PRFs and the GGM construction

Noah Stephens-Davidowitz

June 8, 2023

1 Pseudorandom functions

In the last lecture, we saw how to build *stateful* semantically secure secret-key encryption using a PRG. But, we still have not built a true semantically secure encryption scheme under our original definition.

Now, we fix this by showing how to construct an efficiently computable *pseudorandom* function, a PRF. Formally, it is easier to define "a pseudorandom function" as a *keyed* function, also known as a function *family*, $F_{\boldsymbol{s}}(\boldsymbol{r})$, where $\boldsymbol{s} \in \{0,1\}^*$ and the length of the input $\boldsymbol{r}$ depends on the length of $\boldsymbol{s}$. This notation is formally equivalent to writing $F(\boldsymbol{s}, \boldsymbol{r})$, but we write $F_{\boldsymbol{s}}(\boldsymbol{r})$ to emphasize that we think of $\boldsymbol{s}$ as the key and $\boldsymbol{r}$ as the input. When the key $\boldsymbol{s}$ is n bits long, our PRF will take n bits as input and output $\ell(n)$ bits. (One can be slightly more general and take $m(n)$ bits of input for any $m(n) = \text{poly}(n)$, but it does not change much.)

The security of a PRF is defined in terms of the following two games. In one game, the adversary $\mathcal{A}$ is given *oracle* access to $F_{\boldsymbol{s}}$. I.e., the challenger samples $\boldsymbol{s} \sim \{0,1\}^n$, and $\mathcal{A}$ can make as many queries as it likes (though the number of queries must of course be polynomially bounded, because $\mathcal{A}$ must run in polynomial time) to $F_{\boldsymbol{s}}$. Each query consists of an n-bit string $\boldsymbol{x}$, and $\mathcal{A}$ receives in response an $F_{\boldsymbol{s}}(\boldsymbol{x})$. Eventually $\mathcal{A}$ outputs either zero or one.

Super succinct (and convenient!) notation for this game is given by $b' \leftarrow \mathcal{A}^{F_{\boldsymbol{s}}}(1^n)$ where $\boldsymbol{s} \sim \{0,1\}^n$, which means "$b'$ is the output of $\mathcal{A}$ on input 1^n with oracle access to $F_{\boldsymbol{s}}$ for $\boldsymbol{s} \sim \{0,1\}^n$." Giving $\mathcal{A}$ "oracle access" to $F_{\boldsymbol{s}}$ means that $\mathcal{A}^{F_{\boldsymbol{s}}}$ is an algorithm that has access to an extra procedure that computes $F_{\boldsymbol{s}}$. The full game is shown below.

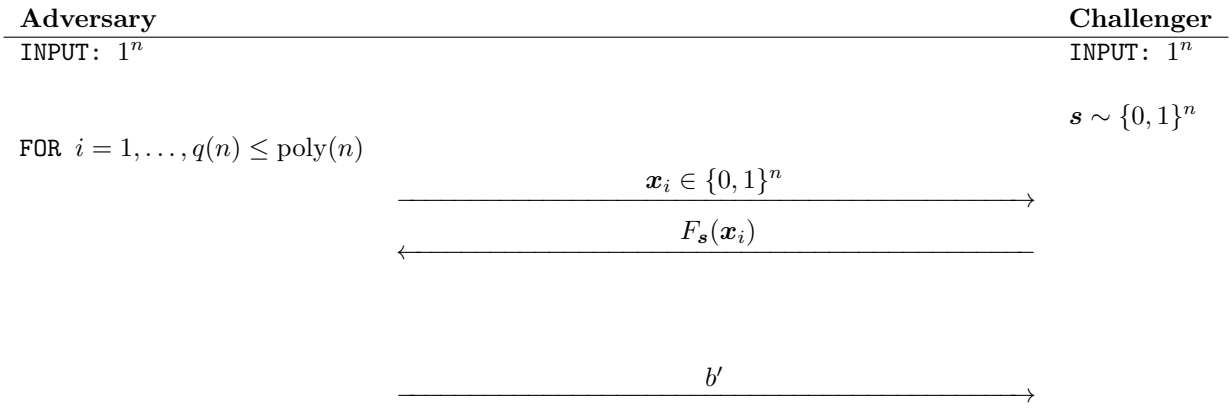

Figure 1: The game in which $\mathcal{A}$ is given oracle access to $F_{\boldsymbol{s}}$ for $\boldsymbol{s} \sim \{0,1\}^n$.

