CSCI-GA.3210-001 MATH-GA.2170-001	**Introduction to Cryptography** October 21, 2008

Lecture 8

Lecturer: Yevgeniy Dodis *Spring 2012*

In this lecture we build some IND-CPA-secure schemes. Our initial schemes, based on iterating a PRG, are *stateful*. We then turn to the question if there exists a stateless IND-CPA-secure SKE.

1 Building CPA-Secure SKE: Stream Ciphers

No CPA-secure encryption can be deterministic, since the adversary B_2 can always ask its oracle to re-encrypt m_0 and m_1. There are two ways out. One way is to make the encryption probabilistic. We will come back to this later. For now, we explore the second way: make the encryption *stateful*. Namely, every time the encryption/decryption operations are performed, the secret key (so called *state*) will change as well.

DEFINITION 1 Stateful SKE is a usual SKE with the following modifications. Encryption and decryption functions E and D additionally output the new value of the secret key s, which be used for subsequent encryption/decryptions. Formally, $E(m; s)$ ouputs a pair (c, s'), where c is the ciphertext and s' is the new state (secret key). Similarly, $D(c; s)$ ouputs a pair (m, s'), where m is the plaintext and s' is the new state (secret key). We require that the decryption and encryption are always in sink: a decryption of c must be performed after every encryption of m. $\diamondsuit$

We must also mention that in the definition of CPA-secure SKE, oracle access must keep state as well. This is so because the policy of updating the secret key should be considered public as a part of the algorithm's encryption and decryption functions.

We notice that having state eneables us to possibly have *determinisitc stateful* schemes, as we illustrate.

Example 1: Blum-Micali Generator For One Bit. CPA-secure encryption of multiple messages can be achieved using the following stateful algorithm.
Let $f : M_k \to M_k$ be a OWP and $h : M_k \to \{0, 1\}$ be a hardcore bit of f.
Encryption function: $E_s(b) \to (c = h(s) \oplus b, s \leftarrow f(s))$.
Decryption function: $D_s(c) = (m = h(s) \oplus c, s \leftarrow f(s))$.

In other words, we simply keep applying the Blum-Micali PRG and using each successive bit as the next one-time pad. The CPA-security of this scheme follows immediately from the one-time pad lemma and the fact that Blum-Micali generator is a PRG. In fact, even if we reveal the adversary the entire state (secret key) after we encrypt the challenge bit b, we proved that the adversary cannot predict b. Thus, this method is *forward-secure*. Loosing the current secret key protects all the previous encryptions.

Notice also that 1-bit limitation is not an issue. Simply generate more bits to encrypt longer message, and use these bits as a longer one-time pad. This is because we showed

the closure of CPA-secure (as opposed to the one-time secure) symmetric encryption under composition.

Example 2: Using Any PRG. Previous construction could be viewed as using the Blum-Micali generator $G(x) = G'(x) \circ f^n(x)$ (see Lecture 5 for notation). In fact, if we write $G(x) = G_1(x) \circ G_2(x)$, where $G : \{0,1\}^k \to \{0,1\}^{n+k}$, and $|G_1(x)| = n$, $|G_2(x)| = k$, we get the that the above construction is a special case of the following more general construction, which works for *any* PRG G.

Let $s \leftarrow \{0,1\}^k$ be the initial secret.
To encrypt $m \in \{0,1\}^n$ having current state s, output $c = m \oplus G_1(s)$, and update $s = G_2(s)$.
To decrypt $c \in \{0,1\}^n$ having current state s, output $m = c \oplus G_1(s)$, and update $s = G_2(s)$.

Again, CPA-security of this stateful construction follow quite easily from the one-time pad lemma, since $G_1(s) \circ G_1(G_2(s)) \circ \ldots \circ G_1(G_2(\ldots G_2(s)\ldots))$ was shown to be a PRG in the previous lectures. In fact, since the PRG above is forward-secure, we get that our general scheme is forward as well (i.e., loosing current state does not compromise previous encryptions).

