CS 282A/MATH 209A: Foundations of Cryptography — Prof. Rafail Ostrovsky

Lecture 5

Lecture date: 2,14,16 Feb. 2005 — *Scribe: B. Poon, I. Woldeit, C. Mar*

1 P vs. BPP

Suppose we have a machine that can't flip coins, but want to simulate a machine that can.

Say that we have a BPP machine for determining whether a given string x is in the language L. Recall that a BPP machine takes as input the string x , with say $|x| = n$, and a string of random bits, of length polynomial in n (say $Q(n)$), and outputs either a yes or a no. If $x \in L$ then the probability that the output is "yes" is at least $\frac{3}{4}$ (probability taken over all possible strings of $Q(n)$ random bits), and if $x \notin L$ then the probability of a "yes" output is no more than $\frac{1}{4}$.

Now if we have such a BPP machine and an x, there is a simple algorithm which will tell us definitely, not just probabilistically, whether $x \in L$ or not. It goes like this:

- Try all $2^{Q(n)}$ random strings
- Count how many give yes and no.
- If there are more yes'es, $x \in L$; if there are more no's, $x \notin L$.

This is an exponential time algorithm. We can do better. If non-uniform one-way functions exist, then we can recognize BPP in sub-exponential time (i.e. algorithm can run in $2^{Q(n)^{\epsilon}}$ time, for any $\epsilon > 0$.)

Theorem 1 [Yao]*: If there exist non-uniform one-way functions then BPP is contained in subexponential time.*

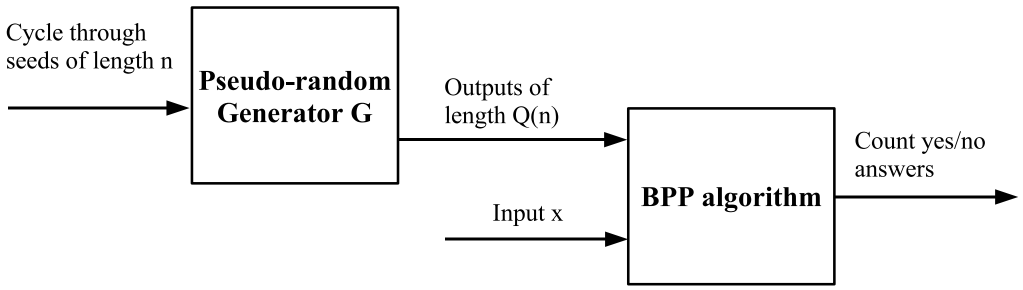
That is,

$$BPP \subseteq \bigcap_{\epsilon>0} DTime(2^{n^{\epsilon}})$$

The algorithm, call it P', uses a pseudo-random generator G : n-bits $\rightarrow Q(n)$-bits . The algorithm is:

- Cycle through all 2^n possible seeds of G.

- Take the outputs of G as the $Q(n)$-bit strings to be input into the BPP machine along with the input x.
- Out of those strings, count how many times the BPP machine says yes, how many times it says no.
- If yes' are more common, $x \in L$; if no's are more common, $x \notin L$.



Proof Assume our algorithm P' makes a mistake. We will prove this implies that: We can construct a non-uniform poly-size distinguisher of $g \in \{G(n)\}_{Q(n)}$ and $u \in \{U\}_{Q(n)}$, where $G(n)$ is the outputs of the pseudo-random generator and U is the uniform, i.e. truly random, distribution.

Suppose that x, $|x| = n$, is a string on which P' makes a mistake, that is, either $x \in L$ and P' says "no" or $x \notin L$ and P' says "yes'. Note that x is given to us in a non-uniform fashion, and we just "hard-wire" it into our circuit. We will show that in either case, x together with our BPP machine can be used for a decision process T on $Q(n)$-bit strings which is polynomial time, and which has a non-negligible difference between the probabilities of saying "yes" for a truly random or a pseudo-random string. But this contradicts the definition of a pseudo-random generator G: there is no poly-time distinguisher of the output of G from a truly random distribution, i.e. no poly-time algorithm that says 1 for one of those distributions non-negligibly more often than for the other. This contradiction establishes the fact that P' works.

