Last Revised: 21 Jan 99.
Copyright ©1999 by Phillip Rogaway

ECS 227 — Modern Cryptography — Winter 99
Lecture 2

From Simple Primitives to Complex Ones: The PRG → PRF → PRP Constructions (Part I)

14 January 1999

1 Introduction and Leftover Business

In Lecture #1 we described the classical and the reductionist approaches to cryptography. We also described a couple of block ciphers (DES and RC6). At the very end we had just started to try to come up with a way to use a block cipher to encrypt. It is natural then that we should pick up today on this topic of symmetric encryption.

But in preparing today's lecture I felt bored of encryption and decided to put this off a couple of weeks. Today, instead, I want to gives some examples for how we start with "fundamental" objects and make more complex objects out of them. It is a lovely story. We'll use our reductionist approach: if the simple objects do their jobs then the more complex objects we construct will do their jobs, too.

Figuring out how (in principle) to accomplish elaborate cryptographic goals assuming only simple assumptions was considered by many to be the major goal of cryptography throughout most of the 1980's. It remains an important area. Our community has been highly successful, achieving many amazing goals under very weak cryptographic assumptions.

Anyway, since we are not going to do encryption for another lecture or two, you may postpone doing problem #4 from the homeworks set. Do it after we start discussing encryption.

To do problem number #3, you should know something which I meant to cover in the first lecture but didn't. It is a particular method for symmetric encryption: the *one-time pad.* Alice and Bob share an infinite (binary) random string a. Actually, why don't we regard them as sharing an infinite number of infinite random strings, $a_1, a_2, a_3 \ldots$. When Alice wants to send her i-th message, m_i, to Bob, Alice transmits the ciphertext $c_i = m_i \oplus a_i$. (Assume m_i is a binary string, too.) By "$\oplus$" we denote the exclusive-or operation, and it is understood that you are XORing m_i with the first $|m_i|$ bits of a_i.

We haven't defined what is a secure encryption scheme—but I claim that the above scheme *better* be considered secure! Because with a one-time pad, the ciphertext c_i is uncorrelated to the plaintext m_i, apart from $|m_i|$. In particular, for all equal-length m_i and m_i', the probability that Alice's i-th ciphertext is c_i is the same given that the underlying message is m_i or given that it is m_i'.

To argue that this is so let us look at a single bit of c_i. Everything that is done in this encryption method is done bitwise, so there is no loss of generality to just focus on one bit. So now Alice is trying to send Bob a bit m, and what she transmits is $c = m \oplus a$, where a is a random secret bit shared between Alice and Bob. I claim that

$$\Pr[c = 0|m = 0] = \Pr[c = 0|m = 1]$$

and

$$\Pr[c = 1|m = 0] = \Pr[1 = 0|m = 1].$$

The probabilities are over the random choices of a. To see these simple equalities, note that $\Pr[c = 0|m = 0] = 1/2$ since, given that $m = 0$, the event $c = 0$ happens iff $a = 0$, and a was chosen at random. And $\Pr[c = 0|m = 1] = 1/2$ since, given that $m = 1$, the event $c = 1$ happens iff $a = 1$. So these two probabilities are equal and a half, so the next two expressions are equal, too. Whatever secure encryption might be, showing that the ciphertext is independent of the plaintext (apart from its length) is certainly enough.

On homework problem #3, then, when you want to show that Alice can send Bob a 48-bit message m, show the same thing: that the ciphertext c Alice sends is independent of the 48-bit message m.

When you want to show that Alice can't send a 49-bit string, describe some piece of information (other than existence and length) that an adversary can learn about m given c. That is to convince that the scheme is no good.

Now let us move on.

2 One-Way Functions

The traditional starting point for reductionist cryptography is a *one-way function* (OWF). Intuitively, a one-way function is a function $f : A \to B$ which is easy to compute in one direction (the "forward direction") but hard to compute in the other direction (the "reverse direction").

There are "physical" one-way functions, like breaking a glass: it is easier to break a glass than to reassemble the resulting pieces. Here's a picture for a (mathematical) OWF:

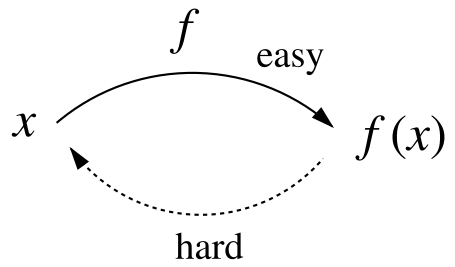

Figure 1: A One-Way Function.

We should refine our intuition before trying to make a definition. What does it mean that an adversary can't go backwards from $f(x)$ to x? Certainly the adversary *can* go backwards—sometimes. If the adversary chooses x, then applies the OWF f to x to get y, then it is trivial for the adversary to go backwards from y: she knows x.