Example 3: Using DDH. This is a special case of Example 2 above, which is obtained by noticing that the DDH assumption immediately gives rise to a new and effiicent PRG! Indeed, let us assume that the prime $p = 2q+1$ and the and generator g of the subgroup $\mathbb{G}$ or quadratic residues modulo p are fixed and public. Now consider the following function $G : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{G} \times \mathbb{G} \times \mathbb{G}$ (recall from the last lecture that $\mathbb{Z}_q$ is isomorphic to $\mathbb{G}$, and also we can easily map between them, so one can view $\mathbb{G}$ as going from Z_q^2 to Z_q^3):

$$G(a,b) = \langle g^a, g^b, g^{ab} \rangle$$

The DDH assumption states that this G is indeed a PRG (with output indistingushable from $\langle g^a, g^b, g^c \rangle$ for random a, b, c). This gives an efficient PRG which expands its random input by 50%, going from $2k$ to $3k$ bits.

Even more optimized, let us fix another random generator h of $\mathbb{G}$ (i.e., one can think of $h = g^a$ for a random a which is chosen once and for all). Now define length-doubling $G : \mathbb{Z}_q \to \mathbb{G} \times \mathbb{G}$ (or, alternatively, from $\mathbb{Z}_q$ to $\mathbb{Z}_q^2$) by

$$G(b) = \langle g^b, h^b \rangle$$

Once again, writing $h = g^a$ for a random (unknown) a, the attacker's view $\langle g, h, G(b) \rangle \equiv \langle g, g^a, g^b, g^{ab} \rangle \approx \langle g, g^a, g^b, g^c \rangle$, implying that G is a PRG which goes from k to $2k$ bits. As it turns out, this construction works even if $h = g^a$ is fixed "forever" and only new b is chosen for every fresh invocation of G.

Lemma 1 *For any polynomial $t = t(k)$, the DDH assumption implies that, for random $a, b_1, \ldots, b_t, c_1, \ldots, c_t \leftarrow \mathbb{Z}_q$, we have*

$$\langle g, g^a, g^{b_1}, g^{ab_1}, \ldots g^{b_t}, g^{ab_t} \rangle \approx \langle g, g^a, g^{b_1}, g^{c_1}, \ldots g^{b_t}, g^{c_t} \rangle \quad (1)$$

Proof: Given a tuple $\langle g, g^a, g^b, g^c \rangle$, where c is either $ab \bmod q$ or random, for $j = 1 \dots t$, pick random $d_j, e_j \leftarrow \mathbb{Z}_q$ and define

$$g^{b_j} = (g^b)^{d_j} g^{e_j}; \quad g^{c_j} = (g^c)^{d_j} (g^a)^{e_j}$$

Notice, g^{b_j} and g^{c_j} can be computed efficiently from g^a, g^b, g^c and d_j, e_j. Mathematically, however, if we write $c = ab + z$, where $z = 0$, when $c = ab$, and random, when c is random, we have

$$b_j = bd_j + e_j \bmod q; \qquad c_j = (ab + z)d_j + ae_j = zd_j + a(bd_j + e_j) = zd_j + ab_j$$

Notice, since e_j is random, all values $b_j = bd_j + e_j$ are random and independent of each other, even conditioned on known d_j. Moreover, if $z = 0$ (DDH-tuple), we always have $c_j = ab_j$, meaning that we get precisely the left hand side of Equation (1), for random b_j. On the other hand, when z is random (non-DDH typle), with high probability $z \neq 0$, and all the values $c_j = zd_j + ab_j$ are completely random and independent from each other, since the d_j's are completely random and independent from each other (and the b_j's). Hence, ignoring the negligible case of $z = 0$, we get precisely the right hand side of Equation (1).

Notice, the reduction above is *tight*, as we surprisingly did not use the hybrdi argument! □

We remark that SKE constructions above, namely those stateful schemes that simply keep outputting a steam of pseudorandom bits (to be used as one-time pads), are called *stream ciphers*. They should be contrasted with *block ciphers* we will mention the next lecture.

2 Criticism + Looking Ahead

We notice that in stateful schemes, the sender and the receiver must be synchronized at all times to ensure correctness of decryption. This may be achieved by either agreeing on the policy of updating the secret key at encryption and decryption of each message. This is a disadvantage.

