Introduction to Cryptography
Lecture Notes for CS-276
Fall 1994

Lecturer: Rafail Ostrovsky*

May 17, 1995

Abstract

This manuscript combines scribe notes of CS-276, a U.C. Berkeley graduate course "Introduction to Cryptography" which was given in the Fall of 1994. Many lecture notes require substantial revision and are incomplete. It is my intention to update these notes whenever I teach this course again.

* University of California at Berkeley and International Computer Science Institute at Berkeley. Supported by U.C. Berkeley, NSF postdoctoral fellowship and ICSI. e-mail: rafail@cs.berkeley.edu

Acknowledgements

First and for-most, I wish to thank Manuel Blum for arranging for me to teach this course, and for numerous stimulating discussions during the course. I am extremely grateful for all his help.

A lot of what I talked about in this lectures I learned from three people: Silvio Micali, my Ph.D. advisor, Oded Goldreich and Shafi Goldwasser, and I am very much indebted to them. Oded's 1989 lecture notes were especially helpful.

Many thanks to all the students who took scribe notes: Micah Adler, Elan Amir, John Byers, Sanjoy Dasgupta, Michael Galbraith, Rahul Garg, Dean J. Grannes, Sean Hallgren, Wendy Heffner, Todd Hodes, Ari Juels, Raph Levien, Jay Lorch, Vassilis Papavassiliou, Lars Rasmussen, and Shuzo Takahashi.

Finally, I thank my family for their support and love.

Contents

1 Introduction

1.1 Outline

This is a one-term graduate course in cryptography[1]. The course is self-contained, though general familiarity with computational complexity concepts and elementary algorithms is recommended. Current research topics will be emphasized. Grading will be based on a final project which will consist of reading and analyzing an additional paper related to one of the topics covered.

Topics will include foundations (complexity background; weak and strong one-way functions; hard-core predicates; uniform vs. non-uniform model, connection to P=NP question); Pseudo-random generators and pseudo-random functions; Public-key and private-key Encryption; Digital Signatures; Single-prover and Multi-prover Interactive Proof Systems; Commitment protocols, Zero-Knowledge; Reducibility among protocol problems, Oblivious Transfer; Completeness Theorems for two-party and multi-party secure protocols; Non-Interactive Zero-Knowledge and its applications. If time permits, we will cover advanced topics, such as software protection, protocols in the the full-information model, mobile adversaries (viruses) and knowledge complexity.

1.2 Organization

Lectures are on Tuesdays and Thursdays 2-3:30 in 507 Soda Hall. Office hours will be held after class (if there are questions) and in Nefeli when needed. Rafi can by reached by e-mail: rafail@melody.berkeley.edu.

1.3 Motivation

Cryptography existed since ancient times. For example, the task of sending a secret message between two allies which even if intercepted by an enemy, can not be understood (by an enemy) was of relevance even in ancient Rome (example: Ceaser cipher). Until recently, however, cryptography and cryptanalysis was more an art then a science, and cryptosystems were frequently broken. The most famous example is breaking of the "ENIGMA engine" by the allies (ENIGMA was an encoding scheme that Nazi used to send messages by radio to their U-boats during World War II).

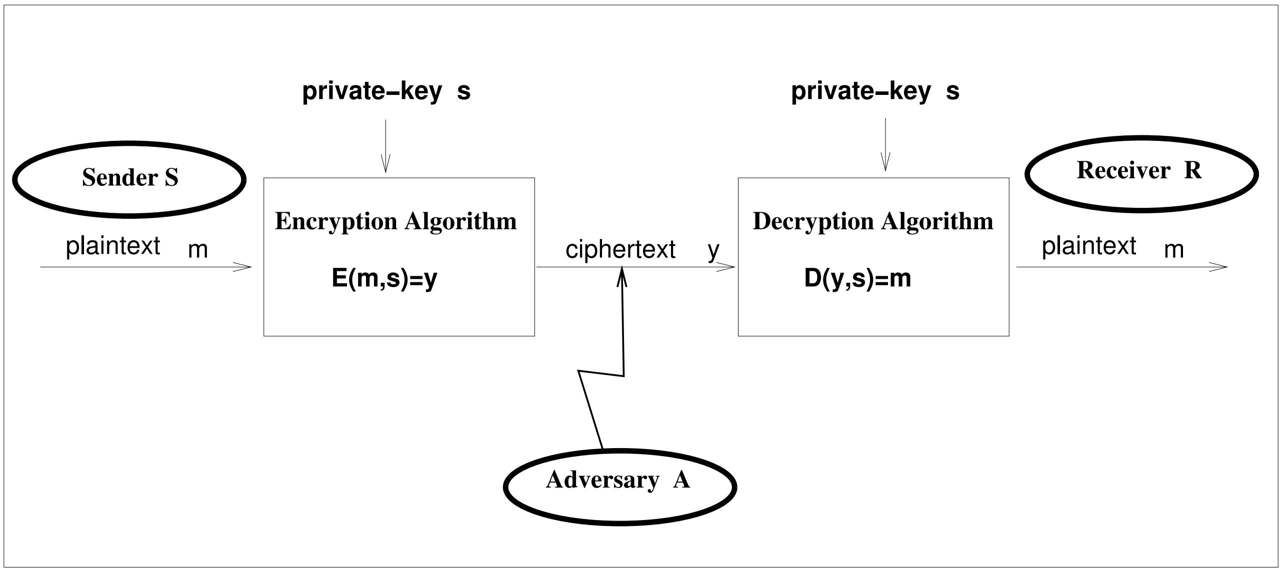
In this course, we explore how the task of secret communication (as well as many other problems which deal with privacy) could be formally defined, and proved correct under various assumptions. We will focus on formalizations of the notions of "privacy" and "fairness" (i.e., what does it mean for some information to be "private", and what does it mean for some game/protocol to be "fair") when considering various interactive tasks (like voting for a leader, or playing poker by telephone).

1.4 Introductory remarks about Private Key Encryption

Informally, private-key encryption is a game between three players: Sender S; Receiver R, and an Adversary A. The setting is as follows: S and R first get together and agree on a common *private key* s which adversary A does not know. Using this shared (between S and R) secret key,

[1]Scribe notes taken by: Zaphod Beeblebrox, August 23, 1994.

the sender S wishes to send messages m_1,...,m_l to receiver R such that R will understand them (since he knows the secret key s) but adversary A will not, even if he gets to hear everything that A sends to B.



In order to achieve that goal, sender S converts each *plain-text* message m_i to a *cipher-text* y_i using an "encryption algorithm" E and agreed upon secret key s (i.e., where $y_i = E(m_i, s)$). When receiver R gets cipher-text y_i, R uses a "decryption algorithm" D to "decipher" y_i (i.e. to get back $m_i = D(y_i, s)$. Notice that it should be the case that for all m, and all s, $D(E(m, s), s) = m$.

Informally, Adversary A gets to hear cipher-text $y_1, \ldots, y_l$ and he tries to "deduce" "some information" about the clear-text $m_1, \ldots, m_l$. However, at this point, we have not yet defined what this intuitive requirement means formally (it will actually take some work before we get back to this).

1.5 Perfect Secrecy and One-Time Pad

One interpretation is that the secrecy is "perfect", as defined by Shannon: suppose we have some *a priori* probability distribution on messages $m = m_1, m_2, \ldots, m_l$ that A is going to send. In particular, for any m let $Pr(m)$ denote an *a priori* probability of A sending m. Then for all m and for all $y = y_1, y_2, \ldots, m_l$ it should be the case that $Pr(m) = Pr(m|y)$.

If length of all messages is equal to the length of private-key, this is possible, and it is called a **one-time pad private-key cryptosystem**: pick a private key s such that $|s| = |m|$. When sending j'th bit of m, compute an exclusive-or (xor) of j'th bit of m with j'th bit of s and send the result $y_j = m_j \oplus s_j$. When R gets y_j it computes $y_j \oplus s_j = (m_j \oplus s_j) \oplus s_j = m_j$. The scheme has "perfect" security: For all m_j, $Pr(y_j = 0) = Pr(y_j = 1) = \frac{1}{2}$ where the probability is taken over the choice of s_j.

What if we want to have length of private key to be much less then the total length of m? Shannon proved that this is impossible if we want "perfect" security. However, suppose, instead of requiring that adversary should not be able to decrypt at all, we only require that adversary should not be able to decrypt "quickly" (i.e. that it is "hard" for A to decrypt in polynomial time.) Then we will see that (after we formally define this notion) we can use much smaller key!

1.6 Attacks on a Cryptosystem

We are going to consider several types of attacks:

- **known cipher-text attack:** adversary gets to see only cipher-text $E(m_1, s)$, $E(m_2, s)$, ..., $E(m_l, s)$. He must deduce "some" information about the plain-text (what this means is to be defined later).

- **known plain-text attack:** adversary gets to see both plain-text and cipher-text $l-1$ times: $(m_1, E(m_1, s))$, $(m_2, E(m_2, s))$,...,$(m_k, E(m_{l-1}, s))$ and then he has to deduce "some" information about some "new" message m_l given only $E(m_l, s)$.

- **adaptive chosen plain-text attack:** adversary chooses m_1 and then gets to see $E(m_1, s)$, then (based on what he seen) he chooses m_2 and gets to see $E(m_2, s)$, and so on for $l-1$ times. Then, he has to deduce "some" information about a "new" message m_l, given only $E(m_l, s)$.

- **chosen cipher-text attack**: adversary chooses arbitrary message y_1 and then gets to see $D(y_1, s)$, then (based on what he seen) he chooses y_2 and gets to see $D(y_2, s)$ and so on for $l-1$ times, and then he has to deduce "some" information about some "new" m_l, given $E(m_l, s)$.

- **active/Byzantine attacks** where adversary not only listens, but can also change the cipher-text in an arbitrary manner in order to learn how to "break" some future message.

In all these attacks, we assume that adversary knows algorithms E and D, but does not know s. We will see (among other things) how these attacks (and there combinations) could be dealt with.

As an additional remark, it should be pointed out that even though a lot of initial work in cryptography was about secure communication, (and this is where we will also going to start) it now evolved into a mature field which deals with issues of interplay between privacy and computation in general, including secure two-party and multi-party protocols, general partial-information games and various security considerations.

2 Complexity Background

2.1 Familiar Complexity Classes

As a review, recall the following definitions[2]:

- P is the set of all languages L decidable in polynomial time.
- NP is the set of all languages L decidable in nondeterministic polynomial time. In other words, it is a class of languages which can be **verified** in polynomial time: imagine a game between two players a Prover and a Verifier. Verifier is a polynomial-time machine. Prover has unbounded computational resources. We say that a language L is in NP, if whenever $x \in L$, prover can send to the verifier some short (i.e. polynomial-length) message, such that if $x \in L$ verifier always excepts, and if $x \notin L$ prover can never "cheat" the verifier and make him except (i.e. there is no message that Prover can send that will convince verifier.)
- RP is the set of all languages L which are decidable (with 1-sided error) by a machine which runs in polynomial time and can flip coins: whenever $x \in L$ machines accepts with probability greater then a half (where probability is token over coin-flips of the machine) and whenever $x \notin L$, machine must always reject.
- BPP is the set of all languages L which are decidable (with 2-sided error) by a machine which runs in polynomial time and can flip coins: whenever $x \in L$ machine accepts with probability greater then $\frac{2}{3}$; whenever $x \notin L$ machine must accept with probability less then a $\frac{1}{3}$ (again probability is taken over coin-flips of the machine).

2.2 FACT: $P \subseteq RP \subseteq NP$

It is obvious that $P \subseteq RP$, since any machine which flips coins can simply ignore its coins and work deterministically. Proving $RP \subseteq NP$ involves the following. Suppose $L \in RP$, so there is some machine M which computes whether $x \in L$ but is sometimes incorrect when it says that the answer is false. For this machine, a sequence of coin flips which causes it to answer yes is a witness to x being in L, and we know there there is always such a witness.5 On the other hand, if $x \notin L$, such witness does not exist. Thus, $L \in NP$.

2.3 Reducing error probability for RP

We are interested in machines which have lower error rates than those necessary for the definitions of RP and BPP. We would like error rates to be less than $2^{-P(n)}$ where P is some polynomial and n is the length of the input string. We now discuss methods for taking machines satisfying the definitions of RP and BPP and reducing their error rates.

Suppose we have some $L \in RP$ decided by a machine M which says correctly that $x \in L$ only with probability $\epsilon(n)$. This means that the error probability when $x \in L$ is at most $1 - \epsilon(n)$. We can construct a new machine M' out of M by having this new machine run old machine M some $k(n)$ (each time using new independent coin-tosses) and accepting iff it is accepted by old M at

[2] Scribe notes taken by: Jay Lorch, August 25, 1994

least once. As long as $k(n)$ is polynomial in n, this new machine M' still runs in polynomial time, but it will in general have a lower probability of error. Specifically,

$$P(M' \text{ makes a mistake} \mid x \in L) \leq [1 - \epsilon(n)]^{k(n)} \leq e^{-k(n)\epsilon(n)},$$

where we have used the fact that $e^z \geq z + 1$. If $\epsilon(n) = 1/2$, as in the definition of RP, we can make $k(n) = 2P(n)$ to get the error bounded by $e^{-P(n)}$. Even if $\epsilon(n)$ is some function that decreases as the reciprocal of a polynomial in n, such as $1/n^3$, we can set $k(n)$ to some polynomial exceeding $P(n)/\epsilon(n)$ in order to bound the error the way we want.

2.4 Reducing error probability for BPP

Now, suppose we have some $L \in BPP$ decided by a machine M whose error probability is a $1/3$ (as in the original definition of BPP). We construct a machine M' which simply runs M using independent coin-tosses for each run $k(n)$ times and accepts iff M at least half the time (i.e. take the majority vote). As long as $k(n)$ is polynomial in n, this new machine still satisfies the requirements of BPP, but will have, in general, a lower error rate than M. We now compute a bound on this error rate. We need the following

> *Chernoff bound:* Let $X_1, X_2, \ldots, X_m$ be independent $0, 1$ random variables with identical probability distribution. Let $X = X_1 + X_2 + \ldots + X_m$. Then
>
> $$Pr\left(X \geq (1+\beta)E(X)\right) < e^{-\frac{1}{2}\beta^2 E(X)}.$$

Now, let us use this to bound the probability of error in M' above. In particular, let $X_i = 1$ if M makes a mistake on the i'th run. Then from the definition of BPP we know that $Pr(X_i = 1) \leq \frac{1}{3}$, where probability is taken over coin-flips of the machine. Lets be pessimistic and assume that it is exactly a third, then $E(X) = \frac{k(n)}{3}$ We know that M' makes an incorrect decision if the majority of $k(n)$ runs is wrong: $\sum_{i=1}^{k(n)} X_i \geq \frac{k(n)}{2}$. If we set $\beta = 1/2$ and $m = k(n)$ and plug it into the above bound we get:

$$Pr\left(\sum_{i=1}^{k(n)} X_i \geq \frac{3}{2} \cdot \frac{k(n)}{3}\right) < e^{-\frac{k(n)}{24}}.$$

But, the left-hand side of this inequality is just the probability that M' will answer incorrectly. So, if we set $k(n) = 24P(n)$, we have an error probability for our new machine bounded by $e^{-P(n)}$, for any polynomial $P(n)$.

2.5 Non-uniform Polynomial Time

We now shift to a different tack. Suppose we have an adversary who can construct a special circuit to attack those of your encrypted messages which have a certain length. We give this adversary supernatural power in order to construct such a circuit for each input length, but require that this circuit is of polynomial size (i.e. allow only polynomial time to crack your code.) This kind of attack is more devious than a general attack which requires the same machine to be used for all input lengths, and we would like a way to model it.