Case 1. $x \in L$ for which P' makes a mistake and says no.

This means that on more then half of the pseudo-random strings the output is no but on more then three quarters of random string the output is yes. Hence we have a distinguisher with distinguishing probability a 1/4.

Case 2. $x \notin L$ but we P' makes a mistake and says yes.

This means that for more then half of the pseudo-random strings the machine says yes, while for less the a 1/4 of truly random strings the machine says yes. Hence again the distinguishing probability is a 1/4. ■

2 Bit Commitment Protocol

Many cryptographic protocols use as their basis a bit commitment protocol. Some of these applications are zero knowledge protocols, identification schemes, multi-party protocols, and coin flipping over the phone.

2.1 Pseudo-random Generators

First we note that the existence of pseudo-random generators is equivalent to the existence of one-way functions. This was established by Håstad, Impagliazzo, Levin, and Luby in 1999 [HILL99]. We will omit this proof in one direction due to its complexity.

Theorem 2 *There exists a pseudo-random generator iff there exists a one-way function.*

Here we will only show that any pseudo-random generator is a one-way function. Suppose that a pseudo-random generator $G : \{0,1\}^n \to \{0,1\}^{l(n)}$, $l(n) \geq n+1$, is not a one-way function. Then a pseudo-random string of length $l(n)$ can be inverted with non-negligible probability. However, recall that there are only 2^n seeds of length n and there are $2^{l(n)}$ strings of length $l(n)$, which means that most truly random strings of length $l(n)$ cannot be inverted. This is due to the fact that there are at most $\frac{2^{l(n)}}{2^n}$ times as many outputs as seeds. Based on the assumption that $l(n) \geq n+1$, there are at least twice as many outputs as seeds. If a pseudo-random string of length $l(n)$ can be inverted with non-negligible probability, a distinguisher between truly random and pseudo-random strings can be constructed. If we have such a distinguisher, we can try to invert a given string: if we are able to invert it, it is pseudo-random; if not, then it is truly random. However, a pseudo-random generator, by definition, has an output that cannot be distinguished from truly random in polynomial time. Therefore, if it is a pseudo-random generator, its output cannot be inverted in polynomial time and it is a one-way function.

The work of Håstad, Impagliazzo, Levin, and Luby showed that any one-way function can be used to construct a pseudo-random generator.

2.2 Bit Commitment Protocol

The existence of a good pseudo-random generator allows us to construct a secure bit commitment protocol. The proof was developed by Naor in [NAOR01].

Theorem 3 *If there exists any pseudo-random generator G:* $\{0,1\}^n \to \{0,1\}^{3n}$*, it implies that there exists a bit commitment protocol.*

Bit Commitment Protocol Formal Definition

The protocol consists of two stages: the commit stage and the reveal stage. The formal definition of the bit commitment protocol follows:

- Before the protocol begins:
  1. A good pseudo-random generator, $G(\cdot) : n \mapsto 3n$, known to both Alice and Bob.
  2. Alice is given (as an input), a secret bit b unknown to Bob.
- Commit Stage:
  1. Bob selects bit vector $R = \{0,1\}^{3n}$ and sends it to Alice.
  2. Alice selects a seed $S = \{0,1\}^n$ and computes $G(S) = Y$, where $Y = \{0,1\}^{3n}$.
  3. Alice sends to Bob the vector $Z = \{0,1\}^{3n}$ where $Z_i = Y_i, 1 \leq i \leq 3n$ if $b = 0$ and $Z_i = Y_i \oplus R_i, 1 \leq i \leq 3n$ if $b = 1$.
- Reveal Stage:
  1. Alice sends S to Bob.
  2. Bob computes $G(S)$. If $G(x) = Z$, $b = 0$; if $G(S) \oplus R = Z$, $b = 1$; otherwise, repeat the protocol.