As one was to circumvent this, assume we can ensure that the state of the scheme has the form $(s, count)$, where s is the "real" secret key, which *never changes*, while *count* is a simple counter that tells how many messages have been encrypted so far. Namely, the only change to the state is the instruction $count = count + 1$. In this case, even if the synchronization is temporarily lost, the sender and the recipient can exchange their counters, and reset the counter to the maximum value. It is easy to see that this exchange of counters will not conflict with IND-security, while will partially eliminate the problem of synchronization.

We notice, however, that our scheme from the previous section is not of this form. Thus, the first question we ask is:

- **Question 1:** Can we build a deterministic stateful CPA-secure SKE with the counter, as explained above?

The second question is whether we can have a stateless SKE, as we had for PKE. Naturally, this has to be randomized.

- **Question 2:** Can we build a randomized stateless CPA-secure SKE?

Finally, assuming the answer to the questions above is "yes",

- **Question 3:** What achieves greater security/efficiency: (state+determinism) or (no state+randomization)?

To answer all these questions, we introduce the concept of Pseudo Random Function Family (PRF), a really strong cryptographic primitive with several interesting properties. Afterwards, we present a very important application of PRFs, namely a new argument technique which allows us to separate the discussion of efficiency issues from the analysis of the security of the system.

3 Pseudo Random Function Family

3.1 Introduction

When we introduced Pseudo Random Generators, we saw that they are "efficient stretchers of randomness": given a truly random k-bit string, a PRG outputs a much longer (but polynomial in k) stream of bits that "looks random" with respect to any real (i.e. PPT) algorithm. Now we want to go further, and try to answer the question: can we do better? In other words, can we extract an exponential amount of pseudo random bits from a k-bit long seed?

Stated this way, this question does not really make sense, since it is impossible to even write down such a sequence efficiently. Anyway, we may want to know whether we can get an *implicit* representation of an exponential number of bits. One well-known class of exponentially long objects having "short" descriptions is that of *computable functions*, since the dimension of a mapping from $\{0,1\}^{\ell}$ to $\{0,1\}^{L}$ is $2^{\ell} \cdot L$.
After all, our question may be rephrased as:

> Can we use a k-bit long seed s to efficiently sample a computable function f_s from the space $\mathcal{R}(\ell(k), L(k))$ of all possible functions $F : \{0,1\}^{\ell(k)} \rightarrow \{0,1\}^{L(k)}$, in an 'almost-random' manner?

3.2 Definition

The essence of the above question can be formalized by the following definition.

Definition 2 [Pseudo Random Function Family]
A family $\mathcal{F} = \langle f_s \mid s \in \{0,1\}^k \rangle_{k \in \mathbb{N}}$ is called a family of $(\ell(k), L(k))$ Pseudo Random Functions if:

- $\forall k \in \mathbb{N}, \forall s \in \{0,1\}^k, \quad f_s : \{0,1\}^{\ell(k)} \rightarrow \{0,1\}^{L(k)}$;
- $\forall k \in \mathbb{N}, \forall s \in \{0,1\}^k, \quad f_s$ is polynomial time computable;
- $\mathcal{F}$ is pseudo random: $\forall$ PPT Adv

$$\left| Pr[\mathsf{Adv}^{f_s}(1^k) = 1 \mid s \leftarrow_R \{0,1\}^k] - Pr[\mathsf{Adv}^{F}(1^k) = 1 \mid F \leftarrow_R \mathcal{R}(\ell(k), L(k))] \right| \leq \mathsf{negl}(k)$$

$\diamondsuit$

In other words, for a family $\mathcal{F}$ to be pseudo random, it is not required for its generic element f_s to be indistinguishable from a function F drawn at random from $\mathcal{R}(\ell(k), L(k))$; rather, the *behavior* of any PPT adversary Adv which is given oracle access to the function f_s must be indistinguishable from the behavior of the same adversary when given oracle access to a function F:

$$\forall \text{ PPT Adv} \qquad \text{Adv}^{f_s} \approx \text{Adv}^{F}$$

To put this yet in another way, imagine there are two distinct worlds: in the first world the adversary Adv queries a function chosen from the family $\mathcal{F}$, while in the second world the adversary's queries are answered by a truly random function F. Here by "truly random function" we mean a black box which outputs a fresh random value on each invocation, except that it is *consistent*, i.e. if queried twice on the same value, it always returns the same output. Now we say that $\mathcal{F}$ is a good family of PRFs if, although the outputs given by f_s are clearly correlated while F's answers are completely independent, the behavior of the adversary is essentially the same, so that it is not possible to tell these two worlds apart.