Somehow we need to capture that it is hard to go backwards for "most" y. We could imagine choosing a random y in B and asking the adversary to find a preimage for x. Actually, it turns out more convenient to choose a random x in A (assume A is finite) and then ask the adversary to find a preimage for $y = f(x)$.

Notice that I speak in terms of finding a preimage for $y = f(x)$; I don't speak in terms of finding x. We don't want to assume that f is injective (one-to-one). If $f(x) = y$ and $f(x') = y$, for $x \neq x'$, it doesn't really seem fair that we choose $x \in A$ at random, give the adversary $y = f(x)$, and then expect the adversary to determine x—we haven't given her any information which would allow her to distinguish the preimage x from the preimage x'. We should give the adversary credit if she comes up with either x or x'—any preimage of y. Let $f^{-1}(y)$ denote the set of pre-images under f of y.

To make security quantitative we will imagine an experiment. We have an inverting algorithm (our adversary) I. We will give the adversary $y = f(x)$, chosen by selecting a random $x \in A$. The adversary's goal in life is to find an $x' \in f^{-1}(y)$. If the adversary does this (on some particular run) we say that she has succeeded (on this run). A good adversary is one that succeeds often, and doesn't take an enormous amount of time to do so. Here then is the quantitative definition.

Definition 2.1 [OWF—Concrete definition.] Let $f : A \to B$ be a function, with A a finite set. Let I be an adversary. Define

$$\mathrm{Adv}_f^{\mathrm{owf}}(I) \quad = \quad \Pr[x \leftarrow A\,; y \leftarrow f(x) : I(y) \in f^{-1}(y)]\;. \qquad \blacksquare$$

The probability is over what's explicit in the experiment (choosing a random x in A) as well as I's internal coin tosses (if I is probabilistic).

Asymptotic approach. The above represents the "concrete" approach to defining security; it is not the "traditional" approach, which is asymptotic. You will find that most of the literature uses an asymptotic approach, so I want to introduce that to you. The concrete approach has been practiced and advocated principally by me and Mihir Bellare, but nowadays more people are doing the same.

In the asymptotic approach there is an underlying *security parameter*, k, which quantifies "how secure" is the underlying cryptosystem. We look at security as a function of k, and try to capture that it is a "good" function of k. How exactly the security parameter is brought into the picture varies a bit from problem to problem, but it is usually very simple.

In defining a OWF the simplest way to bring in a security parameter is to make the domain of f all of $A = \{0,1\}^*$ and take the length of the input to f as to be the security parameter. Now we have what amounts to a family of functions: $f = \{f_k\}$, where $f_k : \{0,1\}^k \to \{0,1\}^*$. We look at efficacy of adversaries, as a function of k, as they try to invert f_k. Here is the definition:

Definition 2.2 [OWF—Asymptotic definition.] Let $f : \{0,1\}^* \to \{0,1\}^*$ be function. Let I be an adversary. Define

$$\mathrm{Adv}_{f,I}^{\mathrm{owf}}(k) \quad = \quad \Pr[x \leftarrow \{0,1\}^k\,; y \leftarrow f(x) : I(1^k, y) \in f^{-1}(y)]\;.$$

We say that f is a *one-way function* (with respect to uniform adversaries) if for any polynomial-time adversary I, $\mathrm{Adv}_{f,I}^{\mathrm{owf}}(k)$ is negligible. $\blacksquare$

Let me define *negligible*:

Definition 2.3 [Negligible.] Let $\epsilon : \mathsf{N} \to \mathsf{R}^+$ be function. We say that $\epsilon(k)$ is *negligible* if for any $c > 0$ there exists a K such that for all $k \geq K$, $\epsilon(k) \leq k^{-c}$. $\blacksquare$

That is, a negligible function is one which vanishes faster than the inverse of any polynomial. The traditional approach in cryptography is to say that you've accomplished your goal when polynomial-time adversaries necessarily have negligible advantage.

The argument 1^k in definition 2.2 is just a trick so that when I talk of I being negligible it is clear that I mean negligible in k: we assume that I being "polynomial time" means "polynomial time in the length of the first argument to I" and the length of this first argument is k.

CONCRETE SECURITY EXPRESSED IN TERMS OF ADVERSARIAL RESOURCES. The asymptotic approach makes for more convenient discourse. A function f is either a one-way function or not. The quantitative value of $\mathrm{Adv}_{f,I}(k)$ is normally ignored, apart from this function being negligible. In the concrete setting we don't have a notion of a "one-way function"—or rather, every function $f : A \to B$ should be considered a OWF. We have the real number $\mathrm{Adv}_f^{\mathrm{owf}}(I)$ to measure the quality of f as a OWF. Even the identity function would be a OWF—albeit one with lousy security, because a very efficient adversary can always find a preimage to $y = f(x)$.