Bit Commitment Protocol Proof of Security

In order to prove that this is a secure construction, we must prove what are known as the binding and privacy properties. The binding property requires that Alice cannot change her bit selection and that this is verifiable by Bob. The privacy property requires that Bob cannot determine any information about Alice's bit selection until Alice allows him to.

Claim 4 (Binding Property) *Even if Alice is has infinite computing power and memory, she cannot cheat with probability greater than* $\frac{1}{2^n}$.

Proof Consider any two completely random seeds, S_0 and S_1, from the set $\{0,1\}^n$. Denote the output of the pseudo-random generator, $G(\cdot)$, corresponding to the seeds be Y_0 and Y_1, respectively, and be in the set $\{0,1\}^{3n}$. Also let Bob's choice of random string, R, be in the set $\{0,1\}^{3n}$. We have assumed that Alice has infinite computing power and memory so she can cheat by creating the following lookup table. For every possible S_0 and S_1, Alice makes a mapping to a string, R'. This mapping is done by computing the bitwise exclusive-NOR of the output of the pseudo-random generator, $G(\cdot)$, corresponding to each pair of seeds. The bitwise exclusive-NOR is calculated by assigning $R'(i) = 0$ when $Y_0(i) \neq Y_1(i)$ and $R'(i) = 1$ when $Y_0(i) = Y_1(i)$.

Alice can cheat with this table if $R \in \{R'\}$, that is, if R is in the set of elements of Alice's table. If this is the case, Alice can choose what value she wants for b by sending S_0 for $b = 0$ and S_1 for $b = 1$, where S_0 and S_1 are the seed pair corresponding to $R' = R$.

What is the probability that Alice can cheat in this scenario? The seeds are of length n, so any seed pair can be thought of as a random string of length $2n$. This implies that there are 2^{2n} possible random pairs of seeds. This limits the number of R's on which Alice can cheat to 2^{2n}. But, the length of R is $3n$ so there are 2^{3n} possible sequences for R. This gives us:

$$\begin{aligned} P(R \in \{R'\} = (G(S_0[i]) \oplus G(S_1[j]))\, \forall\, 0 \leq i, j \leq 2^n) &= \frac{2^{2n}}{2^{3n}} \\ &= \frac{1}{2^n} << \frac{1}{n^c} \end{aligned}$$

where $\oplus$ denotes the XOR operation.

This is a simple argument based on the number of possible random strings available. Thus, if R is random, the probability that Alice can cheat, even with infinite computing power and memory, is negligible. ■

Claim 5 (Privacy Property) *Bob cannot predict the bit, b, with probability greater than $\frac{1}{2} + \varepsilon(n)$.*

Proof For this proof, we will use the fact that if a string is truly random, bitwise XOR'ing it with R will create a truly random output. First, assume that Bob can determine the bit, b, when given Z. Now, assume that Y is a pseudo-random string generated from $G(\cdot)$. When Alice gives Bob the string, Y, Bob can determine b. Next we must look at the case when Y is a truly random $3n$ bit sequence. In this case, it is information theoretically impossible to predict whether Z is Y or $R \oplus Y$ because both are truly random sequences. However, in the first case, Y is pseudo-random and Bob can predict b. This implies that Bob can be queried as a distinguisher of random and pseudo-random sequences. Due the the proof that there is not a distinguisher of random and pseudo-random sequences, we have a contradiction which proves that Bob cannot determine the bit, b. ■

Having now established both the binding and the privacy property, we can see that the bit commitment protocol is secure.

3 Pseudo-Random Functions

Pseudo-random functions are a useful extension of pseudo-random generators. As the name suggests, pseudo-random functions create strings of seemingly random bits and, for any fixed input, will give the same string of bits every time. We will first define pseudo-random functions with more precision and then show one method of constructing them. Next, we will give a detailed proof of security for pseudo-random functions. Finally, some possible uses of pseudo-random functions will be introduced.