Comments.
Observe that this is a very strong requirement: how is it possible that no efficient adversary can realize whether it has been interacting with a function f_s that uses only k bits of randomness or with a function F that consist of $2^{\ell(k)} \cdot L(k)$ random bits? A partial answer is that the adversary can only make polynomially many queries, and thus it doesn't have enough time to infer which "world" it is in.

3.3 PRFs vs PRGs

Our initial intent was to generalize the notion of PRG: this is indeed the case, since it actually turns out that PRGs can be viewed as a particular instantiation of PRFs.

In the case of a PRF, the adversary is given oracle access to the chosen function f_s; in the case of a PRG, since the output of a generator G is just polynomially long (say k^c), the adversary can be given the entire string $G(s)$.

Anyway, it is trivial to simulate the knowledge of the string $G(s)$ using oracle access to a function f_s: the adversary may ask which bit it wants to know (specifying its position), and the oracle replies with the value of that bit. Since the position of one bit in a k^c long string can be determined using $c \log k$ bits, and the answer is always one bit long, PRGs may be thought as PRFs where $\ell(k) = c \log k$ and $L(k) = 1$.

Therefore, for $\ell(k) = O(\log k)$, PRFs *degenerate* to PRGs; to ensure a gain in power with respect to PRGs, we have to enforce the *non-triviality* condition: $\ell(k) = \omega(\log k)$.

Notice, the moment non-triviality of the *input* length is ensured, there is no theoretical reason to lower bound for the *output* length value $L(k)$. This is because since any "non-trivial" family $\mathcal{F}$ of PRFs, — even the one with output lengh $L(k) = 1$, — can be easily extended to a family $\mathcal{F}'$ with $L(k) = k^c$: for each $f_s \in \mathcal{F}$, we include in $\mathcal{F}'$ the function

$f'_s : \{0,1\}^{\ell(k)-c\log k} \to \{0,1\}^{k^c}$ defined as follows:

$$f'_s(x') = \underbrace{f_s(\overbrace{0\ldots0}^{c\log k}\, x') \circ f_s(\overbrace{0\ldots1}^{c\log k}\, x') \circ \ldots \circ f_s(\overbrace{1\ldots1}^{c\log k}\, x')}_{k^c \text{ bits}}$$

Of course, in practice evaluating such a function bit-by-bit is very slow, but it demostrates that worrying about the output length is somewhat of a secondary issue, as long as we can construct a PRF with a "non-trivial" input length.

4 A general construction for PRFs

Once we have defined the notion of PRF family, we look at the problem of building such a family, and of the minimal assumption for the construction to go through. Surprisingly, it turns out that the existence of PRG implies the existence of PRF, although the transformation is too elaborated to be useful in practice. This result is due to Goldreich, Goldwasser and Micali, and is therefore known as **GGM construction**.

The GGM construction presents a loose resemblance to the technique used to obtain an IND-CPA secure stateful SKE scheme from any length-doubling PRG G. Recall from the previous lecture that to this aim we denote by $G_0(x)$ and $G_1(x)$ respectively the first and the second halves of the output of $G(x)$:

$$G : \{0,1\}^k \to \{0,1\}^{2k} \qquad G_0, G_1 : \{0,1\}^k \to \{0,1\}^k$$
$$G(x) \doteq G_0(x) \circ G_1(x)$$

To encrypt the first message, we apply G to the shared key s_0 and set the new state s_1 to be $G_0(s_0)$, while masquerading the message with the pad $p_1 = G_1(s_0)$. The next time a message must be encrypted, the generator G will be applied to the new state s_1, yielding $s_2 = G_0(s_1)$ and $p_2 = G_1(s_1)$. We can think of the whole process as the construction of an *unbalanced* binary tree (sketched in figure 1), in which we always go down to the left: the problem with this approach is that to have n leaves, we have to construct a tree with height n.