Thus, we define a class of languages called P/poly as follows. $L \in P$/poly if there exist machines $M_1, M_2, \ldots$ and a polynomial $Q(n)$ such that:

- The description of M_i has less than $Q(i)$ bits.
- The running time of M_i is less than $Q(i)$.
- $\forall n \forall x \in \{0,1\}^n$, $M_n(x) =$"yes" iff $x \in L$.

Another way to think about this class of languages is that they are decided by a single machine in polynomial time which is always given some advice, but this advice must be polynomial in length and must depend only on the input length (i.e. for every input of the same length the advice is the same).

2.6 Power of P/poly

What the relationship between P/poly and NP? It was once suggested (INCORRECTLY!) that $P/\text{poly} \subseteq NP$, and the following bogus proof for this claim was given. Suppose there is an $L \in P/\text{poly}$ decided by a machine M which takes advice as described in the alternate definition above. Then, we consider this advice to be a witness, and construct a new NP machine: guess an advice which makes the machine accept on input x. If there is such an advice, accept, else reject.

What's wrong? The problem with this "proof" is that M can not actually "trust" this advice; if the advice is wrong, it will come to the wrong conclusion! And there is no way to check if the advice is good for *all* inputs, as this be not an NP statement.

To give an idea of the power of P/poly we will in fact show that it can recognize undecidable languages. Consider some standard enumeration of the Turing machines $M_1, M_2, \ldots$. Now consider the following language L: $x \in L$ iff machine number $|x|$ halts. (i.e. the *length* of x indicates the number of the machine in the standard enumeration above). Since we are allowed magic advice for each input length, the magic advice could tell us which machines halt and which do not in the above enumeration. But this is undecidable... so there are very hard languages which can be recognized in P/poly.

2.7 $BPP \subseteq P$/poly

We will now further show the power of P/poly by proving that $BPP \subseteq P/\text{poly}$ (i.e. in the class P/poly we do not have to flip coins in order to do everything that BPP can do!) This proof is due to Adelmann. Suppose $L \in BPP$, so it has a deciding machine M which flips at most $r(n)$ coins and has an error with probability less than $2^{-(n+1)}$. (Note that this low error rate can be obtained by methods described earlier if the original error rate is higher.) Consider a matrix with 2^n rows and $2^{r(n)}$ columns. The element at row x and column c is 1 if M will make a mistake given input x and coin flip sequence c, and is 0 otherwise. Since the probability of making a mistake on any given input is less than $2^{-(n+1)}$, the total number of 1's in each row is less than $2^{-(n+1)}2^{-r(n)}$. What is the total number of 1's in the matrix? It is the number of ones in each row, times the number of rows: $(2^{-(n+1)}2^{-r(n)})2^n$. So there are at most $2^{r(n)-1}$ 1's in the entire matrix. Observing that there are more columns ($2^{r(n)}$) than there are total number of 1's in the matrix (at most t$2^{r(n)-1}$), the pigeonhole principle tells us that there must be a column containing no 1's. This column corresponds to a coin flip sequence which, if it were hard-wired into M, would cause it to never make a mistake on inputs of length n. We call this hard-wired machine M_n, and observe that the sequence of machines $M_1, M_2, \ldots$ satisfies the conditions for $L \in P/\text{poly}$.

2.8 Terminology

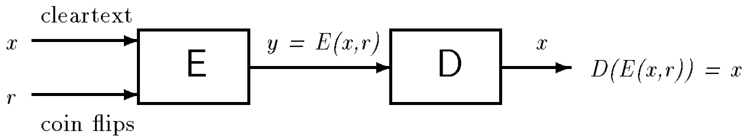
The class of RP (and co-RP) machines corresponds to what are called Monte Carlo algorithms, which are defined as algorithms which always finish in polynomial time and are correct with one-sided error bounded above by $1/2 - \epsilon$ (where 1-sided error is either for $x \in L$ or for $x \notin L$, but not both.) The class of BPP machines corresponds to what are called Atlantic City algorithms, which are defined as algorithms which always finish in polynomial time and are correct with two-sided error bounded above by $1/2 - \epsilon$. Another type of algorithm is the Las Vegas algorithm, which is always correct but only has an *expected* running time which is polynomial. Algorithms like this, which flip coins to decide whether or not to continue computing, will be discussed later, but for today we are finished.

3 Introduction to One-Way Functions

3.1 Overview

This lecture[3] is devoted to the definition of one way functions. While informally the idea should be quite clear,(i.e., a function which is "easy to compute" but "hard to invert") the precise formal definition is quite tricky and easy to get wrong.

As a motivating example, suppose we wish to construct a cryptosystem consisting of an encryption E and a decryption D, both of which are poly-time. The encryption takes a clear-text message x and some random bits r, and gives $y = \mathrm{E}(x, r)$. A polynomial-time adversary has access only to the cipher-text y. For the cryptosystem to be secure, it should be hard for the adversary to recover the clear-text, i.e. a poly-time adversary who is given $E(x, r)$ should not be able to figure out x.



This brings up two questions: what assumptions do we need to design that such a cryptosystem, and what is meant by the security of the cryptosystem? In this lecture we will answer only the first question.

3.2 Worst-case complexity: P and NP

It is clear that if one can "invert" a function $E(x, r)$ (i.e, given y of polynomial length, and description of algorithm $E(\cdot,\cdot)$, find x and r such that $E(x, r) = y$) then the cryptosystem is not secure. However, this is an NP problem, hence if $P = NP$, the cryptosystem can always be broken. So, let us assume that $P \neq NP$. Notice that usually, P and NP refer to languages, but this formulation is equivalent to expressing the machine in terms of functions. To show this, we will reformulate the Hamiltonian cycle problem, a well known NP-complete problem:

$$f(G, H) = \begin{cases} G & \text{if } H \text{ is a Hamiltonian cycle} \\ 000\ldots0 & \text{(number of 0's} = |G|\text{) otherwise} \end{cases}$$

If $P \neq NP$, then f^{-1} is hard for infinitely many (G, H) pairs. We remark that the reason $P \neq NP$ implies *infinite* (and not just finite) number of hard (G, H) instances is that otherwise there exists an algorithm which contains answers to this finite number of pairs as a look-up table, and for this algorithm everything is easy. So, we must simply assume that P≠NP. Is this enough, then, to have a one way function? No, not yet.

3.3 Worst-case complexity: BPP and NP

One of the problems is that security of a cryptosystem is not defined "fairly" if the encryption E is supplied with coin-flips (random string r) but the adversary is not allowed to flip coins. So,

[3]Scribe notes taken by: Raph Levien, August 30, 1994

to be fair, we must allow any polynomial-time adversary to be able to flip coins, and be wrong once in a while, i.e. we view our adversary as a BPP (bounded probabilistic poly-time) machine. We choose BPP over, say, RP, because BPP is a larger class, and thus can potentially do more, then RP, and we wish to say that "no matter" what the polynomially -bounded adversary does he can not invert E.

Now do we have one-way functions? Not yet. For one thing, we need a new assumption: that there are languages in NP which are not in BPP, i.e. that $NP - BPP$ is not empty. (By the way, since BPP contains P, if we assume that $NP - BPP$ is not empty, then clearly $NP - P$ is not empty, and hence $NP \neq P$. Thus, the assumption that $NP - BPP$ is not empty is stronger then $P \neq NP$. So, lets assume that $NP - BPP$ is not empty. Hence, there exists a function f in NP such that any BPP machine fails on infinitely inputs.

3.4 Negligible, and Noticeable functions

What does it mean for the BPP machine to fail? Let us suppose that the machine has a probability of $.5 + \epsilon(n)$ of guessing correctly. We already know that if $\epsilon(n)$ is greater than the reciprocal of some polynomial, then it is possible to amplify the probability (by running the machine a polynomial number of times, and taking the majority vote) to any practical degree of certainty (specifically, a failure rate less than $e^{-poly(n)}$). So, we want to define the notion of *negligible* probability, one which can not be amplified in polynomial time. Informally, $\epsilon(n)$ is negligible if it is smaller than the reciprocal of any polynomial. Formally,

Definition 3.1 Let $g : N \rightarrow N$. We say that g is **negligible** if $\forall c \; \exists N_c$ such that $\forall n > N_c$

$$g(n) < \frac{1}{n^{-c}}$$

The N_c term is there to make the definition not depend on constant factors. Asymptotically, the degree of the polynomial will always overcome the constant factor. The notion of a negligible functions should be robust against any polynomial-time machine. Indeed,

Fact 3.2 For all negligible functions $g(n)$ and all polynomials $p(n)$, their product $g(n)p(n)$ is negligible.

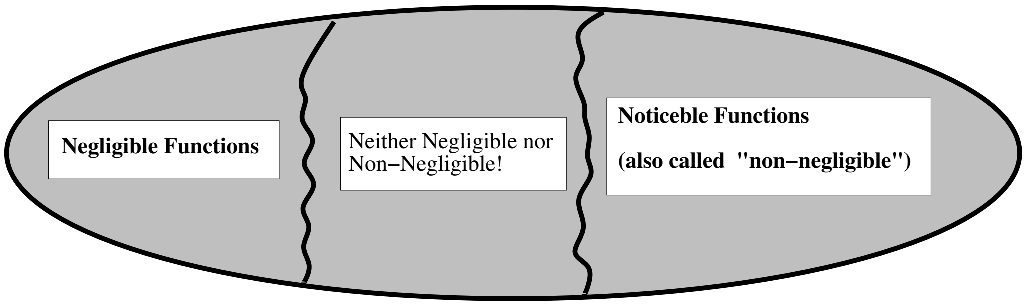
To see this, notice that the above statement is equivalent to: $\forall c_1 c_2 \; \exists N_{c_1 c_2}$ such that $g(n)n^{c_2} < n^{-c_1}$. Dividing both sides of the inequality by n^{c_2}, we get $g(n) < n^{-(c_1+c_2)}$. Thus, by taking $N_{c_1 c_2} = N_{(c_1+c_2)}$, the fact becomes equivalent to the negligibility of g, which was given. □

The "opposite" of negligible functions are *noticeable* functions, those that are greater than the reciprocal of some polynomial. Formally $g(n)$ is noticeable if:

Definition 3.3 Let $g : N \rightarrow N$. We say that $g(n)$ is **noticeable** if $\exists c$ and $\exists N$ such that $\forall n > N_c$

$$g(n) > \frac{1}{n^{-c}}$$

A word of warning is appropriate here. There exist some functions that are neither negligible nor noticeable. The world looks like this:



For example, there are functions which on an infinite sequence of inputs is negligible and on a complement infinite sequence is noticeable, where the two sequences interleave. The composite sequence is neither negligible not noticeable.

3.5 Average Case Complexity

So lets assume that there exists a language L which is in NP and not in BPP. This implies that there exists a function f such that for any probabilistic polynomial time machine M there exists infinitely-many inputs $x_1, x_2, \ldots$ such that probability (over coin-flips of M) that M inverts on $f(x_i)$ is negligible.

Is this assumption (i.e., for infinitely many x probability of successful inversion is negligible) enough to characterize a one way function? No, not quite. The problem is that for cryptography we need to efficiently find hard instances (in order to grantee the security of the cryptosystem). While there might be infinitely many instances for which f^{-1} is hard, it is not clear how to find this hard instances. (Moreover, the assumption that infinitely many hard instances exist does not even guarantee that they exist for every input length.) What we really want to say that not only hard instances exist, but they are also easy to find. One way is to say that "most" instances are hard.

An attempt to define this is to say that for most $f(x)$, when $|x|$ is sufficiently large, it is "hard" to invert f. In other words, if we pick y at random, and ask ourself to find $f^{-1}(y)$, this is hard "most" of the time (This was formalized by Levin as an average case analog of P vs. NP question.) Let us assume that such problems exists, i.e. ave-P $\neq$ ave-NP. Is this sufficient? It is not clear, since it could be the case that whenever we pick y at random and try to find $f^{-1}(y)$ it is hard, but whenever we pick x at random and ask our enemy to invert $f^{-1}(f(x))$ it is easy! The reason for this is that the distribution of $f(x)$ if we start from the uniform distribution on x could be very far from uniform. Thus, we need to assume not only that ave-P $\neq$ ave-NP but also that there are functions which are hard on the uniform distribution of the inputs!

3.6 One-way functions

To summarize, the type of a polynomial-time computable function $f(\cdot)$ we really need could be best described as a game between a Player (a probabilistic polynomial-time machine) and an Adversary (a probabilistic polynomial-time machine), as follows:

- Step 1: Player pick an input length n for a one-way function, which he hopes is "large enough". Then Player picks x at random, $|x| = n$, and computes $y = f(x)$, and gives the result y to Adversary.
- Step 2: Adversary tries to compute $f^{-1}(y)$ (for some polynomial amount of time in the length of $|f(x)|$) and sends his guess z back to Player.
- Outcome: Adversary wins if $f(x) = f(z)$, otherwise Player wins. We define f to be a one way function if the probability of all probabilistic polynomial-time adversaries to win is negligible, for n big enough.

Note that the size of the input and output of f must be polynomially related, otherwise, if (say) $|f(x)| = \log(|x|)$, then Player is cheating, because it gets to do exponentially more work (as a function of its input length) than the Adversary. Tying all this together, we arrive at the following definition:

Definition 3.4 f is a one-way function if:

1. f can be computed in (deterministic) polynomial time.
2. f is "hard to invert" : $\forall c$ $\forall$ probabilistic polynomial-time A, there exists $N_{c,A}$ such that $\forall n > N_{c,A}$

$$Prob\left[A(f(x)) \in f^{-1}(f(x))\right] < \frac{1}{n^c}$$

   where $|x|$=n, and probability is taken over x and coin-flips of the adversary.
3. f has polynomially-related I/O: $\exists c_1, c_2$ such that $|x|^{c_1} \leq |f(x)| \leq |x|^{c2}$.

We remark that in requirement 3, the second inequality is not really necessary: since f is computable in polynomial time, it can not output more the some fixed polynomial number of bits. We decided to explicitly state this anyway. Another remark is that an adversary can always invert on a negligible set of the inputs, buy simply "guessing" what the inputs are.

We will see that with the above definition of a one-way function, it is possible to build a good encryption scheme (which was our motivating example), we will see how to this can be done in the near future.

3.7 One-way functions and NP

The P=NP question is one of worst-case complexity. But in general, an NP-hardness result says nothing about the *average* hardness of a particular problem. For cryptography, we wish to say that average instances are hard, yet easy to generate. For this to hold, we would need to have ave-P $\neq$ ave-NP, which would imply the existence of hard *unsolved* problems, or problems for a friend: "Pick y at random. Try to find $f^{-1}(y)$." But one-way functions give us even more, in that they guarantee the existence of hard *solved* problems, or problems for an enemy: "Pick x at random. Compute $f(x) = y$. Ask enemy to find $f^{-1}(y)$." To conclude, note that the existence of one-way functions guarantees that ave-P $\neq$ ave-NP, which in turn guarantees P $\neq$ NP. The reverse implications are not known, however.

4 In-depth look at One-way Functions

4.1 Introduction

Today's lecture[4] will concern one-way functions and various definitions of them. Our objective is to show several equivalent definitions and to prove that so-called "weak" one-way functions exist iff there exist so-called "strong" one-way function.

4.2 One-way Functions

Recall the following definition from lecture 3, where "PPT" stands for probabilistic polynomial time, and "A inverts $f(x)$" denotes $A(f(x)) \in \{f^{-1}(f(x))\}$:

Definition 4.5 A function f is said to be a **uniform** strong one-way function if the following conditions hold:

1. f is polynomial-time computable.
2. f is hard to invert for a random input: $\forall c > 0\ \forall A \in$ PPT $\exists N_c\ s.t.\ \forall n > N_c$:

$$Pr_{\{x,\omega\}}[A \text{ inverts } f(x)] < \frac{1}{n^c}$$

   where $|x| = n$ and ω are coin-flips of the probabilistic algorithm A.