Our intuition, remember, is that a OWF is good if "reasonable" adversaries can't achieve "good" advantage. The asymptotic approach normally defines "reasonable" as polynomial-time in the security parameter, and defines "good" as not negligible. While the concrete approach avoids these absolute boundaries, it is sometimes convenient to look at how the advantage depends on resources. Here is how we would do this for a OWF:

Definition 2.4 [OWF—Resource-parameterized definition.] Let $f : A \to B$ be function, with A a finite set. Define

$$\mathrm{Adv}_f^{\mathrm{owf}}(t) \quad = \quad \max_I \ \{\Pr[x \leftarrow A\,;\, y \leftarrow f(x) : I(y) \in f^{-1}(y)]\} \qquad \blacksquare$$

where the maximum is over all adversaries I which run in time $\leq t$.

Forgive the overloading of the function $\mathrm{Adv}_f^{\mathrm{owf}}$: with an argument which is an adversary I, $\mathrm{Adv}_f^{\mathrm{owf}}(I)$ is the number which is that adversary's advantage; with an argument that is a number t, $\mathrm{Adv}_f^{\mathrm{owf}}(t)$ is the number which is an adversary's best possible advantage if she uses that amount of time. (I used to always denote the second type of function "Sec" instead of "Adv", but I am always changing these things. Can you think of why I no longer like the word "security" (and symbol "Sec") for the second measure?

We will sometimes call a function f a OWF if, we believe, $\mathrm{Adv}_f^{\mathrm{owf}}(t)$ is a desirable function—a function which stays near 0 until t becomes very large. For example, if we think that $\mathrm{Adv}_f^{\mathrm{owf}}(t) < 2^{-40}$ as long as $t < 2^{80}$, then that would be a good OWF.

In the Definition 2.4 you must fix an underlying model of computation and a convention for measuring time in that model; otherwise, it makes no sense to speak of maximizing over all adversaries which run in time t. One usually imagines some fixed RAM (random access machine) as the model of computation.

One should also deal with the fact that if I is a huge program then its time could be short. For example, imagine I having an internal table of $|B|$ elements. Then I can just look up the value of $y = f(x)$ to find x. To get around this we normally define the run time of I as the actual run time of I plus the length of the description of I.

Alternatively, one can demand that all adversaries I have some maximal size description, like 1 GByte. No cryptographer has ever dreamed up an algorithm which takes more than 1 GByte to describe. This again avoids the adversary putting some enormous table into the her descriptions.

EXAMPLES OF OWFS. We have now defined OWFs but not given any examples of them. Let us give a couple of examples, building on what we did in the last lecture.

1. Last class we defined the function DES : $\{0,1\}^{56} \times \{0,1\}^{64} \to \{0,1\}^{64}$. We can make a OWF from this by setting $f(x) = \mathrm{DES}_x(0)$, where 0 is the 64-bit string which is the binary representation of the number 0. Slightly better is $f(x) = \mathrm{DES}_x(0) \parallel \mathrm{DES}_x(1)$. We think that f is a OWF. In fact, if DES is a good block cipher (to be described later today) than f is guaranteed to be a good OWF. This is a result by Luby and Rackoff, recently improved by Dave Wagner.

2. Multiplication is a good one way function. Let's take the domain of f, A, to be the set of pairs of 100-digit primes. Then $f((x_1, x_2)) = x_1 x_2$. It is easy to compute $y = f((x_1, x_2))$, but finding a preimage for y amounts to factoring a 200-digit number which is the product of random 100-digit primes. Nobody knows how to do this.

Note that in the second example, it is multiplication (not factoring) which is the one-way function. One often hears careless discourse in this connection, people speaking of factoring as a one-way function. They mean multiplication.

3 Pseudorandom Generators

In the first lecture we defined PRGs. Let us review that definition.

Remember the intuition. Let $g : \{0,1\}^\ell \to \{0,1\}^L$ be a function. Assume $L > \ell$. We give an adversary D one of two types of strings. The first type of string a pseudorandom one: $y = g(x)$ for a random $x \leftarrow \{0,1\}^\ell$. The second type of string is a truly random one: $y \leftarrow \{0,1\}^L$. The adversary's job is to try to tell which type of string she has.