3.1 Pseudo-Random Functions

For our definition of pseudo-random functions, we need to begin by defining an oracle Turing machine. An oracle Turing machine can ask a series of questions, one at a time. Each question must wait for an answer before a successive one may be asked. The questions are responded to by an oracle which has access to a single function. This oracle, as the historical reference suggests, reveals nothing about how it got the information, only an appropriate response. Not even information such as heat dissipation, power requirements, or time of operation can be discerned from the output of the oracle. This is an important requirement because it eliminates any attacks on the physical operation of the function, which is an implementation issue and not our motivation here.

When the oracle Turing machine has a response from the oracle, it uses this information to come up with a binary decision. We do not define the decision process more than this because we seek to make the requirements as general as possible. When a decision has been reached, the oracle Turing machine can make another query to the oracle. This can be repeated any polynomial number of times.

A truly random function U: $\{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ is informally a lookup table with 2^n entries of random strings of length $l(n)$. A pseudo-random function F: $\{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ is a function that an oracle Turing machine cannot distinguish from a truly random function in poly-time. More formally:

Definition 6 *Pseudo-random functions $\{F\}$ and $\{U\}$ are poly-time indistinguishable ensembles of functions if $\forall$ probabilistic poly-time oracle machines A, $\exists N$ such that $\forall n > N$:*

$$(Pr(A^{\{F\}}(1^n) = 1)) - (Pr(A^{\{U\}}(1^n) = 1)) < \frac{1}{n^c}$$

$(Pr(A^{\{F\}}(1^n) = 1))$ means that the oracle machine A has access to some function, defined by its seed as $F_S(\cdot)$, from the ensemble of functions $\{F\}$. The definition means that any BPP oracle Turing machine, A, has a negligible probability of distinguishing whether a given function is from $\{F\}$ or $\{U\}$.

This means that, for any adversary with access to an oracle, the adversary cannot distinguish whether the oracle is using a pseudo-random function or a truly random huge string.

3.2 Constructing Pseudo-Random Functions

Having defined pseudo-random functions, we now move to their construction given a pseudo-random generator as outlined in [GGM86].

Given a pseudo-random generator $G : \{0,1\}^n \to \{0,1\}^{2n}$ with seed S, where $|S| = n$, we can create a hypothetical construction of the pseudo-random function $F_S(\cdot)$. First, we compute $G(S) = [S_0, S_1]$, where $|S_0| = |S_1| = n$. Specifically, S_0 and S_1 are the first and second halves of the output string, $G(S)$, respectively. Next, we compute $G(S_0) = [S_{00}, S_{01}]$ and $G(S_1) = [S_{10}, S_{11}]$. Doing this recursively n times will create a tree with height n. Note that this tree cannot be created and stored by any poly-time machine because it is exponential in size. We can still use it in an efficient algorithm by constructing and remembering parts of the tree on an as-needed basis.

Denoting x as the input string, we can use x as a set of instructions for which nodes of the tree to actually create and store. This can be thought of as the path to be taken down the tree: for index $i : 1 \leqslant i \leqslant n$, we take the left branch if $x_i = 0$ and the right branch if $x_i = 1$. For example, if $x = 0100...$, we would begin by taking the left branch and compute $G(S_0) = [S_{00}, S_{01}]$. Then we would take the right branch and compute $G(S_{01}) = [S_{010}, S_{011}]$, followed by taking the left twice. Proceed in this manner for all i. Finally, we have a string of $2n$ bits at the leaf resulting from the walk defined by input x.

