6.875J/18.425J Cryptography and Cryptanalysis Mar 1, 2017

Lecture 6: Pseudorandom Generators from One Way Functions

Instructor: Shafi Goldwasser *Scribes: Srinivasan Raghuraman*

Acknowledgements: These lecture notes are based on scribes by Bingfei Cao, Bristy Sikder, Sagnik Saha, Andrew He, Gweneth McKinley, Yuancheng Yu, Omer Cerrahoglu, Daniel Epelbaum and Daniel Shaar from 2016 (lecturer: Shafi Goldwasser), Elizabeth Denys from 2011 and Prasant Gopal from 2010 (lecturer: Silvio Micali).

In the last lecture, we introduced the notion of pseudorandom generators and provided two definitions that turned out to be equivalent. In this lecture, we describe how one could construct such an object and describe some of its applications. We also describe a more powerful object called pseudorandom functions.

1 Construction of PRGs

We describe a construction of PRGs from one way permutations. Later, we discuss discuss why we need a permutation as opposed to any one way function. Here is a very natural approach to try and construct a PRG, $G : \{0,1\}^n \to \{0,1\}^m$, from any one way function f. Suppose we define

$$G(x) = f^m(x)||f^{m-1}(x)||\dots||f(x)$$

The intuition is clear – for any $k \geq 1$, given $f^k(x)$, computing $f^{k-1}(x)$ is hard since that would require inverting the one way function. Hence given one block of the output of G, namely $f^k(x)$, computing the next block of the output of G, namely is $f^{k-1}(x)$ is hard. However, this does not take us all the way because while computing the entire block might be hard, computing a single bit of it might not. In fact, we have seen examples of one way functions like the discrete logarithm function whose LSB is easy to guess.

What if we extract a bit from that block that we know is actually hard to predict? We now go back to hard-core predicates for one-way permutations. Recall that if B is a hard core predicate for a one way permutation f, then given $f(x)$, it is hard to predict $B(x)$. This implies that $f(x)$ hides $B(x)$. Thus, we now construct cryptographically secure PRGs using hard-core bits as follows. Let B be a hard-core predicate for the one way permutation f. Our PRG is defined as

$$G(x) = B(f^{m-1}(x))||B(f^{m-2}(x))||..||B(f^2(x))||B(f(x))||B(x)$$

We claim that G is a PRG. In order to prove this claim, as we have seen last lecture, it is sufficient to show that G passes all next-bit tests, that is, given any prefix of the output of G, predicting the next bit is hard. We prove this below.

Remark. We now note why we considered one way permutations as opposed to one way functions. For one way functions, the range of $f^k(x)$ might become very small and might differ greatly from the range of $f(x)$. Intuitively, this means that the longer our pseudorandom string is, the higher the chance of a certain group of strings repeating. This results in the string being more predictable than random. To avoid this problem, we will only consider one-way permutations f. This ensures that the distributions of x and $f^k(x)$ are the same.

Theorem 1. *If there exists a one-way permutation, then there exists a PRG for all polynomial stretches.*

We can construct a cryptographically-strong pseudorandom generator from n to m bits (for m polynomial in n) as follows: given a one-way permutation f with a hard-core predicate B, then define

$$G(x) = B(f^{(m-1)}(x))\|B(f^{(m-2)}(x))\|\cdots\|B(f(x))\|B(x)$$

where $f^{(i)}(x)$ represents f applied i times on x.

	Internal	
Input	**Configuration**	**Output**
$x \longrightarrow$	$f(x)$	$B(f^{(m-1)}(x))$
	$f^{(2)}(x)$	$B(f^{(m-2)}(x))$
	$f^{(3)}(x)$	$B(f^{(m-3)}(x))$
	.	.
	.	.
	.	.
	$f^{(m)}(x)$	$B(x)$

We can even make $f^{(m)}(x)$ public, but not any other internal state $f^{(k)}(x)$ for $k < m$.