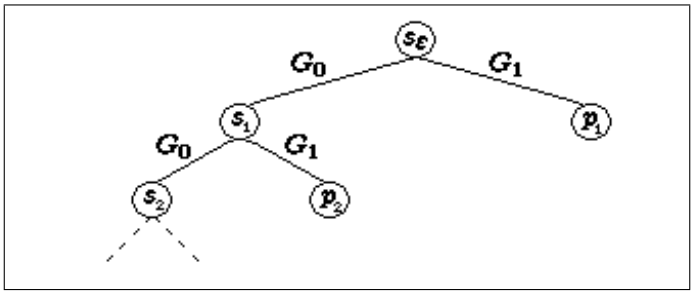

Figure 1: The unbalanced binary tree in the stateful SKE construction.

Clearly, it would be much more efficient to construct a *complete* binary tree, since this would give 2^n leaves on a tree of height n. To do so, for each fixed $s \in \{0,1\}^k$ we define

$G_x(s)$ through the following recursion:

$$G_\varepsilon(s) = s$$
$$G_{0\circ\bar{x}}(s) = G_{\bar{x}}(G_0(s))$$
$$G_{1\circ\bar{x}}(s) = G_{\bar{x}}(G_1(s))$$

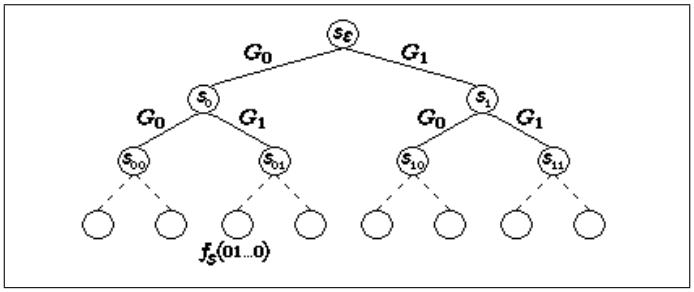

Figure 2: The complete binary tree in the GGM construction.

But how does this construction relate to Pseudo Random Functions? To sample a function $f_s : \{0,1\}^{\ell(k)} \to \{0,1\}^k$ given a random seed $s \in \{0,1\}^k$, define $f_s(x) = G_x(s)$. In this way, each input x identifies a path in the complete binary tree having s as root, and the output of the function f_s is the value associated to the leaf down such path. According to the definition of $G_x(s)$, to compute the function f_s on a single input it is necessary to evaluate the generator G $\ell(k)$ times, which is still polynomial, although not very efficient.

It is not obvious that the above construction defines a family $\mathcal{F}$ of PRFs: we are extracting $2^{\ell(k)}$ different values out of one single truly random string! Still, no efficient adversary bears a (significantly) different behavior whether it interacts with this function or with a genuine random function (i.e. a function in which the $2^{\ell(k)}$ "leaves" are all random values). This claim is proved in the following theorem, which makes an extensive use of the hybrid argument.

Theorem 1 (Goldreich-Goldwasser-Micali)
The family $\mathcal{F} = \langle f_s \mid s \in \{0,1\}^k \rangle_{k\in\mathbb{N}}$ where $f_s(x) = G_x(s)$ (as explained above) is a family of $(\ell(k), k)$ Pseudo Random Functions.

Proof: To prove this theorem we want to use the hybrid argument: anyway, in this case the situation is a little different from what we have seen so far, since what we want to prove is not that a single pair of objects are computationally indistinguishable (like in $G(s) \approx R$), but rather that an infinity of related pair of objects are indistinguishable from each other:

$$\forall \text{ PPT Adv} \qquad \mathsf{Adv}^{f_s} \approx \mathsf{Adv}^{F}$$

Accordingly, to apply the hybrid argument, we need to find a sequence of *oracles* $T_0 \ldots T_{\mathsf{poly}(k)}$ such that $f_s \equiv T_0, F \equiv T_{\mathsf{poly}(k)}$, and for all the intermediate oracle it holds that:

$$\forall \text{ PPT Adv} \qquad \mathsf{Adv}^{T_i} \approx \mathsf{Adv}^{T_{i+1}}$$

Let's start defining the appropriate sequence of oracles. Observe that both f_s and F are queried from the adversary Adv about the value (in some specific point) of the function they "represent". As a consequence, we can think of those oracles as a set of $2^{\ell(k)}$ nodes, each containing the value of the function in one of the possible inputs.