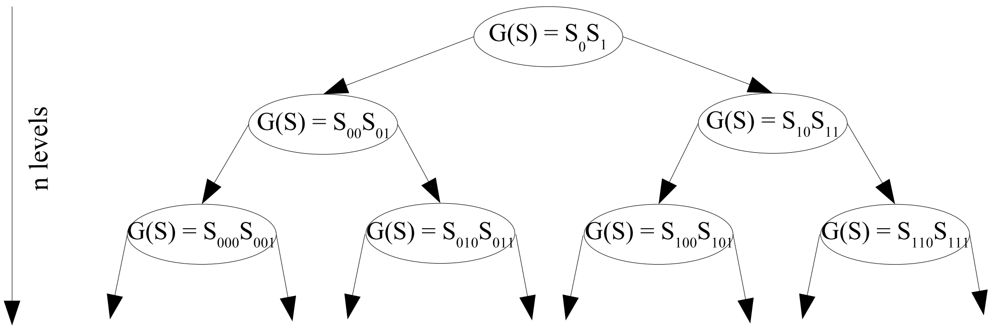


Figure 1: Tree construction

3.3 Proof of Security

Theorem 7 *The output of a pseudo-random function $F_S(\cdot)$, is indistinguishable from a truly random sequence of the the same size under the following assumptions:*

1. *$F_S(\cdot)$ is constructed from a good pseudo-random generator $G(\cdot)$*
2. *The seed S is truly random*

We first define the extended statistical test.

Polynomial indistinguishability definition is robust

Next, we consider the following questions: why do we limit a polynomial-time statistical test to receive only *one* sample from one of the two distributions? What happens for a test which gets any polynomial number of samples?

The *extended statistical test*, T', is defined as a program which takes a polynomial number of inputs and outputs a single bit. Given two ensembles of distributions, $\{X\}$ and $\{Y\}$, T' is given some polynomial number of samples from either $\{X\}$ or $\{Y\}$ (i.e. all from the same distribution). It then makes some binary decision and represents this as the output of either a 0 or a 1. $\{X\}$ and $\{Y\}$ pass the extended statistical test if T' can only distinguish with negligible probability whether all the samples came from $\{X\}$ or $\{Y\}$. That is,

Definition 8 $\{X\}$ and $\{Y\}$ pass extended statistical test T' if $\forall c_1, c_2$, $\exists N$ such that $\forall n > N$

$$\left[| \Pr_{\{X,\text{coins of } T'\}} (T'(X_1, \ldots, X_{n^{c_2}}) = 1) - \Pr_{\{Y,\text{coins of } T'\}} (T'(Y_1, \ldots, Y_{n^{c_2}}) = 1) | \right] < \frac{1}{n^{c_1}}$$

We claim that this definition is not stronger than the first definition. That is, if two polynomially sampleable distributions can be distinguished on polynomially many samples, then they can be distinguished on a single sample.

Claim 9 If X and Y [are sampleable] distributions which can be distinguished by a [uniform] extended statistical test T', then there exist a (single sample) [uniform] statistical test T which distinguishes X and Y.

Proof Let let k =poly(n) and let $\epsilon(n) = 1/k$. We assume that two there exists T' and show how to construct T. Assuming that there exists T' means, w.l.o.g. that

$$\Pr_{X_n}(T'(X_1, X_2, X_3, ..., X_{poly}) = 1) - \Pr_{Y_n}(T'(Y_1, Y_2, Y_3, ..., Y_{poly}) = 1) > \epsilon(n)$$

Consider "hybrids" P_j, for $0 \le j \le k$, where in P_j the first j samples come from Y_n and the remaining samples come from X_n:

$$
\begin{aligned}
P_0 &= x_1\, x_2\, x_3\, x_4 \ldots x_k \\
P_1 &= y_1\, x_2\, x_3\, x_4 \ldots x_k \\
P_2 &= y_1\, y_2\, x_3\, x_4 \ldots x_k \\
P_3 &= y_1\, y_2\, y_3\, x_4 \ldots x_k \\
&\ldots \\
P_k &= y_1\, y_2\, y_3\, y_4 \ldots y_k
\end{aligned}
$$

We know that $P_0 - P_k > \epsilon(n)$, and therefore, $\exists j$ such that $P_j - P_{j+1} > \epsilon(n)/k$ (which is another 1/poly fraction!) Consider a distribution:

$$P(\boxed{\mathrm{z}}) = y_1\, y_2\, y_3 \ldots y_j\, \boxed{\mathrm{z}}\, x_{j+2} \ldots x_k$$

Notice that if z is a sample from Y_n then $P(z) = P_j$ and if z is a sample from X_n then $P(z) = P_{j+1}$. Hence, if we are given z on which we have to guess which distribution it came from, if we put z in the box above, and somehow fix other locations we could distinguish on a single sample z. Two questions remain: (1) how do we find the correct $j+1$ position, and (2), how do we fix other values. The answers differ in uniform and non-uniform case:

<u>non-uniform case</u> (i.e. both T' and T are circuits): Since T is a circuit, we can non-uniformly find correct $j+1$ value and find values to other variables which maximizes distinguishing probability.