Proof of Theorem. We prove the theorem by contradiction. If G is not a cryptographically secure PRG, then as shown in the previous lecture, it must fail a next bit text. In other words, there exist an index $1 \leq j \leq m$ and a PPT algorithm P such that

$$\Pr\left[P(b_1,\ldots,b_{j-1}) = b_j \text{ for } (b_1,\ldots,b_m) = G(\mathcal{U}_n)\right] > \frac{1}{2} + \epsilon$$

for some non-negligible ϵ. We will construct a PPT prediction algorithm PRED, that can guess $B(x)$ from $f(x)$ with non-negligible probability. This will contradict our assumption that B is hard core for f.

We define PRED as follows. Given input $f(x)$:

1. Compute $f^{(2)}(x), f^{(3)}(x)), \ldots, f^{(j-1)}(x)$.
2. Compute $B(f(x)), B(f^{(2)}(x)), \ldots, B(f^{(j-1)}(x))$
3. Output $P\left(B(f^{(j-1)}(x)), \ldots, B(f^{(2)}(x)), B(f(x))\right)$

Observe that PRED runs in PPT; each step consists of calculating f, B, or P a polynomial number of times, and each of these can be calculated in PPT. Now we introduce the main ingredient of this proof.

Claim: $\Pr[\mathsf{PRED}(f(x)) = B(x)] = \Pr[P(b_1,\ldots,b_{j-1}) = b_j \text{ for } (b_1,\ldots,b_m) = G(\mathcal{U}_n)] > \frac{1}{2} + \epsilon$.

Once we have proved this claim, we will have the contradiction we need; we will have shown that the PPT algorithm PRED predicts $B(x)$ given $f(x)$ with one half plus non-negligible probability.

Proof of Claim. Since f is a permutation, it has a well-defined inverse f^{-1}. So from the definition of G, we have

$$\begin{aligned} G(f^{-(m-j)}(x)) &= B(f^{(m-1)}(f^{-(m-j)}(x)))\|\cdots\|B(f(f^{-(m-j)}(x)))\|B(f^{-(m-j)}(x)) \\ &= B(f^{(j-1)}(x))\|\cdots\|B(f(x))\|B(x)\|B(f^{-1}(x))\|\cdots\|B(f^{-(m-j)}(x)). \end{aligned}$$

Thus as we have defined it, $\mathsf{PRED}(f(x))$ outputs $P(b_1, \ldots, b_{j-1})$ for the first $j-1$ bits of $(b_1, \ldots, b_m) = G(f^{-(m-j)}(x))$. Moreover, we have $b_j = B(x)$. Therefore,

$$\Pr[\mathsf{PRED}(f(x)) = B(x)] = \Pr[P(b_1, \ldots, b_{j-1}) = b_j \text{ for } (b_1, \ldots, b_m) = G(f^{-(m-j)}(\mathcal{U}_n))]$$

But we have assumed that f is a permutation, and so $f^{-(m-j)}$ is also a permutation. And the uniform distribution $\mathcal{U}_n$ is invariant under permutation. So $G(f^{-(m-j)}(\mathcal{U}_n)) = G(\mathcal{U}_n)$. Therefore we have

$$\Pr[\mathsf{PRED}(f(x)) = B(x)] = \Pr[P(b_1, \ldots, b_{j-1}) = b_j \text{ for } (b_1, \ldots, b_m) = G(\mathcal{U}_n)] > \frac{1}{2} + \epsilon$$

for some non-negligible ϵ (by assumption). This proves the claim. So as argued previously, the PPT algorithm PRED predicts $B(x)$ given $f(x)$ with one half plus non-negligible probability, contradicting the fact that B is hard-core for f. Thus we conclude by contradiction that G passes all next-bit tests, and is thus a PRG. $\square$

For the special case of Goldreich-Levin Hard-Core Bit, our input consists of r and x, and the hardcore bit is $B(f^{(i)}(x), r) = \langle f^{(i)}(x), r \rangle$. We can make both $f^{(m)}(x)$ and r public.

The previous theorem also can be strengthened to work for any one-way function. The proof of this theorem is harder, and is given in [HILL99].