3. Input/output length of f is polynomially related: $\exists \epsilon, c\ s.t.\ |x|^\epsilon < |f(x)| < |x|^c$.

Such a one-way function f is said to be *uniform*, as opposed to a *non-uniform* one-way function. A non-uniform one-way function is defined exactly as above, except that the adversary is formulated not as a PPT machine, but a family of polynomial-size circuits:

Definition 4.6 A function f is said to be a **non-uniform** strong one-way function if the following conditions hold:

1. f is polynomial-time computable.
2. f is hard to invert: $\forall c > 0\ \forall$ non-uniform polynomial-size families A of circuits $\exists N_c\ s.t.\ \forall n > N_c$:

$$Pr_{x,|x|=n}[A \text{ inverts } f(x)] < \frac{1}{n^c}$$

3. Input/output length of f is polynomially related: $\exists \epsilon, c\ s.t.\ |x|^\epsilon < |f(x)| < |x|^c$.

[4]Scribe notes taken by: Ari Juels, September 2, 1994

REMARK : Notice the in the non-uniform definition, the probability is only over x.

The following fact shows that this last condition is weaker than the non-invertability condition imposed on uniform circuits. In consequence, results proven about the existence of uniform one-way functions will also pertain to non-uniform one-way functions.

Fact 4.7 If f is a non-uniform one-way function, then f is also a uniform one-way function.

Proof: We will prove the contrapositive. That is, we will show that if f does not satisfy the definition of the uniform one-way function, then it can not satisfy definition of the non-uniform one-way function. Suppose that f is not a uniform one-way function. Then there exists a constant c, and an adversary A such that for an infinite number of integers n, for all strings x of length n, $Pr_{\{x,\omega\}}[A \text{ inverts } f(x)] > \frac{1}{n^c}$ (where ω are the coin-flips of the adversary. Our objective is to find a poly-size collection of circuits A' to substitute our adversary A. But A is probabilistic. We have to somehow "hardwire" coin-flips ω of A to get an A'.

Define $\epsilon(n) = \frac{1}{n^c}$. Define $GOOD = \{x \mid Pr_\omega[A \text{ inverts } f(x)] > \frac{\epsilon(n)}{2}\}$. By conditioning on whether $x \in GOOD$:

$$\begin{aligned} Pr_{x,\omega}[A \text{ inverts } f(x)] &= Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \in GOOD] \cdot Pr_x[x \in GOOD] \\ &\quad + Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \notin GOOD] \cdot Pr_x[x \notin GOOD] \end{aligned}$$

Therefore:

$$\begin{aligned} Pr_x[x \in GOOD] &= \frac{Pr_{x,\omega}[A \text{ inverts } f(x)\,] - Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \notin GOOD] Pr_x[x \notin GOOD]}{Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \in GOOD]} \\ &> Pr_{x,\omega}[A \text{ inverts } f(x)\,] - Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \notin GOOD] Pr_x[x \notin GOOD] \\ &> Pr_{x,\omega}[A \text{ inverts } f(x)\,] - Pr_{x,\omega}[A \text{ inverts } f(x) \mid x \notin GOOD] \\ &\geq \epsilon(n) - \frac{\epsilon(n)}{2} \\ &= \frac{\epsilon(n)}{2} \end{aligned}$$

By standard techniques of repeated coin flips, we can boost $Pr_\omega[A \text{ inverts } f(x) | x \in GOOD]$ to $1 - \frac{1}{2^{poly(n)}}$. Analogous to the proof that $BPP \in P/poly$ we can show that there exist coin-flips γ for A such that A correctly inverts *all* elements of $GOOD$ on γ. Therefore, by hardwiring γ, we can build a circuit family A' which inverts at least $\frac{\epsilon(n)}{2}$ of all strings x. Therefore f is not a non-uniform one-way function. ■

The following fact will also prove useful in many contexts in the future, and we will often assume it implicitly:

Fact 4.8 *If there exists a one-way function, then there exists a length-preserving one-way function.*

Proof: Suppose that f is a one-way function. By definition, there exists a c such that for any n and x, $|f(x)| \leq n^c$. We can therefore define a function g such that $g(x) = f(x)\ 0\ 1^{|x|^c - |f(x)| - 1}$ – i.e., $f(x)$ followed by a 0 and then $|x|^c - |f(x)| - 1$ ones. The function g is one-way, since the existence of a successful polynomial-time adversary A_g for g implies the existence of such an adversary A_f for f: by definition of a one-way function, $|x|^\epsilon < |f(x)| < |x|^c$ for some c, ϵ, this A_f need only apply A_g to $f(x)\ 0\ 1^m$ for all m, $0 \geq m < |x|^c - |x|^\epsilon - 1$: one of these paddings[5] will yield $f(x)\ 0\ 1^{|x|^c - |f(x)|}$, and therefore guarantee the success of A_g with sufficient probability.[6]

Using this intermediate function g, we can construct length-preserving function h. Given input x, the function h computes x', a string of leading bits of x such that $|x'|^c = |x|$, and outputs $g(x')$. (Inexactitudes in length can be compensated for by appropriate padding[7].) Since $|g(x')| = |x'|^c = |x|$, h is length-preserving.

It now remains to show that h is a one-way function. Suppose not. Then there exists an adversary A which can invert h with probability at least $\frac{1}{n^q}$ for some q. This implies the existence of an adversary A' which can invert g: A' computes $A(f(x))$ and takes only the first $|A(f(x))|^{\frac{1}{c}}$ bits of the result. Adversary A' inverts g with success at least $\frac{1}{n^{qc}} = \frac{1}{poly}$, a contradiction. Therefore h is one-way. ■

4.3 Weak One-way Functions

A uniform one-way function as defined above, against which any adversary fails most of the time, can be regarded as a *strong* one-way function. A *weak* one-way function is one in which any adversary fails at least some of the time.

A Motivating Example (due to Oded Goldreich): The problem of factoring the product of two primes lies in the intersection of NP and co-NP. To the best of our knowledge, it is not polynomial-time computable. Assuming that it is a sufficiently hard problem for large primes of the same length, we can construct the following weak 1-way function: Consider the function f which takes two n-bit numbers x_1 and x_2 at random and outputs $x_1 \times x_2$. The probability that an arbitrary $n - bit$ number is prime is about $\frac{1}{n}$. Therefore the probability that $f(x)$ is the product of two primes is about $\frac{1}{n^2}$.Hence, if factoring numbers composed of two primes of equal length is hard, this function is hard hard to invert with probability greater then $\frac{1}{n^2}$.

In other words, a *weak* one-way function is a function f which satisfies the first and third conditions of the definition above for a strong one-way function, but for which the second condition is replaced with the following:

[5] the reason we have to search through all m is that given $f(x)$ we do not know what length of $|x|$ is.

[6] The separating 0 in the padding may seem unnecessary at first glance. If we padded entirely with 1s, though, then we couldn't insure that g would be one-way. In particular, we could construct an f' such that on input xy, where $|x| = |y|$, $f'(xy) = xy$ if y consists entirely of 1s and $f'(xy) = f(x)$ otherwise. If f is a one-way function, f' is also one-way. On the other hand, if we constructed g on f', then g would not necessarily be one-way, for the following reason. If we chose c large enough, then $g(z)$ would always have the form xy, where y consists entirely of 1s. Therefore, xy would always be a correct inverse of $g(z)$, so g would not be one-way.

[7] Note that to compute the length of $|x'|$ given $|x|$ and $|c|$ could be done in $|x|^2$ steps, but the rest of the computation of $g(x')$ takes time **linear** in $|x|$, hence our new function g is not only length preserving, but also runs in quadratic time!

2'. $\exists\ c > 0\ \forall A \in \mathsf{PPT}\ \exists N_c\ s.t.\ \forall n > N_c$:

$$Pr_{x,\omega}[A \text{ does not invert} f(x)] > \frac{1}{n^c}$$

where $|x| = n$; ω are coin-flips of A; and "A does not invert $f(x)$" means $A(f(x)) \notin \{f^{-1}(f(x))\}$.

4.4 Super-Weak One-way Functions

We can weaken the conditions on a one-way function still further by changing the quantifiers in the above definition of a weak one-way function. A *super-weak* one-way function has the following condition:

2". $\forall A \in \mathsf{PPT}\ \exists c > 0\ s.t.\ \exists N_c\ s.t.\ \forall n, n > N_c$:

$$Pr_{x,\omega}[A \text{ does not invert } f(x)] > \frac{1}{n^c}$$

where $|x| = n$; ω are coin-flips of A; and "A does not invert $f(x)$" means $A(f(x)) \notin \{f^{-1}(f(x))\}$.

It follows from work of Levin (1984,85) that super-weak functions exist if and only if weak one-way functions exist. He shows that there exists an "optimal" (within a multiplicative factor) inverting adversary. This adversary could be considered instead of all other adversaries, and hence we can take a c for this adversary to work for all other adversaries as well. As this is a fairly involved argument, we will not deal with "super-weak" one-way functions here.

4.5 Weak and Strong One-way Functions

We will now prove the main result in today's lecture, due to Yao:

Theorem 4.9 [Yao]: There exists a weak one-way function if and only if there exists a strong one-way function.

Proof:
First, let us show a trivial direction that that the existence of a strong one-way function implies that of a weak one-way function: condition 2 of a strong one-way function can be re-written as follows: $\forall c > 0\ \forall A \in \text{PPT}\ \exists N_c\ s.t.\ \forall n > N_c$:

$$Pr_{x,\omega}[A \text{ does not invert } f(x)] > 1 - \frac{1}{n^c} > \frac{1}{n^c}$$

which implies condition $2'$ of a weak one-way function.

We now prove the converse. That is, given a weak one-way function f_0, we will construct a strong one-way function f_1. We will demonstrate that f_1 is a strong one-way function by contradiction: we assume an adversary A_1 for f_1 and then demonstrate an effective adversary A_0 for f_0. W.l.o.g. we can assume that f_0 is length-preserving and maps m bits to m bits. Definition $2'$ of a weak one-way function f_0 can be restated as:

2'. $\exists\, c_{f_0} > 0\ \forall A_0 \in \mathsf{PPT}\ \exists M_{c_{f_0}}\ s.t.\ \forall m > M_{c_{f_0}}$:

$$Pr_{\{x,\omega\}}[A_0 \text{ inverts } f_0(x)] \leq 1 - \frac{1}{m^{c_{f_0}}} = 1 - \epsilon_0(m)$$

where $|x| = m$, ω are coin-flips of A, and $\epsilon_0(m) \stackrel{\text{def}}{=} \frac{1}{m^{c_{f_0}}}$.

To construct f_1, we amplify the "hardness" of weak one-way function f_0 by applying f_0 in parallel $q \stackrel{\text{def}}{=} \frac{2m}{\epsilon_0(m)}$ times:

$$f_1(x_1, \ldots, x_q) \stackrel{\text{def}}{=} f_0(x_1), \ldots, f_0(x_q).$$

where each x_i, $1 \leq i \leq q$ is uniformly and independently chosen m-bit input to f_0. Notice that our f_1 maps $n = \frac{2m^2}{\epsilon_0(m)}$ bits to n bits. We claim that f_1 is a strong one-way function. The proof is by contradiction. Suppose f_1 is not a strong one-way function. Then $\exists A_1$, $\exists c$ s.t. for infinitely many input length n,

$$Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x})] > \frac{1}{n^c} \stackrel{\text{def}}{=} \epsilon_1(n) \stackrel{\text{def}}{=} \epsilon_2(m)$$

Notice that we can redefine $\epsilon_1(n)$ in terms of $\epsilon_2(m)$ since m and n are polynomially related. If we can show how to construct A_0 (using above A_1 as a subroutine) such that A_0 will invert f_0 with probability (over x and ω) greater then $1 - \epsilon_0(m)$ we will achieve a contradiction with weak one-wayness of f_0. Our algorithm $A_0(f_0(x))$ is as follows:

<u>Algorithm $A_0(y)$:</u>
repeat procedure $Q(y)$ at most $\frac{4m^2}{\epsilon_2(m)\epsilon_0(m)}$ times;
  stop whenever $Q(y)$ succeeds and output $f_0^{-1}(y)$,
  otherwise output "fail to invert".

<u>Procedure $Q(y)$:</u>
for i from 1 to $q = \frac{2m}{\epsilon_0(m)}$ do:
  STEP1: pick $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_q$
    (where each x_j is independently chosen m-bit number)
  STEP2: call $A_1(f_0(x_0), \ldots, f_0(x_{i-1}), y, f_0(x_{i+1}), \ldots, f_0(x_q))$
    (procedure $Q(y)$ *succeeds* if A_1 above inverts)

We must estimate the success probability of $A_0(f_0(x))$, where the probability is over x and coin-flips ω of A_0. Define x (of length m) to be BAD if

$$Pr_\omega(Q(f(x)) \text{ succeeds }) < \frac{\epsilon_2(m)\epsilon_0(m)}{4m}$$

We claim that

$$Pr_x[x \text{ is BAD}] < \frac{\epsilon_0(m)}{2}$$

To show this we assume (towards the contradiction) that $Pr_x[x \text{ is BAD }] \geq \frac{\epsilon_0(m)}{2}$. Then

$$\begin{aligned}
Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x})] &= Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x}) \mid \text{ some } x_i \in \text{BAD}] \cdot Pr_{\vec{x}}[\text{ some } x_i \in \text{BAD}] \\
&\quad + Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x}) \mid \forall i,\ x_i \notin \text{BAD}] \cdot Pr_{\vec{x}}[\forall i,\ x_i \notin \text{BAD}] \\
&\leq \sum_{i=1}^{\frac{2m}{\epsilon_0(m)}} \Big[Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x}) \mid x_i \in \text{BAD}] \cdot Pr_{\vec{x}}[x_i \in \text{BAD}] \Big] \\
&\quad + Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x}) \mid \forall i,\ x_i \notin \text{BAD}] \cdot Pr_{\vec{x}}[\forall i,\ x_i \notin \text{BAD}] \\
&\leq \frac{2m}{\epsilon_0 m} \left(\frac{\epsilon_2(m)\epsilon_0(m)}{4m} \right) \cdot 1 + 1 \cdot \left(1 - \frac{\epsilon_0(m)}{2} \right)^{\frac{2m}{\epsilon_0(m)}} \\
&\leq \frac{\epsilon_2(m)}{2} + e^{-m} \\
&< \epsilon_2(m)
\end{aligned}$$

But we assumed that $Pr_{\{\vec{x},\omega\}}[A_1 \text{ inverts } f_1(\vec{x})] \geq \epsilon_2(m)$ a contradiction. Hence we have shown that $Pr_x[x \text{ is BAD}] < \frac{\epsilon_0(m)}{2}$. We are now ready to estimate the failure probability of A_0, using the fact that we try procedure Q in case of failure a total of $\frac{4m^2}{\epsilon_2(m)\epsilon_0(m)}$ times:

$$\begin{aligned}
Pr_{\{x,\omega\}}[A_0 \text{ does not invert } f_0(x)] &= Pr_{\{x,\omega\}}[A_0 \text{ does not invert } f_0(x) \mid x \in \text{BAD}] \cdot Pr_x[x \in \text{BAD}] \\
&\quad + Pr_{\{x,\omega\}}[A_0 \text{ does not invert } f_0(x) \mid x \notin \text{BAD}] \cdot Pr_x[\ x \notin \text{BAD}] \\
&\leq 1 \cdot \frac{\epsilon_0(m)}{2} + \left(1 - \frac{\epsilon_2(m)\epsilon_0(m)}{4m} \right)^{\frac{4m^2}{\epsilon_2(m)\epsilon_0 m}} \cdot 1 \\
&\leq \frac{\epsilon_0(m)}{2} + e^{-m} \\
&< \epsilon_0(m)
\end{aligned}$$

Thus $Pr_{\{x,\omega\}}[A_0 \text{ inverts } f_0(x)] > 1 - \epsilon_0(m)$ contradicting the assumption that f_0 is a weak one-way function. ■

5 Hard-core Bits

5.1 Introduction to Hard-Core Bits

The first of these three lectures[8] is devoted to the motivation and definition of hard-core bits. Hard-core bits were defined by Blum and Micali in 1982. In formally, a hard-core bit $B(\cdot)$ of a one-way function $f(\cdot)$ is a bit which is as hard to compute as it is to invert f. Blum and Micali showed that a particular number theoretic function (which is believed to be one-way) has a hard-core bit. It was later shown that all (padded) one-way function have a hard-core bit. The next two lectures are devoted to presenting this proof (due to Goldreich and Levin 1989).