In the second game, the pseudorandom bits $F_{\boldsymbol{s}}(\boldsymbol{x}_i)$ are replaced by truly random bits $\boldsymbol{y}_i \sim \{0,1\}^{\ell(n)}$. Specifically, in response to each *distinct* query $\boldsymbol{x}$, the challenger responds with a uniformly random string $\boldsymbol{y} \sim \{0,1\}^{\ell(n)}$. However, notice that in the game with $F_{\boldsymbol{s}}$, if $\mathcal{A}$ makes the same query $\boldsymbol{x}$ twice, it will get the same answer $F_{\boldsymbol{s}}(\boldsymbol{x})$ both times. So, the challenger does the same thing in our new game. Specifically, it records the queries and makes sure to respond to duplicate queries in a consistent manner. (This makes the game itself a little tedious to write out, but the challenger's code is literally just implementing the procedure "respond randomly unless you have seen this query before, in which case give the same response that you gave previously.")

Another way to view this is to consider a *random function* $H : \{0,1\}^n \to \{0,1\}^{\ell(n)}$. I.e., let $\mathcal{H}_{n,\ell} := \{H : \{0,1\}^n \to \{0,1\}^\ell\}$ be the set of *all* possible functions from n bits to ℓ bits. Then $H \sim \mathcal{H}_{n,\ell}$ is a random function. In other words, for every input $\boldsymbol{x} \in \{0,1\}^n$ $H(\boldsymbol{x})$ is a uniformly random and independent bit string of length ℓ. We can describe our second game succinctly using the notation $b' \leftarrow \mathcal{A}^H(1^n)$ for uniformly random $H \sim H_{n,\ell(n)}$. Here is the game written out formally.

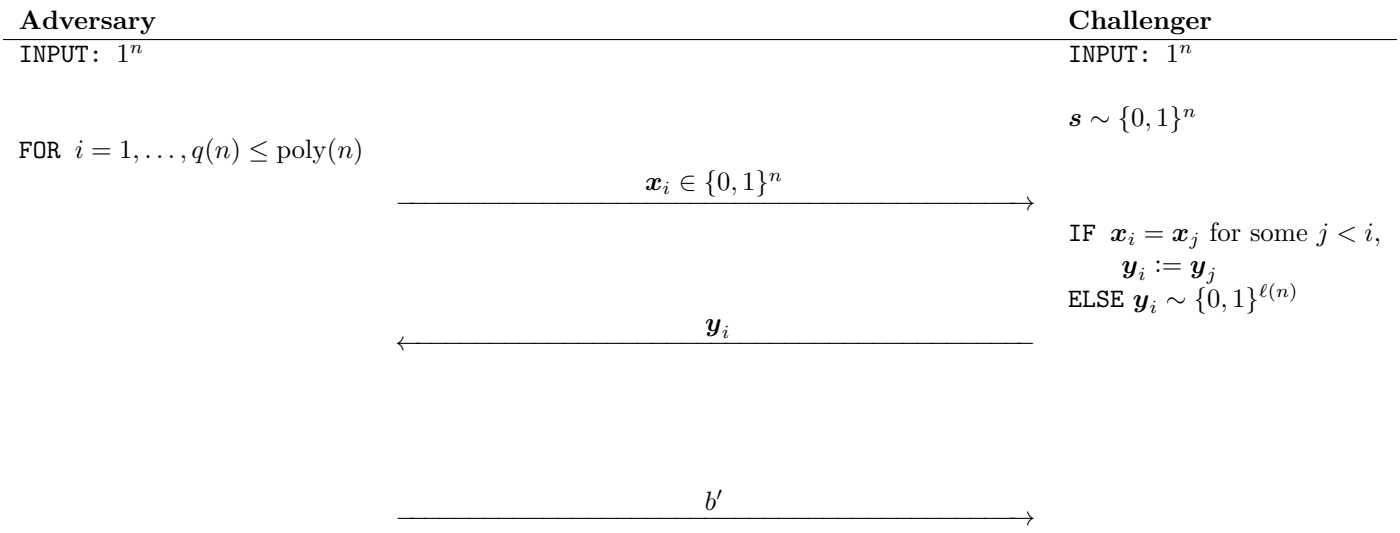

Figure 2: The game in which $\mathcal{A}$ is given oracle access to $H \sim H_{n,\ell(n)}$.