We made the above intuition into a definition of security as follows :

Definition 3.1 [PRG.] Let $g : \{0,1\}^\ell \to \{0,1\}^L$ be a function, with $L \geq \ell$. Define

$$\mathrm{Adv}_g^{\mathrm{prg}}(D) \quad = \quad \Pr[x \leftarrow \{0,1\}^\ell \,;\, y \leftarrow g(x) : D(y) = 1] - \Pr[y \leftarrow \{0,1\}^L : D(y) = 1] \,. \qquad \blacksquare$$

Interpret D answering "1" as an indication that D thinks that y is pseudorandom. Interpret D answering "0" as an indication that D thinks that y is truly random. An advantage of 0 is terrible—the adversary can get this advantage by always answering "1" or by flipping a coin. An advantage of 1 is great—the adversary D is always getting it right. (Actually, this is impossible, but an adversary might come close.) What about an advantage of around -1? In some sense this is terrible—the adversary is almost always wrong. In another sense this great, for if you had an adversary which had advantage of nearly -1 then all you would have to do is to flip when the adversary says 1 and when she says 0, and you'd get an adversary that was almost always right. For this reason it is the absolute value of the advantage, as defined above, that people sometimes take as the definition in contexts like this. That is, the construction is as "tight" as one can hope for.

Definition 3.2 [PRG—Alternative definition.] Let $g : \{0,1\}^\ell \to \{0,1\}^L$ be a function, with $L \geq \ell$. Define

$$\mathrm{Adv}_g^{\mathrm{prg}}(D) \quad = \quad |\ \Pr[x \leftarrow \{0,1\}^\ell \,;\, y \leftarrow g(x) : D(y) = 1] - \Pr[y \leftarrow \{0,1\}^L : D(y) = 1]\ | \,. \qquad \blacksquare$$

Let's stick with Definition 3.1. Definition 3.2 is like getting full credit on a True/False test if you manage to get every question wrong—the reasoning being that you could have gotten every question right by just flipping your answers. In fact, I have never graded that way. Call me cruel.

As before, we can make a resource-oriented version of our definition as follows:

Definition 3.3 [PRG—Resource version.] Let $g : \{0,1\}^\ell \to \{0,1\}^L$ be a function, with $L \geq \ell$. Define

$$\mathrm{Adv}_g^{\mathrm{prg}}(t) \;=\; \max_D \;\{\mathrm{Adv}_g^{\mathrm{prg}}(D)\}$$

where the maximum is over all adversaries that run in time at most t. ▮

The quantity $L - \ell$ is called the *stretch* of a PRG. The value x in $g(x)$ is called the *seed* of the PRG g.

4 Pseudorandom Functions

Having a pseudorandom function (PRF) f is like having a random function: an adversary, given an oracle for $h = f(\cdot)$, shouldn't be able to tell if $h = f$ is a pseudorandom function or if $h = \rho(\cdot)$ is a truly random function.

Well—sort of. If we were to regard $f : \{0,1\}^* \to \{0,1\}^*$ itself as the PRF then the above is pretty screwy. For certainly an adversary can determine well if a fixed function h, given as an oracle, behaves like a truly random function or else like some fixed function f. Just ask 0, say, where 0 is some fixed string in the domain of f. Then have the adversary test if the return value is $f(0)$. If so, the oracle is almost certainly one for f. That is, since we always assume that the adversary knows the description of our cryptosystem, she knows any fixed function f we would like to use, and so she knows if an oracle we give her is behaving like that.

To get around this we will think in terms of families of functions. Syntactically, we'll make a pseudorandom function a map $F : K \times A \to B$, where $K, A, B \subseteq \{0,1\}^*$. For simplicity, let's assume $K = \{0,1\}^\alpha$ and $A = B = \{0,1\}^\beta$. We write $F_a(x)$ instead of $F(a,x)$. We want that the adversary can't tell $F_a(\cdot)$ from a random function *when a is random and unknown.*

We imagine two games that we could play against the adversary. In the first game we choose a random $a \in \{0,1\}^\alpha$ and give the adversary an oracle for $F_a(\cdot)$. This is the case of the adversary's oracle h being a pseudorandom function. In the second game we chose a random $\rho \in \mathrm{Rand}_{\beta\to\beta}$ and give the adversary an oracle for $\rho(\cdot)$. This is the case of the adversary's oracle h being a truly random function. By $\mathrm{Rand}_{\beta\to\beta}$ we mean the set of all function from β bits to β bits. We are choosing one of these functions at random.

If F is a good PRF then the adversary shouldn't be able to tell what type of oracle she has: in other words, what the adversary does when $h = F_a$, for a random a, should be just about the same as what the adversary does when $h = \rho$, for a random function ρ. We make this quantitative as follows:

Definition 4.1 [PRF.] Let $F : \{0,1\}^\alpha \times \{0,1\}^\beta \to \{0,1\}^\beta$ be a function and let D be an adversary. Then

$$\mathrm{Adv}_F^{\mathrm{prf}}(D) \;=\; \Pr[a \leftarrow \{0,1\}^\alpha : D^{F_a(\cdot)} = 1] - \Pr[\rho \leftarrow \mathrm{Rand}_{\beta\to\beta} : D^{\rho(\cdot)} = 1]$$ ▮

In passing to the resource-version of this notion we should pay attention to two resources: the amount of time the adversary runs in, and also the number of queries that the adversary makes. In applications the latter might be under the security architect's control, while the former certainly isn't. We might also think of the total length of the adversary's questions, but since we have assumed that the domain is strings of fixed length, there is no point to do this just now.