Motivating example (due to Manuel Blum): Consider the problem of gambling on the outcome of a random coin flip with an adversary over a telephone line. If you bet on heads and allow the adversary to flip the coin and then inform you of the outcome, he may cheat and say tails without even bothering to flip the coin. Now suppose that after losing quite a bit of money, you decide to play a more sophisticated game in which both you and the adversary select a random bit and you win if the XOR of the two bits is 1. Unfortunately, it is still unsafe to transmit your random bit in the clear to an un-trustworthy adversary, for your adversary can always cheat by claiming that it selected the same bit.

To keep from being swindled further, you decide on the following commitment protocol to play the game described above fairly. You begin by sending the adversary your bit in a locked safe, then the adversary sends you its bit in the clear, and finally you send the adversary the combination to the safe. Both of you then compute the XOR of the two bits certain that the other party had no unfair advantage playing the game. We use this analogy to motivate the idea that it may be possible to send a commitment of a secret bit to an adversary, without revealing any information as to the value of that bit. Our objective is to develop such a legitimate commitment protocol based on one-way functions.

Assume that we have a one-way function. One (unfair) strategy would be to commit to the third bit of x, then to transmit $f(x)$. The flaw with this strategy is that the player can cheat, since $f(x)$ might not have unique inverses. In particular, suppose $f(x)$ has inverses x_1 and x_2 such that the third bit of x_1 and x_2 differ. Then once the adversary presents its random bit in the clear, the player can choose to transmit either x_1 or x_2 to the adversary, and clearly will choose to transmit the one which results in a payoff.

What if we assume much more, i.e. that we have a 1-1, length-preserving one-way function? The player can no longer cheat in the manner described above, but the adversary may still be able to cheat. Just because $f(x)$ is hard to invert does not necessarily mean that any individual bit of $f(x)$ is hard to invert. As an example, suppose we have a one-way function $f(x)$ and another function $g(x) = g(b_1, b_2, b_3, x_4, x_5, \ldots, x_n) = b_1 b_2 b_3 f(x_4, x_5, \ldots, x_n)$. Now since f is one-way, g is also one-way, yet the three highest-order bits of $g(x)$ are simple to invert.

5.2 Definition of a hard-core bit

These examples motivate the following definition of a hard-core bit due to Blum and Micali. Intuitively, a hard-core bit is a bit associated with a one-way function which is as hard to

[8] Scribe notes taken by: John Byers, September 6,8,13 1994

determine as is inverting the one-way function.

Definition 5.10 A **hard-core bit** $B(\cdot)$ of a function $f(\cdot)$ is a boolean predicate such that:

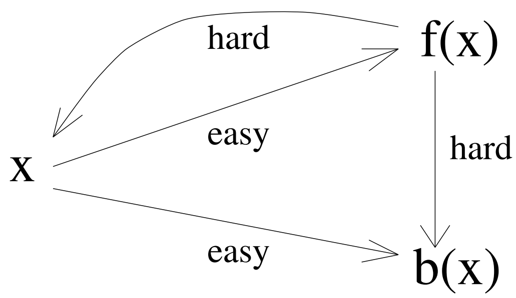
- $B(x)$ is easy to compute given x, i.e. computable in deterministic polytime
- Given $f(x)$, $B(x)$ is hard to guess better than at random:

  $\forall c \, \forall$ probabilistic poly-time A, there exists N_c such that $\forall n > N_c$

  $$\Pr_{\{x,\omega\}}[A(f(x)) = B(x)] < \frac{1}{2} + \frac{1}{n^c}$$

  where $|x| = n$, and probability is taken over x and coin-flips ω of A.

Pictorially, the relationship between a one-way function and a hard-core bit is as follows:



5.3 Does the existence of a hard-core bit $B(\cdot)$ for a function $f(\cdot)$ implies that f is a one-way function?

We note first that the existence of a hard-core bit for f does not necessarily imply that the corresponding one-way function is hard. As an example, the almost-identity function $I(b, x) = x$ has a hard-core bit b but is not hard-to-invert in the sense that we have defined in previous lectures. However, if no information is lost by the function f, then the existence of a hard-core bit guarantees the existence of a one-way function. We prove a somewhat weaker theorem below.

Theorem 5.11 If f is a permutation which has a hard-core bit, then f is a one-way function.

Proof: Assume f is not one-way. Then there exists a good inverter for f which correctly computes inverses with probability $q > \epsilon(n)$, where probability is taken over x and coin-flips of A. The predictor for the hard-core bit B first attempts to invert f using this good inversion strategy. If it succeeds in inverting f, it knows x, and can compute $B(x)$ in polynomial time. Otherwise, with probability $1 - q$, it fails to invert f, and flips a coin as its guess for $B(x)$. The predictor predicts B correctly with probability

$$q \cdot 1 + (1 - q) \cdot \frac{1}{2} = \frac{1}{2} + \frac{q}{2} \geq \frac{1}{2} + \frac{\epsilon(n)}{2}$$

Therefore f does not have a hard-core bit, proving the contrapositive. ∎

5.4 One-way functions have hard-core bits

The next two lectures are devoted to a proof of the following important theorem, first proved in 1989 by Goldreich and Levin, then simplified (by Venkatesan and Rackoff). It says that if $f_1(x)$ is a strong one-way function, then parity of a random subset of bits of x is a hard-core bit:

Theorem 5.12 [Goldreich, Levin] Let f_1 be a strong one-way function. Let $f_2(x,p) \stackrel{\text{def}}{=} (f_1(x), p)$, where $|x| = |p| = n$. Then

$$B(x,p) \stackrel{\text{def}}{=} \sum_{i=1}^{n} x_i p_i \bmod 2$$

is hard-core for f_2.

Notice that a random subset is chosen by choosing p at random. The hard-core bit of x is simply parity of a subset of bits of x, where the subset corresponds to all bits of x where corresponding bits of p are set to one.

5.4.1 Proof outline

The proof that $B(x,p)$ is a hard-core bit will be by contradiction. We begin by assuming that $B(x,p)$ is not a hard-core bit for f_2. That is:
$B(\cdot,\cdot)$ is not hard-core: $\exists A_B \in$ PPT, $\exists c$ such that for infinitely many n,

$$\Pr_{\{x,p,\omega\}}[A_B(f_2(x,p)) = B(x,p)] > \frac{1}{2} + \frac{1}{n^c} \stackrel{\text{def}}{=} \frac{1}{2} + \epsilon(n)$$

where A_B is probabilistic poly-time and probability is taken over x, p and coins ω of A_B

We want to show that we can invert f_2 with noticeable probability, proving that f_2 (and likewise f_1) is not a strong one-way function, i.e.:

f_2 is not a strong one-way function: $\exists A_{f_2} \in$ PPT, $\exists c$ such that for infinitely many n:

$$\Pr_{\{x,p,\omega\}}[A_{f_2} \text{ inverts } f_2(x,p)] > \frac{1}{n^c}$$

where A_{f_2} is probabilistic poly-time and probability is taken over x, p and coins of A_{f_2}

We will show how to construct A_{f_2} using A_B as s subroutine.

5.4.2 Preliminaries

First, let us recall some useful definitions and bounds. Recall that a set of random variables is *pairwise independent* if given the value of a single random variable, the probability distribution of any other random variable form the collection is not affected. That is,

Definition 5.13 Pairwise independence: A set of random variables $X_1, \ldots, X_n$ are pairwise independent if $\forall i \neq j$ and $\forall a, b$:

$$\Pr_{\{X_i,X_j\}}[X_i = a \wedge X_j = b] = \Pr_{X_i}[X_i = a] \cdot \Pr_{X_j}[X_j = b]$$

As an example of pairwise independence, consider distribution of three coins, taken uniformly from: $\{\mathsf{HHH},\mathsf{HTT},\mathsf{THT},\mathsf{TTH}\}$. It is easy to check that given an outcome of any one of the three coins, the outcome of any other (of the two remaining) coins is still uniformly distributed. Notice however, that the number of *sample points* is small (only 4). On the other hand, for total (i.e. three-wise) independence we need all 8 combinations.

We are sometimes interested in bounding tail probabilities of large deviations. In particular, recall a Chernoff bound which we already used:

Definition 5.14 Chernoff bound: Let $X_1, \ldots, X_m$ be (totally) independent $0/1$ random variables with common probability $0 < p < 1$, and let $S_m = X_1 + X_2 + \ldots X_m$. Then

$$\Pr_{\{X_1,\ldots,X_m\}}[|S_m - pm| > \delta m] \leq 2e^{-\frac{\delta^2 m}{2}}$$

Notice that as a function of m, the error-probability in Chernoff bound drops exponentially fast. In case of *pairwise independence* we have an analogous, Chebyshev bound. Like the Chernoff bound, the Chebyshev bound states that a sum of identically distributed 0/1 random variables deviates far from its mean with low probability which decreases with the number of trials (i.e. m). Unlike the Chernoff bound, in Chebyshev bound the trials need only be pairwise independent, but the probability drops off only polynomially (as opposed to exponentially) with respect to the number of trials.

Definition 5.15 Chebyshev bound: Let $X_1, \ldots, X_m$ be pairwise independent 0/1 random variables with common probability $0 < p < 1$, and let $S_m = X_1 + X_2 + \ldots X_m$. Then

$$\Pr_{\{X_1,\ldots,X_m\}}[|S_m - pm| > \delta m] \leq \frac{1}{4\delta^2 m}$$

We also implicitly used before a union bound, which simply states that:

Definition 5.16 Union Bound: For any two events A and B (which need not be independent), the

$$\Pr[A \bigcup B] \leq \Pr[A] + \Pr[B]$$

5.4.3 Two warmup proofs

To motivate the direction we will be heading for in the full proof, we first consider two scenarios in which the adversary (on input $f_1(x)$ and p can guess $B(x, p$ with probability much greater then a half.

In the following two warmup proofs, we use the following notation to (hopefully) clarify the presentation of the results. Given a string x, we use x^i to denote the string x with the ith bit flipped. We use array notation, $x[j]$, to denote the jth bit of x. Also, when referring to a string in the set of strings P, we use p_k to denote the kth string in the set.

The super-easy proof: Suppose the adversary A_B is able to guess the hard-core bit $B(x,p)$ given $f_2(x,p)$ with probability 1. Then A_B can compute x bit-by-bit in the following manner. To compute $x[i]$, the ith bit of x, choose a random string p, and construct p^i. Since the adversary can compute hard-core bits with certainty, it can compute $b_1 = B(x,p)$ and $b_2 = B(x,p^i)$. By a simple case analysis, $x[i] = b_1 \oplus b_2$. After n iterations of this procedure (i.e. separately for each bit of x), we have the entire string x.

The somewhat easy proof: Now suppose that for every x, the adversary A_B is able to guess the hard-core bit $B(x,p)$ given $f_2(x,p)$ with probability (over p) greater then $\frac{3}{4}+\epsilon(n)$: $Pr_p[A_B(f_2(x,p)) = B(x,p)] > \frac{3}{4}+\epsilon(n)$. Using the same procedure as in the super-easy proof, i.e. for each $x[i]$, we pick a random p and compute p^i, then guess $B(x,p)$ and $B(x,p^i)$ to help us determine $x[i]$. If we let E_1 and E_2 denote the events that the adversary's guesses for $B(x,p)$ and $B(x,p^i)$ are correct. We know that:

$$Pr_p[E_1 \stackrel{\text{def}}{=} [A_B(f_2(x,p)) = B(x,p)]] > \frac{3}{4}+\epsilon(n)$$

and

$$Pr_p[E_2 \stackrel{\text{def}}{=} [A_B(f_2(x,p^i)) = B(x,p^i)]] > \frac{3}{4}+\epsilon(n)$$

But this two events are *not* independent. Our guess for $x[i]$ is correct if both E_1 and E_2 occur (we also happen to get lucky if neither E_1 nor E_2 occur, but we ignore this case). We know that $\Pr_p[\neg E_1] = \frac{1}{4}-\epsilon(n)$ and $\Pr_p[\neg E_2] = \frac{1}{4}-\epsilon(n)$. Hence, by using union bound:

$$\Pr_p[E_1 \wedge E_2] = 1 - \Pr_p[\neg E_1 \vee \neg E_2] \geq 1 - \left[\left(\frac{1}{4}-\epsilon(n)\right) + \left(\frac{1}{4}-\epsilon(n)\right)\right] \geq \frac{1}{2}+2\epsilon(n)$$

By employing tricks we have already seen, we can run the procedure for poly-many random p for each $x[i]$ and take the majority answer, which by a Chernoff bound amplifies the probability of success so that all bits of x can be guessed correctly with overwhelming probability.

5.5 One-way function have hard-core bits: The full proof

Now that we have obtained some insight as to how using a predictor of a hard-core bit can help us to invert, we are ready to tackle the full proof. Therefore, we now assume that we are given an algorithm A_B which can compute the hard-core bit with probability $> \frac{1}{2}+\epsilon(n)$ (over x, p and it coin-flips) and show an algorithm A_f(which uses A_B as a black-box) to invert f with noticeable probability.

The main idea of the proof is as follows: from the somewhat easy proof it is clear that we can not use our predictor twice on the same random string p. However, if for a random p we **guess correctly** an answer to $B(x,p) = b_1$, we can get the i'th bit of x (with probability $\frac{1}{2}+\epsilon(n)$) by asking A_B to compute $B(x,p^i) = b_2$ *only once* for this (p,p^i) pair. So if we guess polynomially many $B(x,p_j) = b_j$ correctly for different random p_j's we can do it. But we can only guess (with non-negligible probability) logarithmic number of totally independent bits. However, as we will see, we *can* guess (with non-negligible probability) a polynomial number of pairwise independent bits, and hence can do it. Now we go into the details.

5.5.1 Eliminating x from probabilities

In the somewhat easy proof, we assumed that the predictor A_B had $> \frac{3}{4} + \epsilon(n)$ chances for all x. But our A_B does not have such a guarantee. That is, we are only given that:

$$\Pr_{x,p,\omega}[A_B(f_2(x,p)) = B(x,p)] > \frac{1}{2} + \epsilon(n)$$

We wish to say that for a sufficiently "large" fraction of x, we still have a $> \frac{1}{2} + \epsilon(n)$ guarantee (only over the choice of p and ω) and try to invert only on this fraction of x's. Thus, we begin by formalizing the notion of a *good* x and restrict our attention to adversaries which have reasonable chance of inverting $f_2(x,p)$ only on good x.

Definition 5.17 A string x is said to be **good** if $\Pr_{p,\omega}[A_B(f(x),p) = B(x,p)] > \frac{1}{2} + \frac{\epsilon(n)}{2}$ where probability is taken over p and coin-flips ω of A_B.

Claim 5.18 At least an $\frac{\epsilon(n)}{2}$ fraction of x is good.