(You might wonder why I did not simply have the challenger sample $H \sim H_{n,\ell(n)}$ and then respond to each query $\boldsymbol{x}_i$ with $H(\boldsymbol{x}_i)$. This *would* yield an equivalent game. But, a random function H is not efficiently computable in general! And, it will cause problems for us later if our challenger does not run in polynomial time. So, we avoid this mess via a "lazy sampling" technique. I.e., rather than choose a fixed random function H at the beginning of the game and then computing $H(\boldsymbol{x})$ repeatedly (which cannot be done efficiently), the challenger "only decides what $H(\boldsymbol{x})$ should be when she needs to.")

With this, we can define a PRF. Notice that the security definition is exactly the same as saying "no PPT adversary has non-negligible advantage in distinguishing the first game above from the second game above." We have just used this very convenient oracle notation $\mathcal{A}^{\cdot}$ to write that quite succinctly.

Definition 1.1 (Pseudorandom function)**.** *A function* $F_{\boldsymbol{s}} : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ *for* $\boldsymbol{s} \in \{0,1\}^*$ *and*

$n := |\boldsymbol{s}|$ *is a pseudorandom function (PRF) if it satisfies the following.*

1. **Efficiently computable.** *There is a PPT algorithm* $\mathcal{A}$ *such that* $\mathcal{A}(\boldsymbol{s}, \boldsymbol{x}) = F_{\boldsymbol{s}}(\boldsymbol{x})$ *for all integers* $n \in \mathbb{N}$*, keys* $\boldsymbol{s} \in \{0,1\}^*$*, and inputs* $\boldsymbol{x} \in \{0,1\}^{|\boldsymbol{s}|}$*.*

2. **Pseudorandom.** *For all PPT algorithms* $\mathcal{A}$*, there exists a negligible function* $\varepsilon(n)$ *such that for all* $n \in \mathbb{N}$*,*

$$\left| \Pr_{\boldsymbol{s} \sim \{0,1\}^n} [\mathcal{A}^{F_{\boldsymbol{s}}}(1^n) = 1] - \Pr_{H \sim \mathcal{H}_{n,\ell(n)}} [\mathcal{A}^{H}(1^n) = 1] \right| \leq \varepsilon(n) \ .$$

2 The GGM construction of a PRF from a PRG

We now show how to build a PRF from a PRG. The (elegant) construction that we will use is due to Goldreich, Goldwasser, and Micali [GGM86], and is typically just called the GGM PRF.

To get some intuition for the construction, let's recall our idea from last class for "getting more pseudorandom bits" out of a PRG. I.e., given a PRG G that maps n bits to $2n$ bits, we define $G_0(\boldsymbol{s})$ and $G_1(\boldsymbol{s})$ as the first n bits of the output and the second n bits of the output respectively. Then, to get rn pseudorandom bits from G using a seed $\boldsymbol{s}_0 \sim \{0,1\}^n$ of length n, we define $\boldsymbol{s}_i := G_0(\boldsymbol{s}_{i-1})$ and $\boldsymbol{x}_i := G_1(\boldsymbol{s}_{i-1})$. Notice that these quantities are efficiently computable for polynomial i. Our pseudorandom bits are then $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_r) \in \{0,1\}^{rn}$.

We can equivalently think of G and $\boldsymbol{s}$ as defining a *function*, $F_{G,\boldsymbol{s}}(r)$ that takes as input $r \in \mathbb{N}$ and outputs $\boldsymbol{x}_r$. Unfortunately, it takes time proportional to r to compute $F_{G,\boldsymbol{s}}(r)$. We are therefore effectively restricted to computing this function on polynomially many inputs r. Equivalently, we can think of the domain of $F_{G,\boldsymbol{s}}$ as $\{0,1\}^m$, where $m = O(\log n)$ so that $|\{0,1\}^m| = \text{poly}(n)$.

To get a PRF, we need to handle n-bit inputs. I.e., instead of taking an n-bit random seed $\boldsymbol{s} \in \{0,1\}^n$ and expanding it into $\text{poly}(n)$ pseudorandom bits, a PRF in some sense takes an n-bit random seed and expands it into $\ell \cdot 2^n$ pseudorandom bits! And we must have efficient access to these bits.