Definition 4.2 [PRF—Resource version.] Let $F : \{0,1\}^\alpha \times \{0,1\}^\beta \to \{0,1\}^\beta$ be a function and let t, q be numbers. Then

$$\mathrm{Adv}_F^{\mathrm{prf}}(t,q) \quad = \quad \max \ \{\mathrm{Adv}_F^{\mathrm{prf}}(D)\}$$

where the maximum is over all adversaries that run in time at most t and ask at most q oracle queries. ▮

5 From PRF to PRG: Construction and Analysis

Assuming you have a random string to use as the seed for your PRG $g : \{0,1\}^\ell \to \{0,1\}^L$, and assuming L is large, having a PRG gives one access to a long string which is almost as good as an equally-long random string. But you can only access that string sequentially. Suppose you would like to know the 1,000,000-th bit of $g(x)$. Then you probably have to compute that starting from the beginning. Getting the i-th bit probably takes time proportional to i.

A pseudorandom function is more general. It's like having random access to all the bits of an extremely long L-bit string $g(x)$. That extremely long L-bit string is

$$F_a(0) \parallel F_a(1) \parallel \cdots \parallel F_a(2^\beta - 1)\ .$$

We can easily get at anywhere we like in this string.

The above is one of our first and simplest constructions: we have made a PRG g_F from a PRF F. Think of this construction as parameterized by a number L, where $\alpha < L \le \beta 2^\beta$. Given a PRF $F : \{0,1\}^\alpha \times \{0,1\}^\beta \to \{0,1\}^\beta$ we construct a PRG $g_F : \{0,1\}^\alpha \to \{0,1\}^L$ by setting

$$g_F(x) = (F_x(0) \parallel F_x(1) \parallel \cdots F_x(M-1))[1..L]$$

where $M = \lceil L/\beta \rceil$. I'd like now to analyze this construction. It is easy if you keep your head on about what we must show.

We want to show that if F is a good PRF then g_F is a good PRG. In other words, if g_F is a bad PRG then F is a bad PRF. In other words, if some adversary D_g does a good job to break the PRG g_F then there is an adversary D_F that does a good job to break the PRF F. We want to look at how effective is D_F compared to D_g. In fact, D_F will have the same advantage and run in almost the same amount of time.

Theorem 5.1 Let $F : \{0,1\}^\alpha \times \{0,1\}^\beta \to \{0,1\}^\beta$ be a PRF, let L be a number where $\alpha < L \le \beta 2^\beta$, and define the PRG $g_F : \{0,1\}^\alpha \to \{0,1\}^L$ as above. Let D_g be an adversary for attacking the PRG g_F. Suppose D_g runs in time t and achieves advantage $\delta = \mathrm{Adv}_{g_F}^{\mathrm{prg}}(D)$. Then there is an adversary D_F for attacking the PRF F that runs in time t', asks q' queries, and achieves advantage δ', where

$$\begin{aligned} q' &= \lceil L/\beta \rceil, \\ t' &= t + cL, \text{ and} \\ \delta' &= \delta \end{aligned}$$

for some absolute constant c.

Proof. We must construct the adversary D_F. Recall that D_F has an oracle, h. We define D_F^h as follows:

Algorithm D_F^h:
Let $M = \lceil L/\beta \rceil$.
Let $y = (h(0) \parallel h(1) \parallel \cdots \parallel h(M-1))[1..L]$.
Run $D_g(y)$
When D_g outputs a bit, b, have D_F output the exact same bit, b

We note first that D_F asks q' oracle queries, as required. How long does D_F take to run? The main thing is the running of D_g, which takes time t. But there is also a little bit of overhead. That main overhead is in making the M oracle queries and forming a string y out of the responses. Since we assume that an oracle call takes unit time, the overhead is $O(L)$.