<u>uniform case</u> : Since X_n and Y_n are sampleable, we can fix values different from j to be samples from X_n and Y_n and by guessing correct j (we guessed j position correctly with probability 1/poly). The distinguishing probability could be further improved by experimenting with the distinguisher we get (again using sampleability of X_n and Y_n!) to check if our choice of j and samples of other positions are good. ■

We will now use this result to prove that the pseudo random function $F_S(\cdot)$ described previously is secure.

Proof Consider four imaginary trees with different levels of randomness. The first tree R is completely random. It can be implemented by allowing the adversary to ask for an output and by outputting a walk of random strings. These random strings must then be remembered for future queries, since the value of the function for a given input should not change over time. Because the adversary is poly-time and can ask only polynomially many questions, the machine implementing this completely random tree needs only remember a polynomial number of random strings.

The fourth tree PR is completely pseudo-random and can be implemented as discussed above.

The middle two trees $T1$ and $T2$ are constructed such that each contains a balanced parent subtree tree whose nodes are all composed of random samples of the function $U(i)$ described previously. For the random parent subtrees, the border values to a given prefix of i (corresponding to a unique walk of the subtree) must be remembered, as in the case of the fully random tree. As before however, a polynomial time adversary can only issue polynomially many queries, so a polynomial machine is capable of simulating the trees.

1. T_1 which represents the output of a tree constructed with i levels of truly random nodes and $n - i$ pseudo-random nodes.
2. T_2 which represents the output of a tree constructed with $i + 1$ levels of truly random nodes and $n - (i + 1)$ pseudo-random nodes.

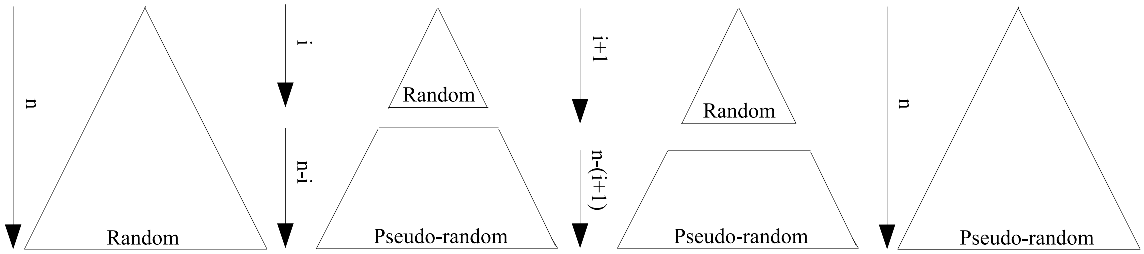


Figure 2: Hybrid Trees T1 and T2

Now, consider an poly-time adversary D which must distinguish between the two trees. It is given an oracle machine A^{T_i} which interfaces to either $T_1(.)$ or $T_2(.)$, but reveals no information other than the value of the function on a given input. D can interactively query the oracle based on previous output.

Now, at level i, D can distinguish between T_1 and T_2 by guessing the value of i, which D will guess correctly a non-negligible $\frac{1}{n}$ times because there are precisely n levels.

We now allow D to see the outputs of $F_s(x)$ for k inputs $x_1, x_2, \ldots, x_k$. By our construction, each x_j corresponds to a walk down the tree. Furthermore, the first i bits of any such walk goes through the truly random portion of the tree. In order to make sure that we use the same truly random string for cases in which x_j starts with the same i bits, we can simply assign each node in the truly random portion of the tree an n-bit truly random string. For any x_j that shares the same i-bit prefix, the same node in that portion will thus be chosen.