The problem with $F_{G,\boldsymbol{s}}$ is of course that it needs to compute all of the values $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{r-1}$ before computing $\boldsymbol{x}_r$. As a data structure, it is a *linked list*—where the entries in the list are indexed by $\boldsymbol{s}_i$ and labeled with $\boldsymbol{x}_i$. To find the label $\boldsymbol{x}_r$ of the rth node in the list, we use G_0 to iteratively compute the indices $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{i-1}$, before finally using G_1 to compute $\boldsymbol{x}_r$. The GGM PRF replaces this linked list with a balanced binary *tree*. (You can make a career in computer science just by finding places to replace linked lists with trees :).)

Here is the construction. Let's write $\boldsymbol{s}_0 := G_0(\boldsymbol{s})$ and $\boldsymbol{s}_1 := G_1(\boldsymbol{s})$. (Note that I have redefined things here.) Then, $\boldsymbol{s}_{00} := G_0(\boldsymbol{s}_0)$, $\boldsymbol{s}_{01} = G_1(\boldsymbol{s}_0)$, $\boldsymbol{s}_{10} := G_0(\boldsymbol{s}_1)$, and $\boldsymbol{s}_{11} := G_1(\boldsymbol{s}_1)$. (Here and below, we will use the notation $\boldsymbol{rt}$ for the concatenation of the bit strings $\boldsymbol{r}$ and $\boldsymbol{t}$.) More generally, $\boldsymbol{s}_{\boldsymbol{r}0} := G_0(\boldsymbol{s}_{\boldsymbol{r}})$ and $\boldsymbol{s}_{\boldsymbol{r}1} := G_1(\boldsymbol{s}_{\boldsymbol{r}})$. I.e., unraveling the definition, we have for $\boldsymbol{r} := (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$

$$\boldsymbol{s}_{\boldsymbol{r}} := G_{r_n}(G_{r_{n-1}}(\cdots(G_{r_1}(\boldsymbol{s})\cdots))) \ .$$

Then, the GGM PRF $F_{\boldsymbol{s}} : \{0,1\}^n \to \{0,1\}^n$ is defined as $F_{\boldsymbol{s}}(\boldsymbol{r}) := \boldsymbol{s}_{\boldsymbol{r}}$. (It's quite important that the input size is fixed for fixed $\boldsymbol{s}$! If we reveal $\boldsymbol{s}_{\boldsymbol{r}}$ for strings $\boldsymbol{r}$ with length, say, $n-1$ as well, then the adversary can certainly distinguish, say, $\boldsymbol{s}_{\boldsymbol{r}0}$ from random, since $\boldsymbol{s}_{\boldsymbol{r}0} = G_0(\boldsymbol{s}_{\boldsymbol{r}})$. So, it is crucial that an adversary making queries to $F_{\boldsymbol{s}}$ will only learn $\boldsymbol{s}_{\boldsymbol{r}}$ for strings $\boldsymbol{r}$ that have length exactly n.) See Figure 3.

To prove the security of this construction, we will need the following lemma, which we proved in the previous lecture.

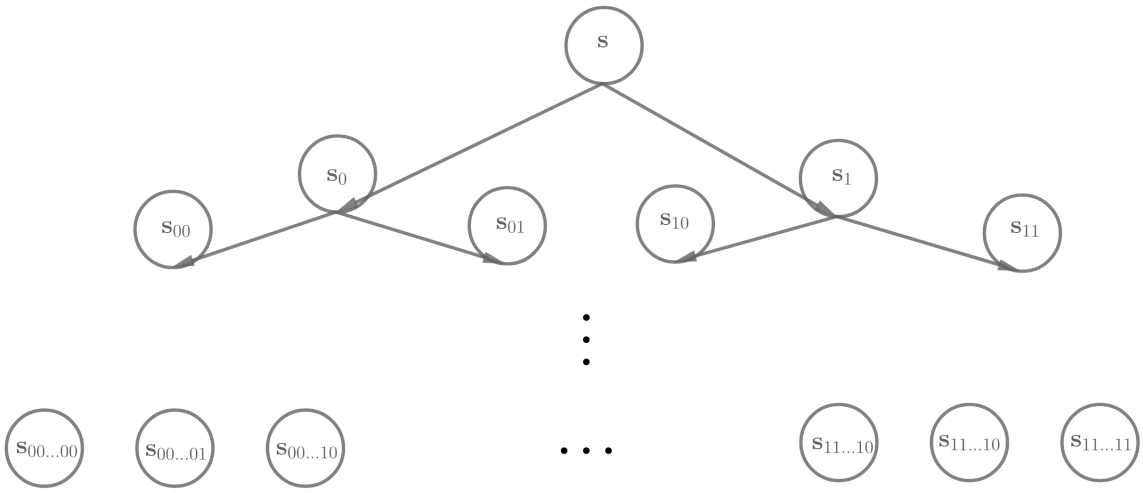

Figure 3: The GGM PRF.