Now we have to check that $\delta' = \delta$. There are two things to think about: what happens when $h = F_a$, for a random key a, and what happens when $h = \rho$, for a random function ρ. In the former case, the environment which D_F provides D_g is identical to the one in which D_g is presented with a pseudorandom string:

$$\Pr[a \leftarrow \{0,1\}^\alpha : D_F^{F_a(\cdot)} = 1] \quad = \quad \Pr[x \leftarrow \{0,1\}^\alpha : D_g(g(x)) = 1] \ .$$

In the latter case, the environment which D_F provides D_g is identical to the one in which D_g is presented with a uniformly random string, since the first L bits of $\rho(0) \parallel \rho(1) \parallel \cdots$, when ρ is a random function, is a random L-bit string:

$$\Pr[\rho \leftarrow \text{Rand}_{\beta\to\beta} : D_F^{\rho(\cdot)} = 1] \quad = \quad \Pr[y \leftarrow \{0,1\}^L : D_g(y)) = 1] \ .$$

Thus

$$\begin{aligned} \text{Adv}_F^{\text{prf}}(D_F) &= \Pr[a \leftarrow \{0,1\}^\alpha : D_F^{F_a(\cdot)} = 1] - \Pr[\rho \leftarrow \text{Rand}_{\beta\to\beta} : D_F^{\rho(\cdot)} = 1] \\ &= \Pr[x \leftarrow \{0,1\}^\alpha : D_g(g(x)) = 1] - \Pr[y \leftarrow \{0,1\}^L : D_g(y)) = 1] \\ &= \delta \end{aligned}$$

and we are done ▌

6 From PRG to PRF: Construction

Now let's do something a little bit harder: we want to make a PRF from PRG.

Let's first assume that we have a PRG g which goes from α bits to 2α bits. Let's say we want to construct a PRF $F : \{0,1\}^\alpha \times \{0,1\}^\alpha \to \{0,1\}^\alpha$. We make a tree. This sort of tree is very common in cryptography.

How do we compute $F_a(x)$? Do the following thought experiment. Make a binary tree and put a at the root. Label this node $a_\varepsilon = a$. Now let a_s be some arbitrary node in the tree, where $|s| < \alpha$. Then the left child for a_s is

$$a_{s0} = g(x)[1..\alpha]$$

while the right child for a_s is

$$a_{s1} = g(x)[\alpha+1..2\alpha] \ .$$

See Figure 2 for a picture.

Don't get lost in the notation; the idea is simple. You start with an α-bit secret—the name of the particular function in the family—as the root of your tree. You stretch it to 2α bits. You'll either use the first α bits or the next α bits—which one depends on the first bit of the index x. Keep doing this, using the bits of the index x to spell out a path in the tree. Output the value at the leaf.

This construction is due to [GoGoMi86]. We will analyze it next time.

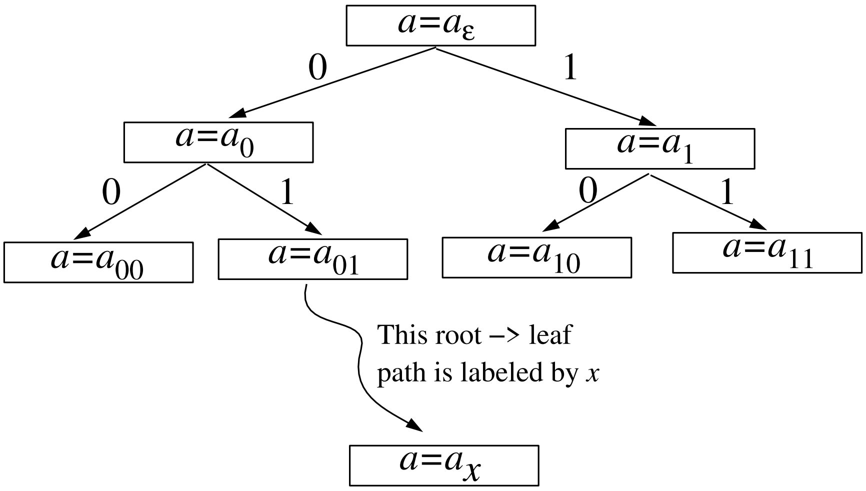

Figure 2: Making a PRF from a PRG. The left and right children of a node a_s are the first α bits of $g(a_s)$ and the remaining α bits of $g(a_s)$. Denote these a_{s0} and a_{s1}, respectively. Then Define $F_a(x) = a_x$.

7 From an $\alpha \to \alpha + 1$ PRG to an $\alpha \to 2\alpha$ PRG

Before we can apply the GGM-tree construction we need to have a PRG which stretches α bits to 2α bits. I would like to show how to construct such a PRG from any PRG. In other words, we have a PRG which stretches its input by just a little, let's say 1 bit. We want a PRG with more reasonable stretch, say one which doubles the length of the input.