Proof: Suppose not. Then,

$$\begin{aligned}
\Pr_{x,p,\omega}[A_B(f_2(x,p)) = B(x,p)] &= \Pr_{x,p,\omega}[A_B(f_2(x,p)) = B(x,p) | x \text{ is good}] \cdot \Pr_x[x \text{ is good}] \\
&\quad + \Pr_{x,p,\omega}[A_B(f_2(x,p)) = B(x,p) | x \text{ not good}] \cdot \Pr_x[x \text{ not good}] \\
&\leq 1 \cdot \frac{\epsilon(n)}{2} + \left(\frac{1}{2} + \frac{\epsilon(n)}{2}\right) \cdot 1 \\
&= \frac{1}{2} + \epsilon(n)
\end{aligned}$$

This yields a contradiction, so the claim holds. ■

5.5.2 Overall strategy

Consider an adversary which attempts to invert $f(x)$ only on the set of good x, and succeeds with probability $> \frac{1}{2}$ on this set. Such an adversary succeeds in inverting $f(x)$ with total probability $\geq \frac{\epsilon(n)}{4}$, which is non-negligible, thereby ensuring that f is not a one-way function. This is exactly what we are going to do.

Our next question is for good x, with what probability does the adversary need to guess each bit $x[j]$ of x correctly in order to ensure that the entire x string is guessed correctly with probability $> \frac{1}{2}$. If the adversary computes each $x[j]$ correctly with probability $1-\gamma$, then we can upper bound the probability that the adversary's guess for x is incorrect by employing the Union Bound:

$$\Pr_{\omega}[A_{f_1}(f_1(x)) \text{ gets some bit of } x \text{ is wrong}] \leq \sum_{i=1}^{n} \Pr_{\omega}[A_{f_1}(f_1(x)) \text{ gets } i\text{th bit wrong}] \leq n\gamma$$

where ω are coin-flips of A_f.

Setting $\gamma < \frac{1}{2n}$ guarantees that the probability that some bit of x is wrong is less than $\frac{1}{2}$, or equivalently, ensures that A_{f_1} guess is correct with probability $> \frac{1}{2}$. That is, if A_f can get each individual bit of x with probability grater then $(1 - \frac{1}{2n})$ then we can use the same procedure to get all bits of x with probability greater then $\frac{1}{2}$ even *if our method of getting different bits of x is not independent*!

5.5.3 Using pairwise independent p's

Our next goal is to devise a strategy for the adversary to guess each bit $x[j]$ with probability at least $1 - \frac{1}{2n}$. Again, we begin by making an assumption which seems difficult to achieve, prove the result given the far-fetched assumption and then show how to derive the assumption.

Lemma 5.19 Suppose we are given a collection of $m \stackrel{\text{def}}{=} \frac{2n}{\epsilon(n)^2}$ pairwise independent $p_1, \ldots, p_m$, where $1 \leq i \leq m$, $|p_i| = n$ and every p_i is uniformly distributed. Moreover, suppose that for every i, we are given a b_i satisfying $x \cdot p_i = b_i$. Then, for good x, we can compute $x[j]$ correctly with probability $\geq 1 - \frac{1}{2n}$ in polynomial time.

Proof: The adversary employs the following polytime algorithm.

1. For each $i \in 1, \ldots m$, construct p_i^j by flipping the jth bit of p_i.
2. Compute $b_i^j = x \cdot p_i^j$.
3. Derive a guess for $x[j]$ as in the "somewhat easy" proof: $g_i = b_i^j \oplus b_i$
4. Take the majority answer of all guesses g_i as the guess for $x[j]$.

We are interested in bounding the probability that the majority of our guesses were wrong, in which case our guess for $x[j]$ is also wrong. Define $Y_i = 1$ if g_i was incorrect and $Y_i = 0$ otherwise, and let $Y_m = \sum_{i=1}^{m} y_i$. Thus, we have $\Pr[Y_i = 1] = \frac{1}{2} - \epsilon(n)$ for all i. Using Chebyshev:

$$\begin{array}{rcl}
\Pr\left[Y_m > \dfrac{m}{2}\right] & = & \Pr\left[Y_m - mp > \dfrac{m}{2} - mp\right] \\
& = & \Pr\left[Y_m - mp > \left(\frac{1}{2} - p\right) m\right] \\
& \leq & \Pr\left[|Y_m - mp| > \left(\frac{1}{2} - p\right) m\right] \\
\text{[Chebyshev]} & \leq & \dfrac{1}{4[(\frac{1}{2} - p)^2 m]} \\
& = & \dfrac{1}{4\epsilon(n)^2 m}
\end{array}$$

Substituting in for $m = \frac{2n}{\epsilon(n)^2}$ ensures that the probability that we misguess $x[j]$ (i.e., that $\Pr[Y_m > \frac{m}{2}]$) is at most $\frac{1}{2n}$, proving the claim. ■

Lemma 5.20 If we are given uniformly distributed completely independent $p_1, \ldots, p_l$ for $l \stackrel{\text{def}}{=} \lceil \log(\frac{2n}{\epsilon(n)^2}) + 1 \rceil$ together with $b_1, \ldots, b_l$ satisfying $B(x, p_i) = b_i$ then we can construct in polynomial time a pairwise independent uniformly distributed $p_1, \ldots, p_m$, $m \stackrel{\text{def}}{=} \frac{2n}{\epsilon(n)^2}$ sample of correct equations of the form $B(x, p_i) = b_i$

Proof: The proof hinges on the following fact, whose proof we omit because it is a simple case analysis:

Fact 5.21 Given correct equations $x \cdot p_1 = b_1$ and $x \cdot p_2 = b_2$, then $x \cdot (p_1 \oplus p_2) = (b_1 \oplus b_2)$.

It is easy to see by induction that this fact extends to the case in which there are arbitrarily many b_i and p_i. Therefore, we can generate a large set of new, valid equations by repeatedly choosing an arbitrary subset of the p_is, XOR them together; XOR the corresponding b_is together to form a new equation of the form, for example, $x \cdot p_{1,3,5,7} = b_{1,3,5,7}$. Since there are $(2^l - 1)$ non-empty subsets of a set of size l, by choosing all possible subsets, the new set is of polynomial size $2^{\log l} = m$ and each new equation is polytime constructible. Furthermore, if we look at the symmetric difference of two different subsets, they are pairwise independent, so the entire set of new equations is pairwise independent. ■

5.5.4 Putting it all together

We now have all the machinery to provide a construction for inverting $f_1(x)$ with noticeable probability given a predictor A_B for predicting $B(x,p)$ with probability $> \frac{1}{2} + \epsilon(n)$. We first present the code and then summarize it below:

Algorithm $A_{f_1}(y = f_1(x))$:

Step 1: Pick a set $P \stackrel{\text{def}}{=} \{p_1, \ldots, p_l\}$ uniformly at random, where $|x| = |p_i| = n$ and $l \stackrel{\text{def}}{=} \lceil \log(\frac{2n}{\epsilon(n)^2}) + 1 \rceil$

Step 2: Compute pairwise independent $\hat{P} \stackrel{\text{def}}{=} \{\hat{p}_1, \ldots, \hat{p}_m\}$ where $m \stackrel{\text{def}}{=} 2^l$ and $\hat{P}$ is computed by taking XOR of all possible non-empty subsets of P.

Step 3: For all possible 2^l bit-strings $b_1, \ldots, b_l$ of length l do:

Step 3.1: [Assume that for every $p_i \in P$, $(1 \le i \le l)$, $B(x, p_i) = b_i$]
From P, and $b_1, \ldots, b_l$ compute for every $\hat{p}_k \in \hat{P}$ bit $\hat{b}_k$, where where $\hat{b}_k$ are computed by taking XOR of the corresponding (to $\hat{p}_k$) b_i's from step 3, and where $\hat{b}_k \stackrel{\text{def}}{=} B(x, \hat{p}_k)$ for all $1 \le k \le m$.

Step 3.2: For j from 1 to n do: [compute all bits $x[j]$ of x]

Step 3.2.1: For every $\hat{p}_k \in \hat{P}$, where $1 \le k \le m$ ask A_B to predict $c_k \stackrel{\text{def}}{=} B(x, \hat{p}_k^j)$. Let $b_{\hat{p}_k} \stackrel{\text{def}}{=} c_k \oplus \hat{b}_k$

Step 3.2.2: define $x[j]$ as the majority of $\hat{b}_{\hat{p}_k}$ from step 3.2.1

Step 3.3: Check if for $z \stackrel{\text{def}}{=} (x[1], \ldots, x[n])$ after step 3.2 $f_1(z) = y$.
If so, **return** z, otherwise go back to step 3.

The adversary randomly selects a set of l strings of length n to form a set P. It then iterates through all possible completions $x \cdot p_i = b_i$, of which there are only $2^{\log l} = m$. For each incorrect completion, the adversary will perform a polynomial amount of useless work which we are not interested in; we focus on the work performed on the correct completion of the set of equations (which we can check in step 3.3). By Lemma 5.10, since the set of l equations is totally independent, we can construct a pairwise independent set of m equations which are also correct. (step 3.1) Now from Lemma 5.9, this set of m equations suffices to invert $f(x)$ if x is good with probability $> \frac{1}{2}$. Early on, we noticed that the existence of a probabilistic poly-time A_{f_1} which succeeds in inverting $f_1(x)$ for good x with probability $> \frac{1}{2}$ proves that we can invert f with probability greater then $\frac{\epsilon(n)}{4}$, since good x occurs with probability greater then $\frac{\epsilon(n)}{2}$. But if we can invert f_1 with probability greater then $\frac{\epsilon(n}{4}$, f_1 is not a strong one-way function. This completes the proof of the contrapositive, so we have shown that every one-way function has a hard-core bit. ■

Remark: Instead of searching through all possible 2^l bit strings $b_1, \ldots, b_l$ in step 3, we can just pick *at random* $b_1, \ldots, b_l$ and try it only once. We guessed correctly with probability $\frac{1}{2^l} = \frac{1}{poly}$, hence we will still invert f_1 on good x with $\frac{1}{poly}$ probability.

6 Pseudo-randomness

6.1 Introduction

Today's lecture[9] will concern provably secure pseudo-random generators.

Our objectives in defining such a pseudo-random generators are twofold: First, we have to define what we mean by "looking random" to a polynomially-bounded player. Second, we want to be able to construct a Pseudo-Random Generator that is based on a one-way function. In this lecture we will only show a construction based on a one-way permutation.

6.2 Motivation and Definitions

Recall from the first class the one-time pad method for encryption. In this method, a random bit sequence k is chosen for a key, and as long as the sender S and receiver R each has a copy of k, they can communicate: S can send R a message m which is bitwise exclusive-ored with k. As long as the adversary does not know k, all s/he sees is random noise.

The restriction to this system is that k can only be used once, and it must be approximately the same length as m. If k is polynomially shorter than n (the length of m), then some information *is* released. However, we want to make this information useless to an polynomially-bounded adversary. For example, we want to be able to pick a much smaller random seed s, feed it into a function, $G(s)$, which would then produce a much longer string which "looks random", and, hence is "just as good" as a totally random string. If we want randomness, why not just flip coins? Physically, it is expensive to flip coins, so we would rather flip a much smaller number of coins, say 300 or so), then use them as a seed to a pseudo-random generator (PRG) to obtain a polynomially large number of pseudo-random bits.

6.2.1 Looking Random: a computational definition

What does it mean to look random? Suppose we have a pseudo-random generator (a deterministic poly-time computable program) which takes a "seed" of length n and outputs a pseudo-random string of length $2n$. Since there are only 2^n possible inputs to such a program, there are only at most 2^n possible outputs, which are strings of length $2n$. But there are 2^{2n} possible strings of length $2n$. Hence, any such pseudo-random generator outputs only a tiny fraction ($\frac{2^n}{2^{2n}} = \frac{1}{2^n}$) of all possible strings of length $2n$. Never-the-less, for a randomly chosen input, the output should still "look random" to any poly-time machine. In other words, given a truly random distribution U_{2n} of $2n$ bit stings and an output of the generator distribution $G_{n,2n}$, (produced by picking an n bit string at random and then computing a pseudo-random sequence of length $2n$ bits using this seed) $G_{n,2n}$ is defined to be pseudo-random iff there does not exist a polynomial-time machine which can tell whether a sample came from U_{2n} or from $G_{n,2n}$.

More formally, a sequence ("ensemble") of random variables $\{X_n\}_{n\in N}$ is an infinite sequence of random variables where X_n output strings in $\{0,1\}^n$. U_n is a uniform distribution on strings of length n.

A *Sampleable probability distribution* is a distribution which can be produced efficiently in polynomial time. In other words, we say that

[9]Scribe notes taken by: Dean J. Grannes, September 15,20, 1994

Definition 6.22 A sequence X_n is *poly-time sampleable* (or just *sampleable*) if $\exists$ poly-time algorithm S such that:

$$\Pr_r[S(1^n, r) = \alpha] = \mathrm{Prob}(X_n = \alpha)$$

where $|r|$ is polynomial in n, and r is chosen uniformly at random.

By this definition, the uniform distribution is sampleable – if we want length n output string, flip and output n coins. Note that the existence of an algorithm S which produces a distribution D (when given as an input a random string) is enough to show that D is sampleable.

6.2.2 Definition of polynomial indistinguishability

A *statistical test* T is a program which takes a single input and outputs a bit. Given two ensembles of distributions, X_n and Y_n, T is given a sample of length n from one of the two distributions. It then outputs a 0 or a 1 (hopefully, depending upon which distribution it believes the sample came from.) X_n and Y_n *pass test* T if T can only distinguish whether a sample came from X_n or Y_n with negligible probability. That is,

Definition 6.23 X_n and Y_n pass test T if $\forall c$, $\exists N$ such and $\forall n > N$

$$\left[| \Pr_{\{X_n, \text{coins of } T\}} (T(X_n) = 1) - \Pr_{\{Y_n, \text{coins of } T\}} (T(Y_n) = 1) | \right] < \frac{1}{n^c}$$

Distributions X_n and Y_n are *indistinguishable* if they pass *all* poly-time (in n) statistical tests. That is,

Definition 6.24 [Yao]: Two sequences X_n and Y_n are *poly-time indistinguishable* iff $\forall c$ and $\forall A \in \mathrm{PPT}$, $\exists n$ such that $\forall n > N$

$$\left[| \Pr_{\{X_n, \text{coins of } A\}} (A(X_n) = 1) - \Pr_{\{Y_n, \text{coins of } A,\}} (A(Y_n) = 1) | \right] < \frac{1}{n^c}$$

6.2.3 Polynomial indistinguishability definition is robust

Next, we consider the following question: why do we limit a polynomial-time statistical test to receiving only *one* sample from one of the two distributions? Perhaps a test which gets polynomially-many samples can distinguish much much better then a test which must make a decision based on only a single sample?

That is, let is define an *extended statistical test* T' as a program which takes a polynomial number of inputs and outputs a single bit. Given two ensembles of distributions, X_n and Y_n, T' is given a polynomially-mane samples from either X_n or Y_n (i.e. all from the same distribution). It then outputs a 0 or a 1 (hopefully, depending upon which distribution it believes the samples came from.) X_n and Y_n *pass extended statistical test* T' if T' can distinguish whether all the samples came from X_n or Y_n with only negligible probability. That is,

Definition 6.25 X_n and Y_n pass extended statistical test T' if $\forall c_1, c_2$, $\exists N$ such and $\forall n > N$

$$\left[| \Pr_{\{X_n, \text{coins of } T'\}} (T'(X_n^1, \ldots, X_n^{n^{c_2}}) = 1) - \Pr_{\{Y_n, \text{coins of } T'\}} (T'(Y_n^1, \ldots, Y_n^{n^{c_2}}) = 1) | \right] < \frac{1}{n^{c_1}}$$

We claim this this definition is not stronger then 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 6.26 If X_n and Y_n [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_n and Y_n.

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 $0 \leq j \leq k$, 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:

non-uniform case (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.

uniform case : 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. ■

Homework: Calculate the best distinguishing probability for T we can achieve in the uniform case by experimenting, (assuming that distinguishing probability for T' is $\epsilon(n)$.).

6.2.4 Statistically Close Distributions

Another related and important notion is of statistically close distributions.