Theorem 2. *If there exists a one-way function, then there exists a PRG for all polynomial stretches.*

Example – EXP generator: If p is a prime, and g is a generator for the multiplicative group $\mathbb{Z}_p^*$, then the EXP generator with parameters p and g takes an input seed $s \in \mathbb{Z}_p^*$ and outputs

$$\text{msb}_p(s) \| \text{msb}_p\left(g^s\right) \| \ldots \| \text{msb}_p\left(g^{g^{\cdot^{\cdot^{\cdot^{g^s}}}}}\right)$$

where $\text{msb}_p(x)$ is the most significant bit of x modulo p. The cost incurred by the EXP generator to generate m bits is $\mathcal{O}(mn^3)$, where n is the length of p. This can be improved to $\mathcal{O}(mn^3/\log(n))$ by outputting the $\log(n)$ most significant bits rather than just the most significant bit; these bits can be proved to be “simultaneously” hard core – a notion you have seen in Problem Set 2.

2 Applications of PRGs

2.1 Derandomization

BPP (bounded-error probabilistic polynomial time) is defined as the complexity class of all languages L for which there exists a polynomial time algorithm M such that

- if $x \in L$, then
$$Pr[M(x, y) \text{ accepts}] > \frac{2}{3}$$
- if $x \notin L$, then
$$Pr[M(x, y) \text{ accepts}] < \frac{1}{3}$$

where the probability (in both cases) is taken over the random coin tosses y of M. Note that $|y| \leq \text{poly}(|x|)$.

The complexity class $\mathsf{DTIME}(f(n))$ is defined as the set of all languages L for which membership of a string x of length n can be decided by a deterministic algorithm in time $f(n)$.

One of the major open questions in complexity theory is that of derandomization. That is, given a randomized algorithm for a certain problem, can we come up with another algorithm that solves the same problem but is deterministic, that is, makes no coin tosses. We will prove that if PRGs exist (or equivalently, if one way functions exist), we can achieve some derandomization. Intuitively, this is because we can take a few random bits and expand them deterministically to many close-to-random bits, in fact, any random bits that the randomized algorithm will need. The question that remains is that where does one get the few random bits?

Theorem 3. *If one way functions exist,*

$$\mathsf{BPP} \subseteq \bigcap_{\epsilon>0} \mathsf{DTIME}\left(2^{n^{\epsilon}}\right)$$

Proof. Fix an $\epsilon > 0$. Let $L \in \mathsf{BPP}$ and let M be a probabilistic algorithm for L. Let y be the random coins tossed by M. We define a deterministic algorithm M' as follows: first, let G be a pseudorandom generator (which can be constructed from any one way function), that on input of size n^ϵ outputs a string of length $|y|$ (note that $|y|$ is polynomial in $|x| = n$ and hence such a generator exists). Then, on each input s of size n^ϵ, M' computes $M(x, G(s))$. After all computations are done, M' takes the majority as the answer and outputs it.

Note that M' is a deterministic algorithm. Furthermore, it runs in time $2^{n^\epsilon} \cdot t$, where t is the running time of M which is polynomial in n. We now have to show that M' decides every input x correctly. Suppose, for the sake of contradiction that it does not. This could be due to the following reasons. Suppose there exists an $x \in L$ such that

$$\Pr[y \xleftarrow{\$} \{0,1\}^{|y|} : M(x,y) \text{ accepts}] > \frac{2}{3}$$

however

$$\Pr[s \xleftarrow{\$} \{0,1\}^{n^\epsilon} : M(x,G(s)) \text{ accepts}] < \frac{1}{2}$$

Since the two probabilities differ by a non-negligible amount, one can construct a distinguisher for G as follows: on input y, run $M(x, y)$ and if it accepts, declare that y was random, and if it does not, declare that y was pseudorandom. While this distinguisher succeeds with non-negligible probability, it is non-uniform in the sense that it has x hardwired in it. Note that the fact that M' does not work correctly only pointed to the existence of such an x and not a means to efficiently find it. Thus, in order for this reduction to work, we will have to assume that the one way function we started out with is secure against non-uniform adversaries – these are a family of adversarial circuits, one for every input length. Note here that the adversarial circuit for each input length n would have a string x of length n hardwired into it.