Suppose $g : \{0,1\}^\alpha \to \{0,1\}^{\alpha+1}$. Lets define

$$g_i(x) \quad = \quad \begin{cases} \varepsilon & \text{if } i = 0, \text{ and} \\ g(x)[1] \parallel g_{i-1}(g(x)[2..\alpha+1]) & \text{otherwise.} \end{cases}$$

That is, $g_i(x)$ gives us i bits by applying g i times and continually peeling off one bit and using the remaining α bits for the next iteration of x. Then we define

$$g^*(x) \quad = \quad g_{2\alpha}(x)$$

as the PRG we have constructed out of g. The construction is depicted in Figure 3

We want to show that if $g : \{0,1\}^\alpha \to \{0,1\}^{\alpha+1}$ is a good PRG then $g^* : \{0,1\}^\alpha \to \{0,1\}^{2\alpha}$ is a good PRG. In other words, an adversary $D_{2\alpha}$ that breaks g^* should give rise to an adversary $D_{\alpha+1}$ that breaks g. This time it is not so clear how to construct our adversary. We will use what is called a *hybrid argument*. You will be seeing more of these. It is a technique you should try to master.

Here is the theorem statement. When you do these, you first do an "informal" theorem, then you do the proof, and then you extract out a formal theorem statement. Let us do that:

Theorem 7.1 [Stretching lemma—Informal statement.] Let $g : \{0,1\}^\alpha \to \{0,1\}^{\alpha+1}$ be a function. Using g, define $g^* : \{0,1\}^\alpha \to \{0,1\}^{2\alpha}$ as above. Let $D_{2\alpha}$ be an adversary that breaks g^*. Then there is an adversary $D_{\alpha+1}$ that breaks g.

What makes this an informal statement is that "breaking" g and g^* aren't defined; all we have is $\mathrm{Adv}_g^{\mathrm{prg}}(D_{\alpha+1})$ and how it might related to $\mathrm{Adv}_{g^*}^{\mathrm{prg}}(D_{2\alpha})$. The formal theorem statement will spell out the relationship.

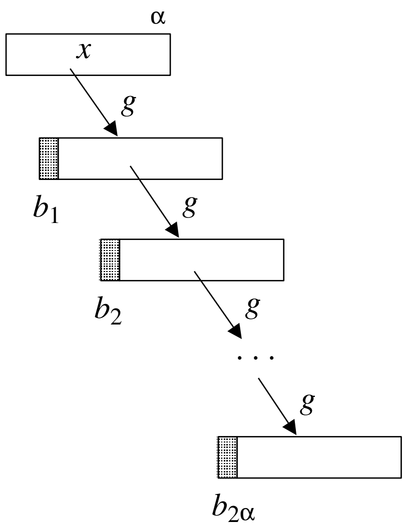

Figure 3: Making an α-to-2α PRG out of a α-to-$\alpha+1$ PRG.

We proceed with the proof.

Proof. Let $D_{2\alpha}$ be an adversary which attacks g^*. The adversary takes an input, $y_{2\alpha}$, and attempts to answer if $y_{2\alpha}$ is random or g^*-pseudorandom. We wish to construct an adversary $D_{\alpha+1}$ which takes an input $y_{\alpha+1}$ and attempts to answer if $y_{\alpha+1}$ is random or g-pseudorandom. Adversary $D_{\alpha+1}$ will work as follows. First, let us parameterize this algorithm by some number i, where $0 \le i \le 2\alpha$. We'll figure out later what to do for this value i.

Algorithm $D^i_{\alpha+1}(y_{\alpha+1})$:
$y_{2\alpha} \leftarrow \$^i \parallel y_{\alpha+1}[1] \parallel g_{2\alpha-i-1}(y_{\alpha+1}[2..\alpha+1])$
Run $D_{2\alpha}(y_{2\alpha})$
When $D_{2\alpha}$ outputs a bit, b, have $D_{\alpha+1}$ output the exact same bit, β

For the first line, omit mention of $y_{\alpha+1}[1]$ and $g_{2\alpha-i-1}(y_{\alpha+1}[2..\alpha+1])$ when $i = 2\alpha$. The notation $y_{2\alpha} \leftarrow \$^i \parallel \cdots$ means to choose a random string of length i.

For $0 \le i \le 2\alpha$, define

$$p_i \quad = \quad \Pr[x_\alpha \leftarrow \{0,1\}^\alpha \, ; y_{\alpha+1} = g(x_\alpha) : D^i_{\alpha+1}(y_{\alpha+1}) = 1]$$

as the probability that the algorithm above answers 1 when you feed it a pseudorandom string. By our definition of the behavior of $D^i_{\alpha+1}$,

$$p_{2\alpha} \quad = \quad \Pr[y_{2\alpha} \leftarrow \{0,1\}^{2\alpha} : D_{2\alpha}(y_{2\alpha}) = 1]$$

while

$$p_0 \quad = \quad \Pr[x_\alpha \leftarrow \{0,1\}^\alpha \, ; y_{2\alpha} = g^*(x_\alpha) : D_{2\alpha}(y_{2\alpha}) = 1] \, .$$

We are assuming that $p_0 - p_{2\alpha} = \delta$ is "significant"—the algorithm $D_{2\alpha}$ does its job well. By the triangle inequality we can conclude that there exists an i, $0 \le i < 2\alpha$, such that $p_i - p_{i+1} \ge \frac{\delta}{2\alpha}$. We can imagine that we know what i to use; think of it as hardwired into the program for $D_{\alpha+1} = D^i_{\alpha+1}$.