Definition 6.27 The sequence of random variables $\{X_n\}$ and $\{Y_n\}$ is *statistically close* if $\forall c$, $\exists N$ such that $\forall n > N$:

$$\sum_{\alpha \in \{0,1\}^n} \left[|\Pr_{X_n}(X_n = \alpha) - \Pr_{Y_n}(Y_n = \alpha)| \right] < \frac{1}{n^c}$$

It should be noted that

Fact 6.28 $\{X_n\}$ and $\{Y_n\}$ are poly-time indistinguishable iff for every poly-time test $A : A(X_n)$ and $A(Y_n)$ are statistically close.

The proof of this fact is left as a homework problem.

6.2.5 Pseudo-random Distributions and Pseudo-random Generators

Definition 6.29 $\{X_n\}$ is pseudo-random iff $\{X_n\}_N$ is poly-time indistinguishable from the uniform distribution $\{U_n\}_N$

A pseudo-random generator produces a (polynomially-larger) sequences from a random seed, which is pseudo-random. That is,

Definition 6.30 A deterministic algorithm $G(\cdot, \cdot)$ is pseudo-random generator if:

- 1. $G(x, Q)$ runs in time polynomial in $|x|, Q(|x|)$ where Q is a polynomial.
- 2. $G(x, Q)$ outputs strings of length $Q(|x|)$ for all x.
- 3. For every polynomial $Q()$, the induced distribution $\{G(x, Q)\}$ is indistinguishable from $\{U_{Q(|x|)}\}$.

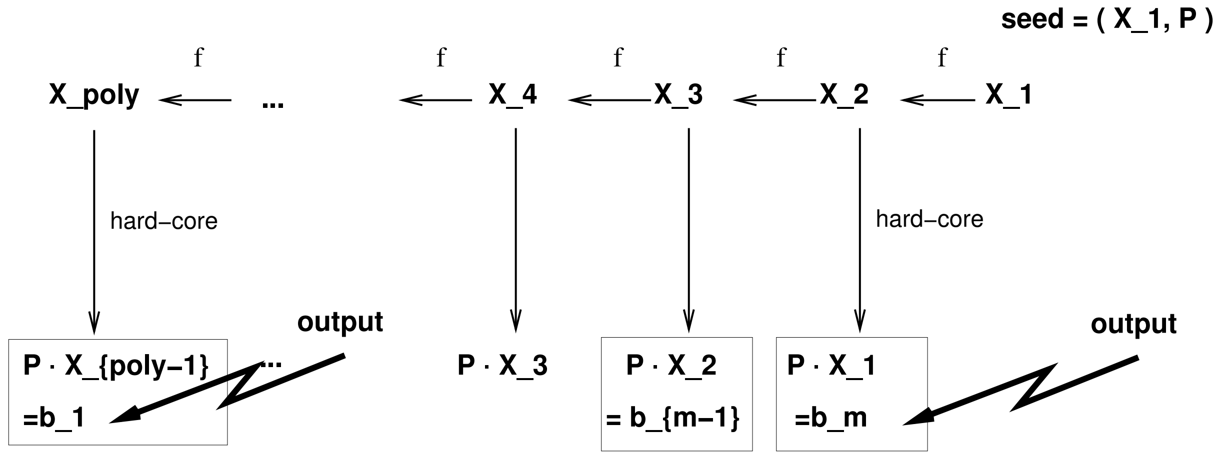
6.3 Construction of Pseudo-Random Generators based on 1-way Permutations

In fact, it is known that the existence of 1-way functions and pseudo-random generators is equivalent. In particular, [Impagliazzo, Levin, Luby, Hastad] have shown how to construct a pseudo-random generator based on any one-way function (the proof is hard, in this lecture we will only show how to construct a pseudo-random generator based on one-way permutations).

It is also the case that if pseudo-random generators exist, then there exist one-way functions. In particular, any pseudo-random generator G *is* a one-way function. To see this, suppose $G : n \to 2n$ is not a one-way function. Then G can be inverted with non-negligible probability, given a pseudo-random string of length $2n$. However, most truly random strings of length $2n$ can not be inverted (there are only 2^n seeds of length n). Thus we can construct an efficient distinguisher between random and pseudo-random strings: given a string, we try to invert it: if we succeed, we say it is pseudo-random, if we fail, we say it is random. If we can invert with non-negligible probability, this gives us a good distinguishing test.

6.3.1 The construction.

Assuming that f is a one-way permutation, we can define a pseudo-random generator G to be the following function ([Blum,Micali],[Yao]), taking as a seed (s) a $2n$-bit string and producing as output a $poly(n) = m$-bit output $b_1, b_2, \ldots, b_m$:



First, s is divided into two halves. One half will be P, the bit string used to generate Goldreich-Levin hard-core bits, and the other half will be X_1, the initial input to f. Bit b_m (the last bit) of the output string is obtained by calculating the hard-core bit of $X_2 = f(X_1)$, which is equal to inner product of P and X_1. Likewise, the second-to-last bit b_{m-1} of the output string is obtained by calculating the hard-core bit of $X_2 = f(X_1)$, which is inner product of X_1 and P and so on. We claim that the output is a pseudo-random string. Clearly, is satisfies conditions one and two of the definition, where the number of times we apply f is equal to $Q(n)$. What remains to show is that this output is pseudo-random.

Remark: in the construction above, the bits are returned in the "reversed" order. However, since we will show that this is pseudo-random, the order (left–to-right or right-to-left) is unimportant: if it is pseudo-random one way, it must be pseudo-random the other way. We have chosen this particular order of the outputs to make the proof easier.

6.4 The Generator output is pseudo-random

The proof is in two steps: First we show that the pseudo-random sequence is *unpredictable* as defined by Blum and Micali. Then we show that every unpredictable sequence is pseudo-random (which was shown by Yao).

Informally, G is *unpredictable* if it is passes the next-bit-test, defined as follows: given $G_1, ..., G_k$, it is hard to predict G_{k+1}, for every prefix k. That is, given the first k bits of the output, it is hard to predict the next bit.

Definition 6.31 X_n passes *the next bit test* (is unpredictable) if $\forall c$ and $\forall A \in PPT$, there $\exists N$ such that $\forall n > N$ and $\forall i, (0 \leq i \leq n)$:

$$\Pr_{X_n,\ \text{coins of A}} [A(\text{ first } i \text{ bits } b_1, b_2, \ldots, b_i \text{ of } x \in X_n\ ) = b_{i+1}] < \frac{1}{2} + \frac{1}{n^c}$$

Claim 6.32 If f is a strong one-way permutation, then G as defined in the previous section is unpredictable.

Proof: The proof is by contradiction. We will show that if G does not pass the next bit test, then we can invert a one-way function on a random input. The construction is as follows: We are given an adversary $A \in PPT$ which for some prefix $b_1...b_i$ of out pseudo-random generator can compute b_{i+1} with probability $> 1/2 + \epsilon(n) = 1/2 + 1/\text{poly}$. We wish to find $f^{-1}(y)$. We know that if we can predict a hard-core bit of $f^{-1}(y)$ and p (for a random p), then we can find the inverse of y (with 1/poly probability). we make y the $n-i$ X_i of the generator. We can then compute (by applying f to y i times, and and computing dot-products, the first i bits of the generator in the strait-forward fashion. Finally, we feed it to our "next-bit" predictor A. Notice that the next bit is exactly the hard-core bit of y. ∎

Remark: Notice that we are dealing with f which is a permutation. Hence, a uniform distribution of the inputs implies a uniform distribution of the outputs, and in the above experiment, the fact that we start with y and compute the first i bits of the output, has the same distribution (uniform) distribution over the seeds.

Next, we show that Unpredictability (next bit test) implies pseudo-randomness:

Claim 6.33 Ensemble X_n is pseudo-random if and only if X_n is unpredictable.

Proof: One direction is trivial: next bit test is a a type of a statistical test. Hence if it passes all polynomial-times statistical tests it passes the next-bit test as well.

In the opposite direction, we must show that passing next bit test implies passing all polynomial time statistical tests. The proof is by contradiction: we assume that there exists some polynomial-time distinguishing test M, and we show how to construct the next-bit test using M as a subroutine.

We assume there exists a test M which distinguishes X_n and U_n with probability $> 1/2+\epsilon'(n)$. We show how to construct the next-bit-test which for some $0 \leq j \leq n$ can predict the "next bit" with probability $> 1/2 + \frac{\epsilon'(n)}{n}$.

We know that M is given a sample from one of the two possible distributions U_n or G_n. Let us call this samples $(U_1...U_n)$ or $G(s) = G_1...G_n$. Moreover, we know that

$$\Pr_{U_n,\text{coins of } M} |(M(U_1...U_n) = 1) - \Pr_{\text{seeds for } G,\text{coins of } M} (M(G_1...G_n) = 1)| > \epsilon'(n)$$

We construct "hybrids"

$$\begin{aligned} P_0 &= U_1\, U_2\, U_3 \dots U_n \\ P_1 &= G_1\, U_2\, U_3 \dots U_n \\ P_2 &= G_1\, G_2\, U_3 \dots U_n \\ &\dots \\ P_n &= G_1\, G_2\, G_3 \dots G_n \end{aligned}$$

W.l.o.g, $P_n - P_0 > \epsilon'(n)$, and therefore, $\exists j$ such that

$$P_{j+1} - P_j > \frac{\epsilon'(n)}{n} \stackrel{\text{def}}{=} \epsilon(n)$$

We will construct the next bit test for G_{j+1} as follows:

Next-Bit-Test $A(G_1, \ldots, G_j)$:

1. pick random bits $U_{j+1}, \ldots, U_n$.
2. Run $M(G_1, \ldots, G_j, U_{j+1}, \ldots, U_j) = b$
3. IF $b = 1$ output U_{j+1}
   ELSE (if $b = 0$) output $(1 - U_{j+1})$

We must measure the probability that the next-bit-test A correctly outputs G_{j+1}.

$$\begin{aligned}
\Pr[A(G_1, ..., G_j) = G_{j+1}] &= \Pr[M(G_1, ..., G_j, (G_{j+1}), U_{j+2}, ...U_n) = 1] \cdot \Pr(G_{j+1} = U_{j+1}) \\
&\quad + \Pr[M(G_1, , ..., G_j, (1 - G_{j+1}), U_{j+2}, ...U_n) = 0] \cdot \Pr(G_{j+1} = 1 - U_{j+1}) \\
&= \frac{1}{2} P_{j+1} + \frac{1}{2} X
\end{aligned}$$

where

$$X \stackrel{\text{def}}{=} \Pr[M(G_1, ..., G_j, (1 - G_{j+1}), U_{j+2}, ..., U_n) = 0]$$

Let us calculate X. We know that

$$\begin{aligned}
P_j &= \Pr[M(G_1, ..., G_j, U_{j+1}, ..., U_n) = 1] \\
&= \Pr[M(G_1, ..., G_j, (G_{j+1}), U_{j+2}, ..., U_n) = 1] \\
&\quad + \Pr[M(G_1, ..., G_j, (1 - G_{j+1}), U_{j+2}, ..., U_n) = 1] \\
&= \frac{1}{2} P_{j+1} + \frac{1}{2}(1 - X)
\end{aligned}$$

Hence, $P_j = \frac{1}{2} P_{j+1} + \frac{1}{2} - \frac{1}{2} X$, thus $\frac{1}{2} X = \frac{1}{2} P_{j+1} + \frac{1}{2} - P_j$. Hence,

$$\begin{aligned}
\Pr[A(G_1, ..., G_j) = G_{j+1}] &= \frac{1}{2} P_{j+1} + \frac{1}{2} X \\
&= \frac{1}{2} P_{j+1} + \frac{1}{2} P_{j+1} + \frac{1}{2} - P_j \\
&= \frac{1}{2} + (P_{j+1} - P_j) \\
&> \frac{1}{2} + \epsilon(n)
\end{aligned}$$

■

NEEDS REVISION

7 Pseudo-random Functions

Topics for today[10]:

1. P vs. BPP

2. Pseudo-random functions (work of Goldreich, Goldwasser, Micali); How to construct them, and proofs; Applications.

7.1 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 7.34 [Yao]: If there exist non-uniform one-way functions then BPP is contained in subexponential time.

That is,

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

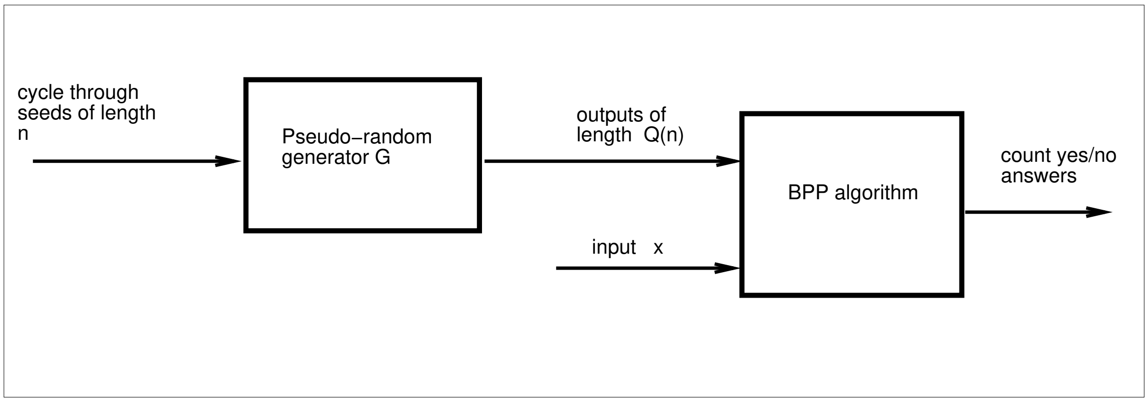
The algorithm, call it P', uses a pseudo-random generator G : n-bits $\rightarrow Q(n)$-bits . The algorithm is:

- 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.

[10]Scribe notes taken by: Michael Galbraith, September 22 and 27, 1994

- Out of those strings, count how many times the BPP machine says yes, how many times it says no.

- If yes'es 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. ∎

7.2 Pseudo-Random Functions, and How to Construct Them

Consider the set of function f : k bits $\rightarrow$ k bits. Since there are 2^k k-bit strings, there are $2^{k \cdot 2^k}$ such functions. By random function from k bits to k bits we mean chosen randomly and uniformly from any 2^{k2^k} all possible functions from k bits to k bits.

We define an oracle Turing machine $A^f(x)$ with access (as a black-box) to a random function f and input x. Instead of functions, we can talk about ensembles of functions, i.e. distributions of functions for each input length.

Definition 7.35 $\{F_n\}$ and $\{G_n\}$ are **poly-time indistinguishable ensembles of functions** if: $\forall c$, $\forall$ probabilistic poly-time oracle machines $A, \exists N\ s.t.\ \forall n > N$:

$$|Prob(A^{F_n}(1^n) = 1) - Prob(A^{G_n}(1^n) = 1)| < \frac{1}{n^c}$$

Here $Prob(A^{F_n}(1^n) = 1)$ means that oracle machine A has access to some function f from F_n family chosen at random and given to A (as an oracle and where A runs in time polynomial in 1^n and outputs 1, where probability is taken over coin-flips of A, and a random function f from n bits to n bits chosen uniformly from F_n family (of all such functions). In $A^{G_n}(1^n)$ A runs with G_n given as a black-box. So this definition means that any BPP machine A, allowed to toss a polynomial number of coins, still has negligible chance of telling whether a given function f_n given in an oracle form is from F_n or G_n.

We define a pseudo-random function by

Definition 7.36 F_n is a **pseudo-random function** if

1. F_n has only 2^n elements $f_n \in F_n$ and every n-bit string x is a label for some f_n and every f_n appears with probability $\frac{1}{2^n}$.