Another source reason why M' might fail is that there exists an $x \notin L$ such that

$$\Pr[y \xleftarrow{\$} \{0,1\}^{|y|} : M(x,y) \text{ accepts}] < \frac{1}{3}$$

however

$$\Pr[s \xleftarrow{\$} \{0,1\}^{n^\epsilon} : M(x,G(s)) \text{ accepts}] > \frac{1}{2}$$

Since the two probabilities differ by a non-negligible amount, one can construct a distinguisher for G as follows: on input y, run $M(x, y)$ and if it accepts, declare that y was pseudorandom, and if it does not, declare that y was random. The reason as to why this distinguisher works is the same as before. This completes the proof. □

2.2 Symmetric (Private-Key) Encryption

Recall the Shannon-approach to construct private-key encryption schemes. The idea was to mask the message with a completely random string that the users of the scheme agreed upon. We had the problem that we needed to agree upon a as many random bits as we needed to transmit. However, we now have a way to generate a large number of close-to-random bits from a small number of random bits. This suggests the following scheme.

First, we agree on a randomly chosen seed s in the domain of a one-way function f. To encrypt an ℓ-bit message m, we compute $G(s)$ for some cryptographically strong PRG G, which stretches s to ℓ-bits, and send $c = G(s) \oplus m$. To recover m, we can compute $G(s)$ and $m = c \oplus G(s)$.

Claim 4. *The above encryption scheme achieves computational secrecy.*

Proof. Since G is a CS-PRG, $G(s)$ is computationally indistinguishable from uniform distribution, and computational indistinguishability is preserved under any polynomial time algorithm, including XOR. So for any m, $c = m \oplus G(s)$ is computationally indistinguishable from a uniformly random string, and hence the encryption scheme achieves computational secrecy. $\square$

This algorithm can be used to encrypt many messages if one maintains state. In order to avoid maintaining state, we need a stronger object, called pseudorandom functions.

3 Pseudorandom Functions

We now define a more powerful object, pseudorandom function families, and show how to generically construct them using pseudorandom generators. At high level, a function family is said to be pseudorandom if random elements from the family cannot be distinguished from truly random functions by any probabilistic polynomial time oracle machine that adaptively queries its oracle in an attempt to figure out if the oracle is a member of the function family or a truly random function. Pseudorandom function families constitute a very powerful tool in cryptographic settings: the functions in such families are easy to select and compute, and yet retain all the desired statistical properties of truly random functions with respect to polynomial-time algorithms.

Consider the set $\mathcal{U}_n$ of functions from n-bit strings to n-bit strings. The size of $\mathcal{U}_n$ is 2^{n2^n}, because for each value in the domain of size 2^n there are 2^n possible outputs; hence, in order to specify a function in $\mathcal{U}_n$. We are going to try to find a much smaller subset F_n of "easily computable" functions within $\mathcal{U}_n$ with the property that it is "hard" to distinguish between functions uniformly drawn from F_n and functions uniformly drawn from $\mathcal{U}_n$. Such a set F_n would intuitively be "as good as" $\mathcal{U}_n$ for all practical purposes; a function uniformly drawn from F_n would then be *pseudorandom*, but not truly random, and yet would *appear* to be truly random under the inspection of any probabilistic polynomial-time procedure. It would also allow us to represent it with a more reasonable number of bits, n, instead of the intractable $n2^n$. This gives us the following definition of pseudorandom functions.

Definition 5 (Pseudorandom Function Family). *We say that a function family $\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$ is* pseudorandom *if*

- *Each function in F_n is easy to sample.*
- *There is an efficient algorithm E (the "evaluator") such that*

$$E(s, x) = f_s(x)$$

  for all $x, s \in \{0,1\}^n$.
- *The function family $\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$ is pseudorandom, i.e., for all PPT algorithms $\mathcal{A}$,*

$$\left| \Pr\left[f \leftarrow F_n \ : \ \mathcal{A}(1^n)^f = 1\right] - \Pr\left[f \leftarrow \mathcal{U}_n \ : \ \mathcal{A}(1^n)^f = 1\right] \right| = \text{negl}(n) \ .$$