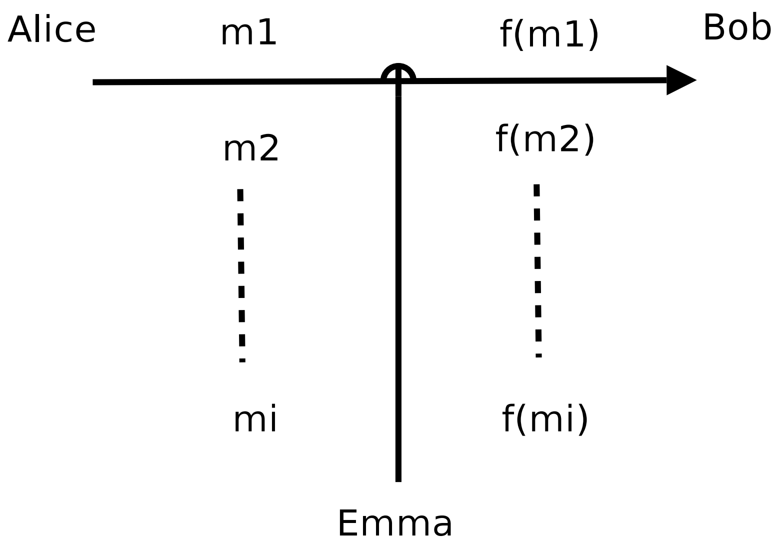
Note that, since D is a poly-time machine, it can only ask for polynomially many $F_s(x_j)$'s, which pass through a polynomially bounded number of locations at level i. Assuming that D can distinguish between these two trees, this is equivalent to saying that D can differentiate between the polynomially many pseudo random values sampled at level i in T_1 and the polynomially many random values sampled at level i in T_2. However, this is just the extended computational test which we have just shown to be secure. Therefore, adversary D cannot exist. ■

3.4 Applications of Pseudo-Random Functions

Now knowing how to construct pseudo-random functions, we can look at two of their applications: message authentication and secret societies.

Message Authentication

One application of pseudo-random functions is in authenticating messages between Alice and Bob, who share a seed s to a pseudo-random function F, i.e., F_s. In message authentication, Alice wishes to send Bob a message m (possibly in plaintext) that Bob can be sure was not a forgery from Eve, who does not know s.



To do so, Alice computes a message authentication code A_c where $A_c = F_s(m)$ and sends it with m as a pair: (m, A_c). Now, when Bob receives (m, A_c), Bob can compute $B_c = F_s(m)$ himself and ensure that $A_c = B_c$. If so, the message is authentic, else it is not. Here, Eve cannot predict n bits of a truly random string so she cannot forge a message m'.

Secret Societies

Pseudo-random functions can be used to prove membership in a secret society. Assume there exists a secret society of which a number of people are members. The people in this secret society wish to verify if someone else is a part of the society. If they were to use a password, that password could be overheard and non-society members could pose as members by repeating the password.

Instead, members memorize the seed s to a pseudo-random function F_s which is well known. Now, to verify that someone is in the society, a challenger ask for the value of $F_s(x)$ for a random input x. If the challenged person reveals the same value that the challenger computes, then the challenged person is indeed a member; if the challenged person reveals another value, then the challenged person is not a member. This works because if a non-member were to overhear a poly number conversation, they still do not know s and thus cannot compute $F_s(x)$ for an unheard x.

4 References

[HILL99] J. Hastad, R. Impagliazzo, L. Levin, and M. Luby. A pseudorandom generator from any one-way function. SIAM Journal on Computing, 28(4):13641396, 1999. A preliminary version appeared in 21st STOC, 1989.

[Naor01] Moni Naor, Bit Commitment Using Pseudorandomness. J. of Cryptology, Volume 4, pp. 151-158.

[GGM86] Oded Goldreich , Shafi Goldwasser , Silvio Micali, How to construct random functions, Journal of the ACM (JACM), v.33 n.4, p.792-807, Oct. 1986.