2. Given x (a label) and input y, $f(x, y)$ is poly-time computable.

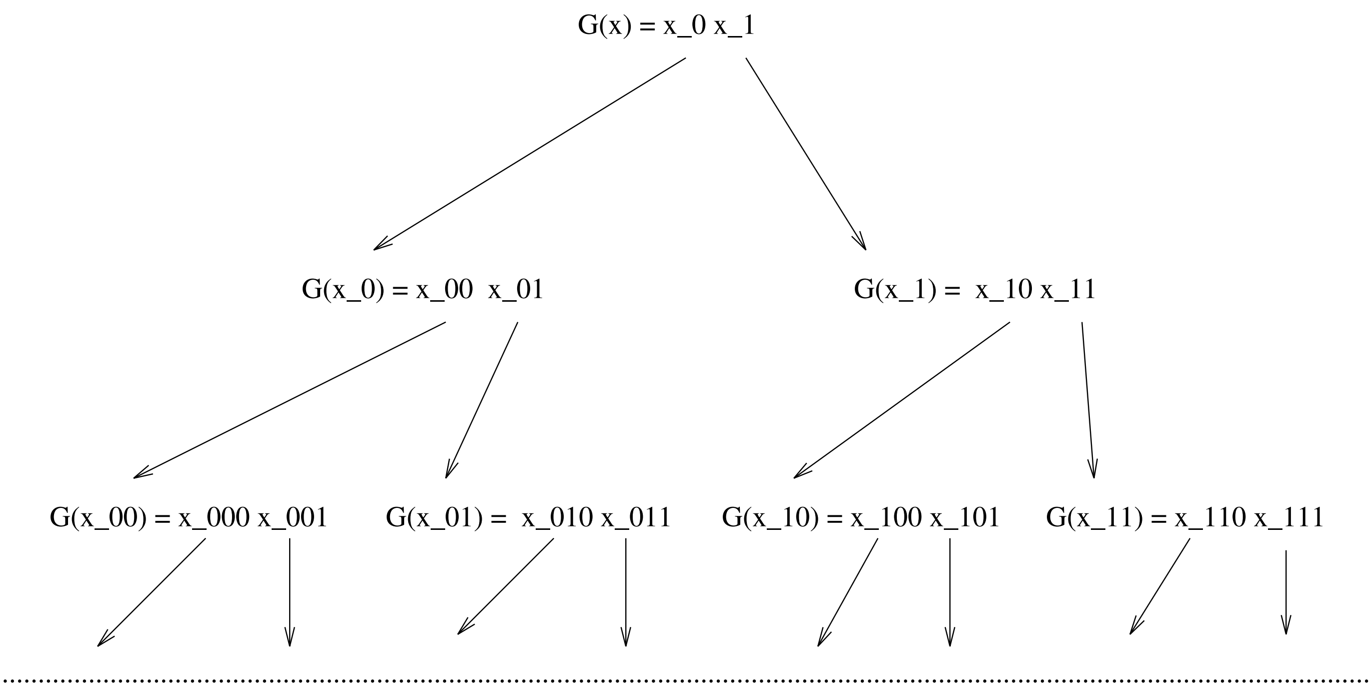
3. $\{F_n\}$ is poly-time indistinguishable from a random ensemble of functions $\{U_n\}$.

Here is the construction:

Assume we have a pseudo-random generator,

$$G : n \text{ bits} \rightarrow 2n \text{ bits}$$

x, with $|x| = n$, is our seed. Compute $G(x) = x_0x_1$, where $|x_0| = |x_1| = n$. That is, x_0 is the first n digits of $G(x)$ and x_1 is the last n digits. Then compute $G(x_0) = x_{00}x_{01}$ and $G(x_1) = x_{10}x_{11}$. Proceed in this manner and define a tree,



Define this tree to n stages (so that at the last stage, the x's are indexed by an n-bit string). Note that even though this is an exponential-size object, we can still define it, as long as we don't have to construct the entire tree, but just explore some small subset of it (which is what we will do).

Finally, our function f takes x as its seed and reads the input y as the path to be taken down this tree, i.e.

$$f_{x=\text{seed}}(\text{input } y = \text{ path down this tree}) = \text{ output leaf}$$

i.e. at each branch, pick path according to next bit in input. For instance, if our input string begins $y = 001...$ then we first go to x_0, then from that we calculate $G(x_0) = x_{00}x_{01}$ and pick x_{00}, from that calculate x_{001}, etc. The final output is whatever leaf we arrive at on the last branch of the tree, after n branchings.

Theorem 7.37 If there are pseudo-random generators then there are pseudo-random functions.

We now prove that our construction gives a pseudo-random function.

Proof: Recall that a pseudo-random generator has output that can't be distinguished from truly random.

We will show that, if a polynomial-time machine T exists which can distinguish the output F_n of this process just described from that of the uniform random ensemble of functions U_n, then there is a polynomial- time distinguisher which can distinguish between outputs of our pseudo-random generator G and randomly chosen strings. This contradiction will establish our theorem.

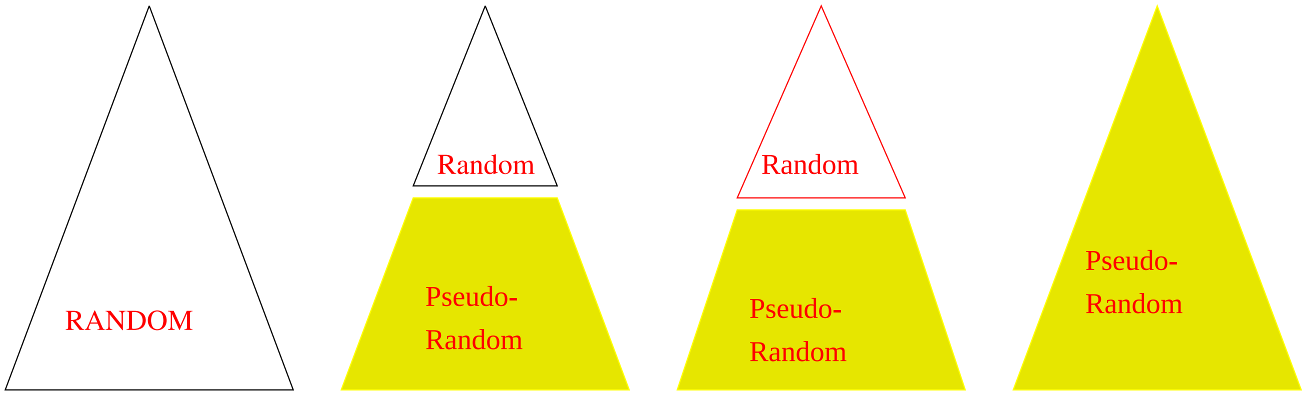
Suppose we have a polynomial-time (in n) machine T, which we give as input either a series of strings $x_1, ..., x_{poly} \in U_n$ from the uniform truly random ensemble or $y_1, ..., y_{poly} \in G_n$ from our pseudo-random ensemble, such that the probability that T outputs 1 for U_n is non-negligibly different from the probability T outputs 1 for G_n. That is, if $M(E)$ is the probability that T outputs 1 when given output from the ensemble E,

$$|M(F) - M(U)| > \epsilon(n)$$

for some non-negligible function $\epsilon(n)$.

We will use a hybrid argument. Consider the four levels of randomness of functions, expressed as trees:

- a completely pseudo-random function F
- $S1$ represents the output of a tree constructed as above, but whose entries are truly random to level i (counting from the top) and pseudo-random thereafter.
- $S2$ which is truly random to level $i+1$ and then pseudo-random.
- a completely randomly generated function U, just a tree with all random entries.



Then, since T can distinguish F from U with distinguishing probability $\epsilon(n)$, and there are n levels to the trees F and U, there must be a level I in between 0 and n such that T distinguishes $S1$ from $S2$ with probability $\frac{\epsilon(n)}{n}$ when $i = I$, because we must for at least one such i have

$$|M(S1) - M(S2)| > \frac{\epsilon(n)}{n},$$

and since $\epsilon(n)$ is non-negligible, so is $\frac{\epsilon(n)}{n}$. We must first guess at which level i this inequality holds. There is a $\frac{1}{n}$ chance we guess right, and thus our distinguishing probability is still non-negligible.

Now suppose our distinguisher T wishes to see the outputs $f(y)$ for inputs $y_1, y_2, ..., y_k$. Each such y_j represents a path down the tree of our algorithm. We can take the beginning of any query y_j, that is, the first i bits of y_j, call them $y_{j1}, y_{j2}, ..., y_{ji}$ and randomly assign an n-bit string x to the internal node in the tree corresponding to those first i bits. Then we apply our algorithm using the pseudo-random generator G to this string x for the rest of the $n-1$ steps down the tree, using $y_{ji+1}, ..., y_{jn}$ the remaining bits of y_j to choose our path. If another of T's queries $y_{j'}$ starts with the same i bits, then we can remember what random string x we chose for that path before, and again start from that x and go on to find the new $f(y_{j'})$ according to the path dictated by the last $n-i$ bits of $y_{j'}$. Since T is poly-time and can only ask for the values of polynomially many $f(y)$'s, we only need to remember polynomially many random numbers x; and repeatedly applying the pseudo-random generator to these is a poly-time process, so we have in all a poly-time algorithm.

Thus we will consistently answer queries from T according to a tree which is random to level i and pseudo-random from level $i+1$ down.

But T can distinguish these answers from those we would make in $S2$, in which we would choose a random string for the first $i+1$ bits of y_j and proceed pseudo-randomly thereafter. Hence T is distinguishing between random strings and pseudo-random strings at the $i+1$ level. Moreover, since T is a poly-time distinguisher, it can only query polynomially-many y_j's. Thus T distinguishes between polynomially-many pseudo-random strings from G and an equal number of random strings.

But as was shown in the last lecture, assertion 4.5 (Robustness of definition of polynomially indistinguishable distributions of strings), when there is a poly-time distinguisher T' between polynomially-many strings from two distributions X_n and Y_n, then there is also a poly-time distinguisher T between single strings from X_n and from Y_n. Thus our T that can tell between poly-many outputs from the hybrid trees $S1$ and $S2$ implies the existence of one that can tell random strings from the output of G. This contradicts the fact that G is a pseudo-random generator and establishes the theorem.

∎

7.3 Applications

Now that we have pseudo-random functions, let's show how incredibly useful they can be.

Suppose in these applications that

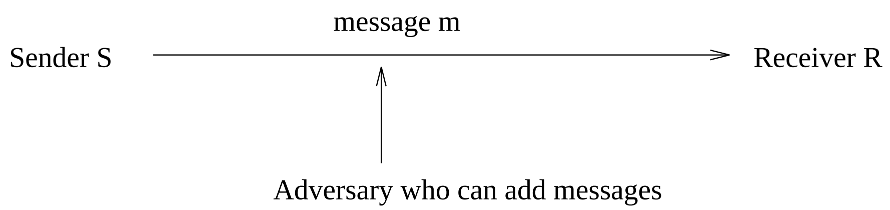
- We can pick our pseudo-random function f : k-bits $\rightarrow$ k-bits (by picking a seed at random)..
- We a given a black box for calculating f.

1. Adaptive security. Suppose an Adversary has access to our black box function and tries to gain information about the function by an adaptive attack: he tries several values of x_i and observes the outputs $f(x_i)$; each time he gets to choose x_i according to what he has learned from his previous $i-1$ attempts. Since our function is pseudo-random, as long as our Adversary is limited to trying polynomially-many (in the input length n) x_i, he won't be able to guess the value of $f(x)$ for any untried value x.

2. Dynamic hashing. A hashing function is an $h : n$ bits $\rightarrow l$ bits for some $l < n$. we can use our pseudo-random f, and let $h(x)$ be the first l bits of $f(x)$

   Then even if our adversary asks us hash several strings of length n, $x_1 \rightarrow h(x_1)$,..., $x_i \rightarrow h(x_i)$, he still cannot guess $h(x_{i+1})$ for a string x_{i+1} that he hasn't yet seen, by the pseudo-randomness of our function f.

3. Message authentication



Sender, receiver, but not adversary have access to f. So each message, sender transmits $(m, f(m))$. Then if Adversary wants to insert phony message m', he can't authenticate it by providing $(m', f(m'))$.

4. Time-stamping.

   Situation same as above. Notice our Adversary could still insert a message m and authentication if it's a message that's been sent before (so that he has seen $(m, f(m))$).

   To prevent this let $y = (m, \text{time}, \text{date})$. Sender sends $(y, f(y))$. Then Adversary can't insert phony message because he hasn't seen $f(y)$ where the y includes the current date and time.

5. Friend or foe system. All friends keep seed s of a pseudo-random function. When meet each other pick a random input x and ask for $f_s(x)$. Even if enemies overhear conversation, if they do not know s they can not compute $f_s(x')$ for random x', hence can not pretend to be included to the club.

NEEDS REVISION

8 Public Key Cryptography

In[11] *public key cryptography* there is a public key p and a private key s used with encryption and decryption algorithms. To send a message m to a person, $E(p,m)$ is computed and sent, where E is the encryption algorithm (which is known to everyone), p is the public key, and m is the message to be sent. The decryption of a message is computed by $D(p,s,E(p,m))$. The algorithm D is known to everyone. Only the secret key s is kept secret.

For public key encryption three things are needed. First is the generator of the public and private key. This is computed by $G(1^k) \to (p,s)$. The next two are the encryption ($E(p,m) \to E(m)$) and decryption ($D(p,s,E(m)) \to m$) algorithms. All three algorithms are probabilistic polynomial time.

The *semantic security* was defined by [Goldwasser, Micali]. Basically it states that whatever you can compute given any partial information about a message m and its encryption, you can also compute given only the partial information (i.e. encryption of a message is useless). The formulation is as follows:

(1) Pick $\mathcal{D}$, a distribution of messages

(2) pick the message m from $\mathcal{D}$,

(3) pick some auxillary function $h()$ of the message (for example, output the first 10 bits of the message).

The adversary A tries to compute $f(x)$. Then, $\forall A, \exists A'$ s.t. $A(x, h(m), E(m)) = f(x)$ is indistinguishable from $A'(x, h(m)) = f(x)$.

Formally, the triple (G,E,D) is semantically secure iff for all sequences X_n, $\forall h \in$ PPT, $\forall f$(even difficult to compute f), $\forall A \in$ PPT, $\exists A'$ s.t. $\forall c \exists N \forall n > N$, $Prob_{G(1^k), coins-of-A}(A(E(X_n), h(X_n), 1^n) = f(X_n)) \leq Prob((A'(h(X_n), 1^n)) = f(X_n)) + 1/n^c$.

To construct E and D, first pick a trap door 1-way permutation (f, f^{-1}), and a hard core bit $H()$ for the function. Then f and H are made public and f^{-1} is kept secret. To send a message m, pick an x at random and a bit b of m to send, and send $< b \oplus H(x), f(x) >$. Then to decrypt, find x with f^{-1}, compute $H(x)$, and then retrieve b. (In practice, quadratic residuocity is used: the integer n and a y which is a non-quadratic residue is made public and and the factorization $pq = n$ is kept secret. Then to send a 0, x^2 (QR) is sent for some random x, and for a 1, yx^2 (QNR) is sent.)

Theorem 8.38 *The protocol above of sending $< b \oplus H(x), f(x) >$ is semantically secure.*

Proof: The proof will consist of two stages. First, proving that the protocol is indistinguishably secure, and then showing that indistinguishable security implies semantic security. ■

Definition 8.39 *A protocol is indistinguishably secure if $\forall$ PPT $A \forall c \exists N \forall n > N$ s.t. after:*

[11]Scribe notes taken by: Sean Hallgren

(1) $A \rightarrow m_0$ *and* m_1

(2) run $G \rightarrow (p, s)$

(3) give A *either* $E(m_0)$ *or* $E(m_1)$

Then $|Prob(A(m_1, m_0, E(m_0)) = 1) - Prob(A(m_1, m_2, E(m_1) = 1))| < 1/n^c$.