Now observe the following: if algorithm $D^i_{\alpha+1}$ is provided a pseudorandom y_{i+1} then, by definition, the probability that $D^i_{\alpha+1}$ outputs 1 is p_i; but if algorithm $D^i_{\alpha+1}$ is provided a truly random y_{i+1} then the probability that $D^i_{\alpha+1}$ outputs 1 is p_{i+1}. That is,

$$p_i \quad = \quad \Pr[x_\alpha \leftarrow \{0,1\}^\alpha \,;\, y_{\alpha+1} = g(x_\alpha) : D^i_{\alpha+1}(y_{\alpha+1}) = 1]$$

while

$$p_{i+1} \quad = \quad \Pr[y_{\alpha+1} \leftarrow \{0,1\}^{\alpha+1} : D^i_{\alpha+1}(y_{\alpha+1}) = 1]\;.$$

We know that

$$\begin{aligned} p_i - p_{i-1} &= \mathrm{Adv}^{\mathrm{prg}}_g(D^i_{\alpha+1}) \\ &\geq \frac{\delta}{2\alpha} \end{aligned}$$

so we are done. ▮

With the proof in hand, we can formalize our theorem statement:

The "non-uniformity" in the above proof —that we did not know the value of i— can be easily eliminated. Not only is there a "good" i but the average value of $p_i - p_{i+1}$ must also be at least $\frac{\delta}{2\alpha}$. So we can make our algorithm $D_{\alpha+1}$ begin by choosing a random i and then behaving according to $D^i_{\alpha+1}$. The algorithm is then

Algorithm $D_{\alpha+1}(y_{\alpha+1})$:
$i \leftarrow \{0, \ldots, 2\alpha\}$ $\qquad y_{2\alpha} \leftarrow \$^i \parallel y_{\alpha+1}[1] \parallel g_{2\alpha-i-1}(y_{\alpha+1}[2..\alpha+1])$
Run $D_{2\alpha}(y_{2\alpha})$
When $D_{2\alpha}$ outputs a bit, b, have $D_{\alpha+1}$ output the exact same bit, β.

Theorem 7.2 [Stretching lemma.] Let $g : \{0,1\}^\alpha \to \{0,1\}^{\alpha+1}$. Using g, define $g^* : \{0,1\}^\alpha \to \{0,1\}^{2\alpha}$ as above. Let $D_{2\alpha}$ be an adversary that breaks g^*. Suppose that $D_{2\alpha}$ runs in time t and achieves advantage $\delta = \mathrm{Adv}^{\mathrm{prg}}_{g^*}(D_{2\alpha})$. Then there is an adversary $D_{\alpha+1}$ that attacks g and runs in time t' and achieves advantage δ', where

$$\begin{aligned} t' &= t + 2\alpha \mathrm{Time}_g + c\alpha \text{ and} \\ \delta' &= \frac{\delta}{2\alpha} \end{aligned}$$

where Time_g is the amount of time needed to compute g and c is an absolute constant. ▮

References

Approach. The concrete approach to security we have been illustrating springs from a body of work by Bellare and Rogaway; see, for example:

[BeKiRo94] "The Security of the Cipher Block Chaining message Authentication Code," Mihir Bellare, Joe Kilian and Phillip Rogaway, Advances in Cryptology—CRYPTO '94, Lecture Notes in Computer Science, Vol. 839, Springer-Verlag, 1994.

[BeGuRo95] "XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions," Mihir Bellare, Roch Guerin and Phillip Rogaway Advances in Cryptology—CRYPTO '95, Lecture Notes in Computer Science, Vol. 963, Springer-Verlag, 1995.

Stretching lemma. This arises out of [GoGoMi86] and earlier work of Yao. The first writeup is in:

[BoHi89] "Pseudo-random Generators and Complexity Classes," Ravi Boppana and Ray Hirschfeld, *Advances in Computer Research*, vol. 5, edited by Silvio Micali, JAI Press, 1989, pp. 1-26.

PRFs from PRGs. This is due to:

[GoGoMi86] "How to Construct Random Functions," Oded Goldreich, Shafi Goldwasser and Silvio Micali, *Journal of the ACM*, Vol. 33, No. 4, 1986, pp. 792–807.