Lemma 2.1. *If G is a length-doubling PRG, then for any PPT $\mathcal{A}$ and any polynomial $p(n)$, there exists a negligible $\varepsilon(n)$ such that for all $n \in \mathbb{N}$,*

$$\left| \Pr_{\boldsymbol{s}_1,\ldots,\boldsymbol{s}_{p(n)} \sim \{0,1\}^n} [\mathcal{A}(1^n, G(\boldsymbol{s}_1), \ldots, G(\boldsymbol{s}_{p(n)})) = 1] - \Pr_{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{p(n)} \sim \{0,1\}^{2n}} [\mathcal{A}(1^n, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{p(n)}) = 1] \right| \leq \varepsilon(n) \,.$$

Theorem 2.2. *If G is a PRG, then $F_{\boldsymbol{s}}$ as defined above is a PRF.*

Proof. The proof is by a hybrid argument. Let F_0 be the oracle $F_{\boldsymbol{s}}$ sampled honestly. I.e., F_0 has a key $\boldsymbol{s} \sim \{0,1\}^n$ sampled uniformly at random, and F_0 then returns $F_{\boldsymbol{s}}(\boldsymbol{r})$ for any query $\boldsymbol{r}$ made by the adversary $\mathcal{A}$. F_1 has two keys $\boldsymbol{s}_0, \boldsymbol{s}_1 \sim \{0,1\}^n$ sampled independently and uniformly at random, and it behaves like the GGM PRF with the pseudorandom strings in the first level below the root replaced by the pseudorandom $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$. I.e., we "remove the root and split the tree into two trees," as in Figure 4.

More generally F_i has 2^i keys, given by $\boldsymbol{s}_{00\ldots0}, \ldots, \boldsymbol{s}_{11\ldots1} \sim \{0,1\}^n$ all sampled uniformly and independently at random, and to compute F_i, we compute the GGM PRF, except that we replace the whole ith level of the tree by these random keys, rather than the pseudorandom elements in the "honest" construction. Notice in particular that F_n is a purely random oracle.

So, if some adversary $\mathcal{A}$ has non-negligible advantage ε in breaking the PRF (i.e., if it distinguishes the PRF from a random oracle with probability ε), then there must be some index $1 \leq i \leq n$ such that $\mathcal{A}$ distinguishes F_i from F_{i-1} with probability at least ε/n.

It therefore suffices to show how to use an adversary $\mathcal{A}$ that distinguishes between F_i and F_{i-1} in order to build an adversary $\mathcal{B}$ that breaks the security of the PRG. Or, rather, we will build an adversary $\mathcal{B}$ that breaks the game described in Lemma 2.1. To do this, we must decide what p should be.

Intuitively, we want to use the p strings $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$ that $\mathcal{B}$ receives as input to replace the strings labeling the nodes in the ith row of Figure 3. Then, if the $\boldsymbol{x}_j$ are random, we will faithfully reproduce F_i. If they are pseudorandom, we will faithfully reproduce F_{i-1}. So, an $\mathcal{A}$ that distinguishes between an F_i and F_{i-1} will yield a $\mathcal{B}$ that breaks our PRG.

