknesses in the ROM.
- Some people don't like the "dirty trick" programming since there's no real analog to programming a random oracle with a hash function that is fixed in advance (and you can't change its values).

Despite these, pretty much all real world crypto systems use the ROM.

3.4.11 Suitable hash functions to replace random oracles

Q. Why not SHA256? It is a Merkle-Damgard function and is therefore susceptible to length-extension attacks. Thus, vanilla SHA256 does not look truly random in the ROM.

If you want to use SHA256, you should use it in an HMAC with key $0 : H(\mathsf{opad} \| H(\mathsf{ipad} \| m))$ (equivalent to running HMAC without the key). You can also use SHA512/256 (just SHA512 with the second half thrown out).

Recall: It is difficult to implement your own crypto!

References

[1] Boneh, D., and Shoup, V. *A Graduate Course in Applied Cryptography.* 2019.