In the case of f_s, this nodes can in turn be thought as the leaves of a complete binary tree of height $\ell(k)$, whose root contains the seed s: this is the tree we talked about when discussing the construction of the function f_s from the PRG $G(x) = G_0(x) \circ G_1(x)$. We can think in a similar way also about the oracle F, even if in the case the "tree structure" is not inherent to its construction: all the nodes in the tree are "empty" nodes, except for the leaves, which contain all the random values of the F.

Stated this way, it is easier to figure out a possible way to "smoothly change" the oracle f_s into the oracle F: instead of having randomness only in the root, and computing all the rest of the tree (and in particularly the leaves, using the PRG G (as in f_s); or having all the randomness in the leaves, so that nothing is to be computed pseudorandomly (as in F), we can define the intermediate oracle T_i to have an empty tree structure from level 0 to level $i-1$, all nodes at level i containing truly random values, and the rest of the tree from level $i+1$ to level $\ell(k)$ (where we find the leaves, i.e. the values returned by this oracle) being calculated via successive applications of the pseudo random generator G.

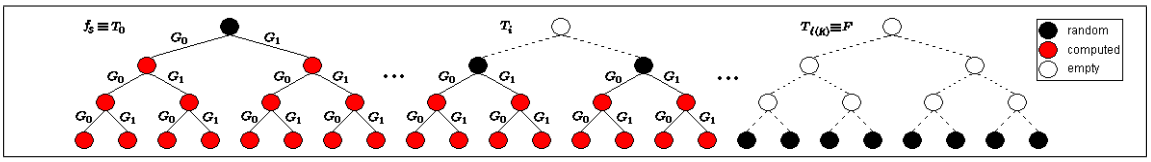

Figure 3: The sequence of oracles used in the first hybrid argument.

In this way we have constructed polynomially many intermediate oracles: in particular $f_s \equiv T_0$, since the randomness is only at level 0 (i.e. the root), and all the rest is computed through applications of G; in addition, $F = T_{\ell(k)}$, since all the structure above the level $\ell(k)$ (i.e. the last level of the tree), is empty, and the randomness is contained directly in the leaves (see figure 3). Therefore, by the hybrid argument, we can reduce the proof of the theorem to proving that:

$$\forall\ \mathsf{PPT}\ \mathsf{Adv} \qquad \mathsf{Adv}^{T_i} \approx \mathsf{Adv}^{T_{i+1}}$$

To this aim, let us fix an arbitrary adversary Adv, and prove that $\mathsf{Adv}^{T_i} \approx \mathsf{Adv}^{T_{i+1}}$: having shown that, from the generality of such adversary the above statement will hold true, and thus the proof of this theorem will follow.

Since Adv is a PPT algorithm, it can do at most a polynomial number of queries to its oracle, say $t = \mathsf{poly}(k)$. In order to prove that the behavior of Adv with oracle T_i is indistinguishable from the behavior of Adv with oracle T_{i+1}, we want to use again the hybrid argument: let's look at the correct sequence of intermediate oracle to use.

It would be tempting to consider the sequence of oracles $\overline{T_{ij}}$ in which the randomness is contained in the first j nodes at level i, and in the children (at level $i+1$) of the remaining nodes at level i (see figure 4).

Clearly the two extreme of such sequence would be T_i and T_{i+1}, but how many intermediate oracles would result? At level i there are 2^i nodes, and so when i approaches $\ell(k)$ there

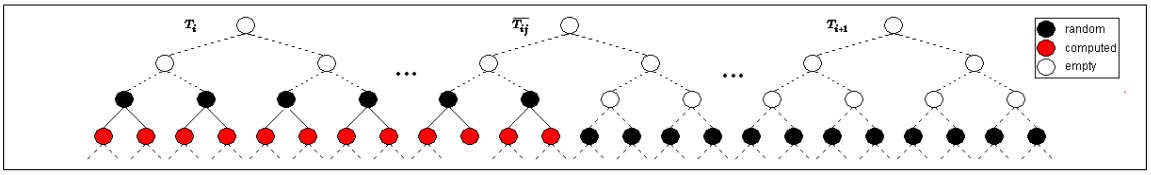

Figure 4: A first (wrong) step towards the second hybrid argument.

would be exponentially many intermediate oracles, way too much for the hybrid argument to go through.