Remark. We have already constructed pseudorandom generators, starting from one-way permutations (in fact, it turns out that one-way functions suffice). Note that a pseudorandom generator with expansion factor $\ell(n)$ "hides" 2^n strings among the $2^{\ell(n)}$ strings with $\ell(n)$ bits. Similarly, we now want to hide 2^n

functions (those in F_n) among the 2^{n2^n} functions from n-bit strings to n-bit strings. However, at least at some intuitive level, the task of constructing pseudorandom function families is harder than constructing pseudorandom generators, because the ratio $\frac{2^n}{2^{n2^n}}$ is exponentially smaller than the ratio $\frac{2^n}{2^{\ell(n)}}$, i.e., the ratio of the size of the subset we are trying to "hide" to the size of the whole set is much smaller! Nonetheless, we show how to use pseudorandom generators to construct pseudorandom function families.

3.1 Constructing PRFs from PRGs

We now describe the construction of a pseudorandom function family from a pseudorandom generator (this is the [GGM86] construction). Let G be a pseudo-random generator with expansion factor $\ell(n) = 2n$, and, for any string s, define $G_0(s)$ and $G_1(s)$ to be the first half and second half of $G(s)$, respectively. We construct the tree below by placing s as the root node, and making $G_0(s)$ its left child and $G_1(s)$ its right child. We recursively create children using the PRG with expansion factor $2n$ to generate all n levels. For any $n \in \mathbb{N}$, we construct this tree implicitly and define $F_n = \{f_s\}_{s \in \{0,1\}^n}$, where $f_s \colon \{0,1\}^n \to \{0,1\}^n$ is defined as:

$$f_s(x) := G_{x_n}(G_{x_{n-1}}(G_{x_{n-2}}(\ldots G_{x_2}(G_{x_1}(s))\ldots)))$$

Recall that G is deterministic, so that f_s is indeed a function.

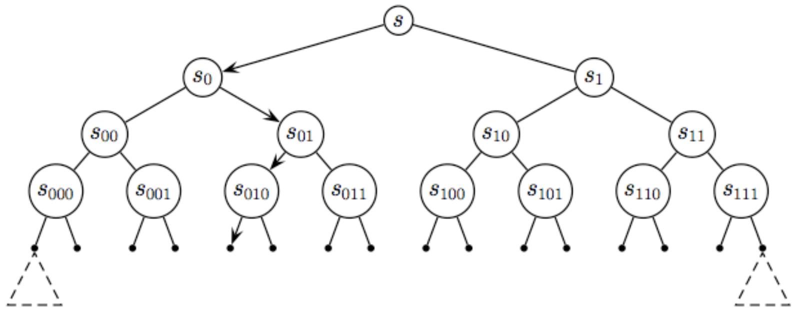

Figure 1: Constructing a pseudorandom function using a pseudorandom generator. The exponentially large tree is not actually constructed, but, given any input $x \in \{0,1\}^n$, it is possible to find $f_s(x)$ by simply following the path of the tree corresponding to the bits of x; doing so requires only n applications of the pseudo-random generator G, starting with seed s.

The efficient evaluation of the function f_s can be visualized by considering a full binary tree of depth n, with the n-bit seed s at the root; the remaining nodes are initially empty. On input a n-bit string x, the computation of $f_s(x)$ proceeds as follows: compute $s_{x_1} = G_{x_1}(s)$ (by computing $G(s)$ and considering the first half if $x_1 = 0$ and the second half if $x_1 = 1$), compute $s_{x_1x_2} = G_{x_2}(s_{x_1})$, and so on, until the last value $s_x = G_{x_n}(s_{x_1\ldots x_{n-1}})$; the output is s_x. See Figure 1 for a diagram of this binary tree. Clearly this takes time polynomial in n because we only use one node at each level and there are n levels. In the next lecture we prove that the construction works. At high level, if G actually produced random strings, it is easy to see that this construction does work. however, the output of G is pseudorandom and there seems to be reason to worry that applying G iteratively on progressively less-random strings might result in predictable outputs. We show in the next lecture that this is not the case.

Theorem 6. *$\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$ is a pseudorandom function family.*

References

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.