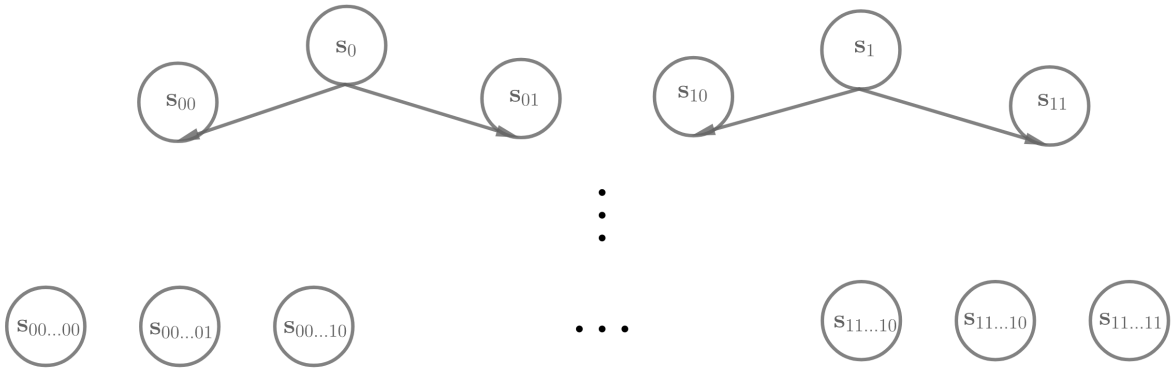

Figure 4: The GGM tree with the root removed, used in Game 1. Here, $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ are both sampled uniformly at random

The only problem is that the number of nodes in the ith row is 2^i, which is superpolynomial for large i. We need p to be polynomial in n. (This isn't just a technical issue. Lemma 2.1 is simply false for superpolynomial p.) To handle this, we will use the lazy sampling idea that we discussed above. I.e., we will not assign a string to each node in the ith row right away. Instead, we will only assign strings to nodes that are queried, and we will do so only when they are queried.

So, let p be a polynomial that bounds the number of queries that $\mathcal{A}$ makes to its oracle. $\mathcal{B}$ receives as input the strings $\boldsymbol{x}_1 := (\boldsymbol{x}_{1,0}, \boldsymbol{x}_{1,1}), \ldots, \boldsymbol{x}_p := (\boldsymbol{x}_{p,0}, \boldsymbol{x}_{p,1}) \in \{0,1\}^{2n}$. (To be clear, we have simply split the $2n$-bit strings into pairs of n-bit strings for convenience.) When $\mathcal{B}$ receives its first oracle query $\boldsymbol{r} \in \{0,1\}^n$ from $\mathcal{A}$, it behaves as follows. Let $\boldsymbol{r}' \in \{0,1\}^{i-1}$ be the first $i-1$ bits of $\boldsymbol{r}$. $\mathcal{B}$ sets $\boldsymbol{s}_{\boldsymbol{r}'0} := \boldsymbol{x}_{1,0}$ and $\boldsymbol{s}_{\boldsymbol{r}'1} := \boldsymbol{x}_{1,1}$. It then computes $\boldsymbol{s}_{\boldsymbol{r}} := G_{r_n}(G_{r_{n-1}}(\cdots G_{r_{i+1}}(\boldsymbol{s}_{\boldsymbol{r}r_i})\cdots))$, just like F_i.

For the jth oracle query $\boldsymbol{r}$, we again define $\boldsymbol{r}' \in \{0,1\}^{i-1}$ to be the first $i-1$ bits of $\boldsymbol{r}$. If $\boldsymbol{s}_{\boldsymbol{r}'0}, \boldsymbol{s}_{\boldsymbol{r}'1}$ have not been defined previously, then $\mathcal{B}$ sets $\boldsymbol{s}_{\boldsymbol{r}'0} := \boldsymbol{x}_{j,0}$ and $\boldsymbol{s}_{\boldsymbol{r}'1} := \boldsymbol{x}_{j,1}$. It then computes $\boldsymbol{s}_{\boldsymbol{r}} := G_{r_n}(G_{r_{n-1}}(\cdots G_{r_{i+1}}(\boldsymbol{s}_{\boldsymbol{r}r_i})\cdots))$ as before.

After at most p queries $\mathcal{A}$ will return either 0 or 1, and $\mathcal{B}$ simply outputs the same.

Clearly, $\mathcal{B}$ is efficient. To show that the advantage of $\mathcal{B}$ in the game defined in Lemma 2.1 is equal to the advantage of $\mathcal{A}$ in the ith hybrid, we simply need to observe that (1) when $(\boldsymbol{x}_{j,0}, \boldsymbol{x}_{j,1}) = G(\boldsymbol{s}_j)$ for independent $\boldsymbol{s}_j$, then the responses of $\mathcal{B}$ to the queries made by $\mathcal{A}$ are distributed identically to the responses of F_{i-1}; and (2) when the $\boldsymbol{x}_{j,b}$ are uniformly and independently random, then we need to show that the responses are distributed identically to the responses of F_i. Both observations follow immediately from the definitions of F_i and the game in Lemma 2.1. □

References

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4), 1986.