How can we find a way out this situation? Recall that the adversary Adv queries its oracle at most t time; intuitively, the problem with the intermediate oracles $\overline{T_{ij}}$ was that they induced a "too fine" differentiation: since Adv makes just t queries, having t intermediate oracles must suffice.

Accordingly, for $j \in [0..t]$, define the intermediate oracle T_{ij} as follows: to answer each of the first j queries posed by Adv, the oracle T_{ij} associates a random value to the unique node at level i that lies in the path from the root to the leaf containing the value requested by the query, and then it computes the requested value via $\ell(k) - i$ applications of the generator G (notice that up to now this is exactly the behavior born by the oracle T_i). Beginning with the $(j+1)^{st}$ query, the oracle T_{ij} starts behaving like T_j: to respond to a request for the value associated with a certain leaf, the oracle picks a random value and puts it inside the ancestor at level $i+1$ of the leaf at hand; afterwards, it computes (as usual) the desired value through $\ell(k) - i - 1$ calls to the PRG G. There is a last technicality to be added to completely specify the oracle T_{ij}: it acts consistently, i.e. if, while looking at the path from the root to the leaf associated to the value requested by Adv, the oracle T_{ij} finds out that an ancestor of that leaf has already been filed out (in answering a previous query), it uses that ancestor to compute the value of the leaf, without adding any new randomness to the tree.

Now this sequence is well-suited: it consists of $t+1$ intermediate oracles, and $T_i \equiv T_{i0}$, while $T_{i+1} \equiv T_{it}$. To complete this hybrid argument, it is left to prove that $\mathsf{Adv}^{T_{ij}} \approx \mathsf{Adv}^{T_{ij+1}}$. But this is of course the case, since the only difference between the two oracles is that one answered Adv's $(j+1)^{st}$ query putting a random value z at level i (and thus filling its left child l and its right child r with $G_0(z)$ and $G_1(z)$ respectively), while the other answered the same query putting two random values R_1 and R_2 respectively in l and r. If Adv behaves differently in the two cases, it would imply that Adv is able to distinguish between $G(z) \doteq G_0(z) \circ G_1(z)$ and $R_1 \circ R_2 \equiv R$, or, in other words, $G(z) \not\approx R$, contradicting the pseudorandomness of the generator G used in the GGM construction. It follows that $\mathsf{Adv}^{T_{ij}} \approx \mathsf{Adv}^{T_{ij+1}}$, for any j, which entails (by the hybrid argument) that $\mathsf{Adv}^{T_i} \approx \mathsf{Adv}^{T_{i+1}}$. From the arbitrariness of Adv, this holds true for any i, and finally (again by the hybrid argument):

$$\forall\ \mathsf{PPT}\ \mathsf{Adv} \qquad \mathsf{Adv}^{f_s} \approx \mathsf{Adv}^{F}$$

□

Remark 1 *This is by far the most intensive use of the hybrid argument we have seen: it is actually so intense that in the reduction we lose an important fraction of the advantage.*

This is because, from the proof of the validity of the hybrid argument (see Lecture 5), we know that if the advantage in distinguishing two intermediate distributions is ϵ, then the advantage in distinguishing between the two initial distributions can increase by a factor equal to the number of intermediate elements. Therefore, in our case, we are losing a factor of $t \cdot \ell(k)$ in comparison with the advantage in breaking the initial PRG.

5 More Efficient Construction under DDH

The GGM construction is very interesting: it features an original application of the hybrid argument and it demonstrates the use of complete binary trees to build complex primitives out of known, more basic ones. Anyway, the structure of the reduction is such that the construction loses both in efficiency and in security with respect to the underling "building block", namely the pseudo random generator G.

For these reasons, PRFs to be used in practice cannot be obtained in this way: a more concrete, number-theoretic construction is needed. Below we present one of the best-to-date practical (and yet provable!) construction, due to Naor-Reingold, which is based on the DDH assumption.

The construction works in the group $\mathbb{G} = QR_p$ of quadratic residues modulo p, where p is a strong prime (i.e. it is of the form $p = 2q + 1$, for some prime q.) In this setting, the DDH assumption can be stated as:

$$\langle g, g^a, g^b, g^{ab} \rangle \approx \langle g, g^a, g^b, g^c \rangle$$

where g is a random generator of $\mathbb{G}$ and a, b, c are chosen uniformly at random in $\mathbb{Z}_q$.

For a given choice of the (public) parameters p, q, g, the Naor-Reingold pseudo random function family is $\mathcal{NR} = \langle NR_{p,g,a_0,a_1,\ldots,a_\ell} \mid a_0, a_1, \ldots, a_k \leftarrow_R \mathbb{Z}_q \rangle_{\ell \in \mathbb{N}}$, where each function $NR_{p,g,a_0,a_1,\ldots,a_\ell} : \{0,1\}^\ell \to \mathbb{G}$ is defined as follows:[1]

$$NR_{p,g,a_0,a_1,\ldots,a_\ell}(x_1, \ldots, x_\ell) = (g^{a_0})^{\prod_{i \in S(x)} a_i} \qquad \text{where } S(x) = \{i \geq 1 \mid x_i = 1\}$$

In other words, the input $x = x_1, \ldots, x_\ell$ is considered bit by bit and, for each x_i equal to 1, the corresponding value a_i is included in the modular exponentiation. The advantage of such definition is that the value of the function on a particular point can be computed with $O(\ell)$ multiplications: we can compute the exponent $\alpha = a_0 \prod_{i \in S(x)} a_i \bmod q$ with at most ℓ multiplications, and then compute $g^\alpha \bmod p$ with at most 2ℓ multiplications (using the "Square & Multiply" algorithm.)

The *NR* construction for PRFs can be though as a particular instantiation of the complete binary tree technique seen in the GGM construction, where the PRG used is $G_{p,q,g,a}(g^b) \doteq G'_0(g^b) \circ G'_1(g^b) = \langle g^b, g^{ab} \rangle$. However, in some sense, the solution proposed by Naor Reingold also generalizes the previous construction, since a different exponent a_i is considered at each level of the tree, while in the standard GGM the same PRG is used at all levels. This is shown in figure 5.

[1] Notice, the output of the construction is a random element of $\mathbb{G}$. However, we already know a deterministic map that can turn it into a random element of $\mathbb{Z}_q$.

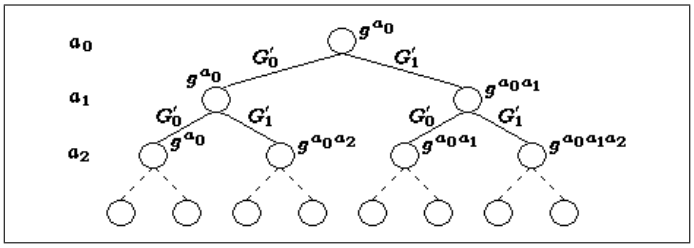

Figure 5: The complete binary tree in the NR construction.

The proof of security of this PRF family bears a close resemblance to the proof of the previous theorem; consequently, we state the result without explicitly including the supportive argument. We mention, however, that the actual security of the NR PRF is better that that of the GGM construction. In particular, the former "lost" a factor $t\ell$ in its security, where t is the qumber of PRF queries made by the attacker and ℓ is the input length. In contrast, a clever "random self-reducibility" argument from Lemma 1 allowed Naor and Reingold to lose "only" a factor ℓ in the security reduction. Namely, we only use the "outer" ℓ hybrids of the GGM construction (one per level of the tree). Inside each level i, we use Equation (1) (with value $a = a_i$) to directly show that giving oracle access to T_i is indistinguishable from giving oracle access to T_{i+1}, under the DDH assumption.

Theorem 2 (Naor Reingold)
Under the DDH *assumption, the family* $\mathcal{NR} = \langle NR_{p,g,a_0,a_1,\ldots,a_\ell} \mid a_0, a_1, \ldots, a_\ell \leftarrow_R \mathbb{Z}_q \rangle_{\ell \in \mathbb{N}}$ *is a* PRF *family with* $O(\ell)$ *security loss when reduced to DDH.*