Proof of stage 1 (i.e. that it is indistingibly secure) is done by a hybrid argument, where if there is an algorithm which can distinguish $E(m_0)$ from $E(m_1)$ then there is an algorithm which can distinguish:

- a prefix up to i bits of m_0 followed by a suffix of m_1 from
- the prefix of $i+1$ bits of m_0, followed by the remaining bits of m_1, where the $i+1$ bit is different in m_0 and m_1.

But this implies that we can predict a hard-core bit, a contradiction.

For stage 2, (i.e. that indistingisbility implies semantic security). We first note that $A(E(m), h(x))$ is indistinguishable from $A(E(0...0), h(x))$. Then, we construct A' using $A(E(0...0), h(x))$.

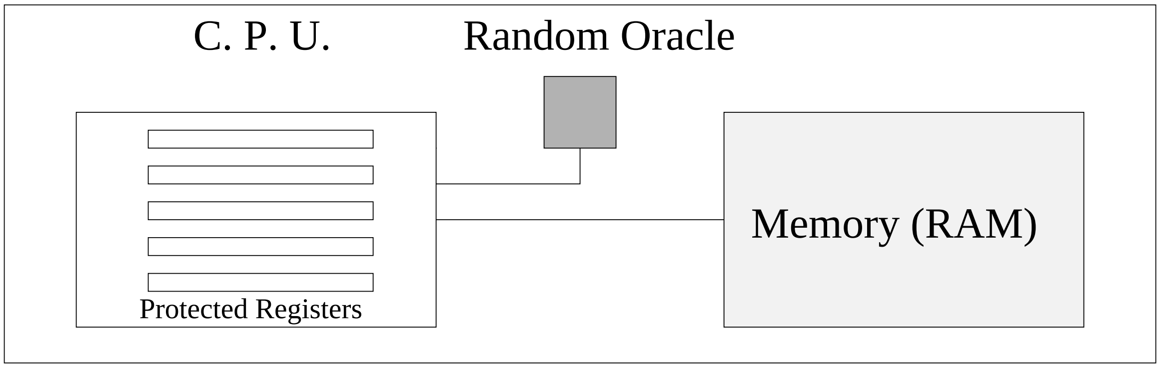
NEEDS REVISION

9 Software Protection

In this lecture[12] we consider the problem of software protection. Informally, we show a method of simulating an arbitrary program by a program which runs on a CPU having access to a random oracle and a few protected registers such that no information other than the input-output behavior of the program can be deduced.

The cryptographic tool that we are using is pseudo-random functions of [Goldreich, Goldwasser, Micali] which were discussed in the previous lecture. Pseudo-random functions are used in this work for both (private-key) encryption and to simulate a random oracle (see previous lecture for definitions).

9.1 The probabilistic RAM Model



Probabilistic RAM Model

The probabilistic RAM consists of a CPU which has has some protected registers and has access to a random oracle. The CPU is connected to a random access memory (RAM) whose contents can be read or altered by the adversary (or the software pirate). The adversary can also snoop on the address bus of the CPU and figure out the order in which the RAM is accessed. However there is no way by which one can find the contents of the CPU's register at a given point in time.

9.2 Protection of Program's Data

9.2.1 Protection against non-tempering adversaries

A non-tempering adversary can read the RAM and watch the communication between the RAM and the CPU, but it can't modify or alter the RAM contents. The first step is to encrypt the program and its data so that no one else copy or use it. This can be done by encrypting the contents of the RAM. One can replace the bit b_i stored at the ith location by $b'_i = F(i) \oplus b_i$, where $F(.)$ is the random function computed using the oracle. The program reads b'_i from the RAM and computes $b_i = b'_i \oplus F(i)$ to compute the actual value of the data. Since $F(.)$ is a

[12]Scribe notes taken by: Rahul Garg, September 27, 29, 1994

random function, the contents of the RAM after this transformation are random to anyone who doesn't know $F(.)$. The next step is to simulate a read (or a write) by a (dummy) read followed by a write at the same memory location. This would hide whether the access was a read or a write. (Thus the most frequently used RISC instructions "load" and "store" would become indistinguishable from each other).

9.2.2 Protection against tempering adversaries

A tempering adversary is allowed to change the contents of RAM. Such an adversary might try to change the RAM contents to get additional information about the working of the program. To secure against such adversaries store of a value v at a location i is simulated as store of a pair $(v, F(v))$. While reading back the value, a pair (v, y) is read and $F(v)$ is computed to check if $y = F(v)$. An adversary which wants to replace v by v' should also replace $F(v)$ by $F(v')$, so that the simulator doesn't come to know that the RAM contents have been altered. Since $F(.)$ is a random function known only to the CPU the adversary can't compute $F(v')$ and hence can't change the RAM contents to store a fake value. However it can still copy a $(v, F(v))$ pair from one RAM location to another and clobber the program's memory. The program can be protected from this by storing a value v at location i as $(v, F(i.v))$ where $i.v$ represents concatenation of i and v. The adversary could still alter a location by placing a value which was stored there is the past. This can be avoided by storing a version number with the value. The version number is incremented each time the location is updated. Thus a value could be stored at location i as $(v, \text{version}, F(v.i.\text{version}))$ (we discuss later how appropriate version number is checked).

With the above encryption a program and its data can be hidden securely. Even after this an adversary might be able to get some information about the working of a program by carefully analyzing the order in which the program accesses its memory. The final milestone in achieving the goal of software protection is to hide the access pattern of a program. In other words the programs have to be transformed in such a way that the access patterns of any two program taking the same running time are indistinguishable from each other.

This problem is formalized as efficient simulation of any RAM program on an oblivious RAM. A machine is oblivious if the sequence in which it accesses memory locations is equivalent for any two inputs with same running time. For example an oblivious Turing Machine is one for which the movement of heads on the tapes is identical for each computation. In 1979 Pippenger and Fischer showed how a two tape oblivious Turing Machine can simulate, on-line, a one tape Turing Machine, with a logarithmic slowdown in the running time. An analogue result for the RAM model of computation is presented.

9.3 The restricted problem

Consider a simpler problem for restricted set of programs which access each location of the RAM exactly once. A solution to the problem can be attained by shuffling the memory locations. After the memory is shuffled (randomly permuted), the accesses to the RAM would appear to be like a random permutation and hence would be oblivious. The same idea can be extended to the general case by organizing the memory into hierarchy.

Lets assume that the program accesses each memory location exactly once. If the memory is permuted randomly before the accesses are made then the accesses will appear to be a random permutation. Thus the access pattern of any two programs will be indistinguishable. The problem reduces to the problem of permuting the memory obliviously.

Assume that the original program accesses n words of the RAM. Allocate $4n.O(\log n)$ words of memory and organize it into $4n$ buckets of size $O(\log n)$ each. Now the original memory is hashed into these buckets using the random oracle to compute the hash function. The hash function $h_s(i)$ can be defined as $F(i.s)$ mod$4n$, where s is chosen randomly.

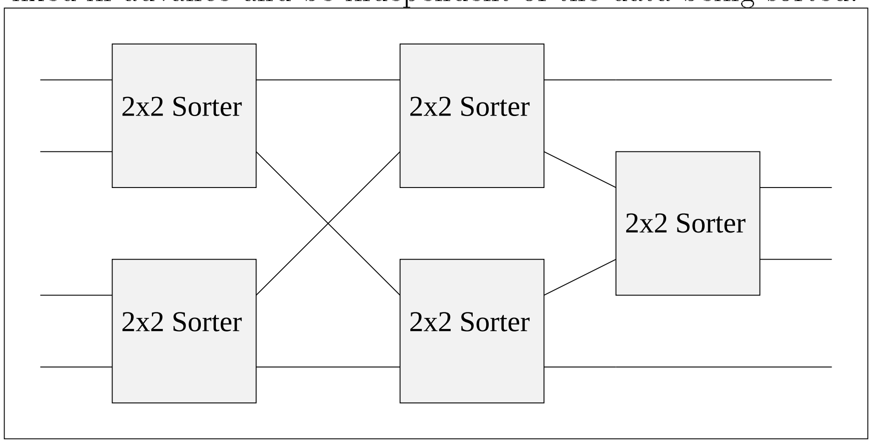
Notice that n items are stored into a hash table with $4n$ entries according to a random oracle. Hence the probability (taken over a choice of randomly chosen function) that the number of entries in a bucket will exceed $O(\log n)$ (bucket overflow) is negligible.

If one can somehow transfer the original RAM contents into the data structure described above, then the access patterns can be hidden. An access to a memory location i can be translated into searching for the appropriate tuple in bucket $h_s(i)$. Any two distinct memory locations are assigned independent buckets and each memory location is accessed exactly once, thus the access pattern (i.e. the bucket access pattern) is independent of the memory access pattern. Whenever a bucket is accessed, each of its element is examined, as a result the memory accesses looks as if one is randomly choosing a bucket and scanning it sequentially.

The final problem is of hashing the elements into buckets obliviously (oblivious hash). This problem can be solved if one can perform sorting of elements obliviously. We first describe the a scheme to perform oblivious sorting and then we show a scheme to do oblivious hash.

9.3.1 Oblivious Sorting

The idea is to implement Batcher's Sorting Network which consists of a number of 2-element sorters as shown in the figure. A 2 element sorter takes two inputs and it outputs them in sorted order. To sort n elements, a network of $(\log n)^2$ such 2-element sorters can be built. Since the network consists of 2-element sorters only, an implementation of this in software will only consist of operations which read two a-priory predetermined memory location and write them back after swapping them (if needed). Thus the order in which the memory locations are accessed by the program will be fixed in advance and be independent of the data being sorted.



4x4 Sorting Network using 2x2 Sorters

9.4 Oblivious Hashing

Now that we know how to perform oblivious sorting, we can make its use to perform oblivious hash. The oblivious sorting is used to to the following.

- Sorting elements within a bucket to group together the empty and full entries. Empty entries are tagged 0 and full entries are tagged 1, the sorting is done on tags.

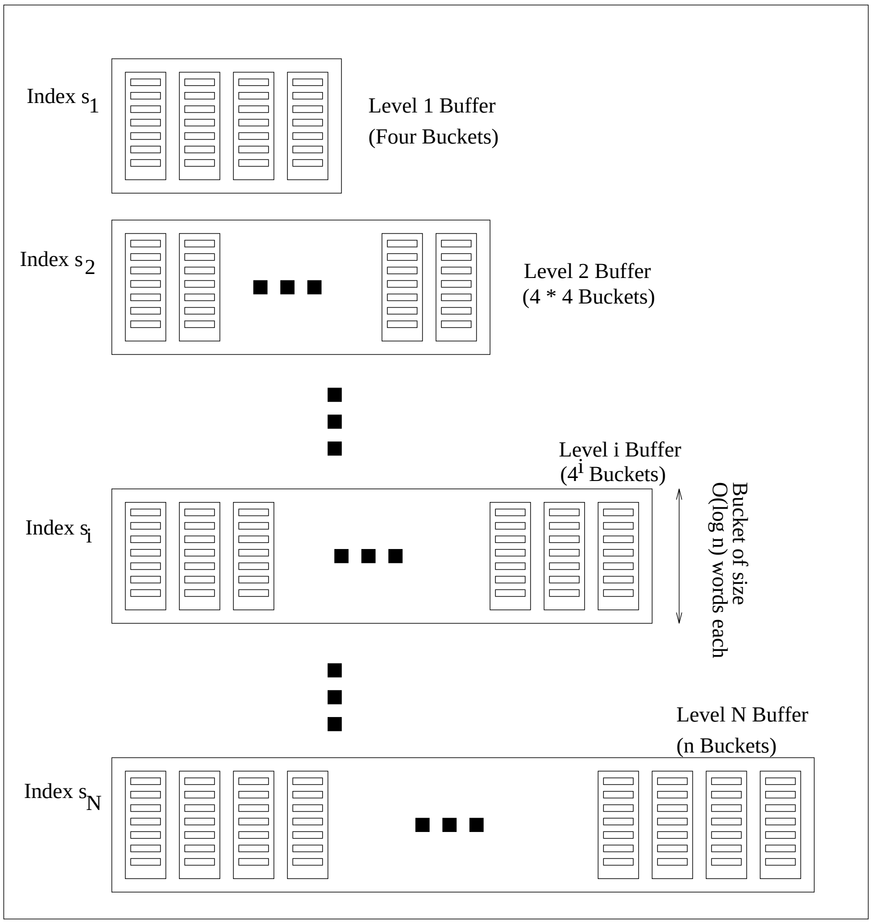
- Sorting elements within two buckets to move all the non empty entries into a single bucket.

- Sorting buckets as individual elements.

First, all the elements of the RAM are tagged by their hash function and then, $4n$ dummy elements corresponding to the $4n$ possible tag values are introduced to make sure that each bucket has at least one element after hashing. The elements are sorted according to their tags and all the elements with the same tag value are put into the same bucket (using oblivious sort in way as described above). The buckets are then sorted according to tags of their elements to make sure that the ith bucket contains all the elements having a tag value of i. This is always ensured because dummy elements make sure that bucket corresponding to each tag value will contain at least one element. Thus an element of the RAM stored at location i is now moved to the bucket numbered $h_s(i)$.

9.5 Simulation on an Oblivious RAM

Instead of a single hash table introduced to solve the simple problem, several hash tables of different sizes are used in the general case. There are N hash tables and each table associated with an index s_i (see figure). These hash tables are called "buffers" which are obliviously hashed at different intervals (smaller buffers are hashed more often than larger ones). The idea is to ensure that for each buffer, no element in that buffer is accessed more than once in between two consecutive oblivious hashes, thus reducing the general problem to the restricted problem already solved.



The Hierarchical organization of RAM

Initially memory is allocated for $N = (1 + \lceil \log_4 n \rceil)$ (n is the length of the input) distinct buffers, where ith level buffer is a hash table of size 4^i buckets and each bucket is of size $\log n$. For each buffer of level i, a random seed s_i is picked and a hash function $h_{s_i}(.)$, where $h_{s_i}(v) = F(s_i.v) \bmod 4^i$, is associated with it. All the buffers are empty in the beginning. The program begins with copying (i.e. hashing) the program and input into the biggest, Nth level buffer. During the simulation if the program refers to a location v, the first level 1 buffer is completely scanned to find it. If it is not found in the first level buffer, then $h_{s_2}(v)$ bucket of the next level buffer is scanned. If it is still not found there then $h_{s_3}(v)$ bucket of the next level buffer is scanned and so on. Note that address of each element is stored along-with the element, so one can compare the address to find whether the an element in the bucket is actually the one which is needed. After the required element is found, the buffers of remaining level are randomly scanned till the Nth buffer. The value is put into level 1 buffer after the access. After some time the buffer at level 1 may start overflowing. To avoid this potential overflow, we move the contents of buffer level i to level $i+1$ after every 4^{i-1} accesses. The contents of buffer at level i and $i+1$ are obliviously hashed into level $i+1$ according to a newly chosen random seed s_i'. Thus, each time when a buffer at level i is at most a quarter full, it is emptied into a higher level. If the higher level does not exist, it is allocated.

Notice that at round 4^i, the total number of entries in the first i buffers is at most 4^i and hence merging their contents into ith buffer (of size 4^{i+1} buckets) can have only a quarter of the entries. When the contents of a lower buffer are moved into the larger buffer, a new hash function is picked and the elements are rehashed according to this new hash function. Thus the collisions which occurred in the smaller buffer do not influence the collisions which may occur in the bigger buffer. That is, the collisions do not accumulate.

9.5.1 Obliviousness of Access Pattern

From the restricted problem, we know that as long as any memory reference is made at most once, the choice of the hash bucket accessed appears completely random to the adversary.

The crucial observation is that for all buffers other than 1, the program never looks for the same memory location more than once, in between hashing stage of that buffer. After a tuple is found at some level, it is moved to the first level. Thus subsequent searches for the same location will find the required tuple in smaller levels unless the tuple migrates to the same level again. Each time something is migrated to a same level, the level is rehashed according to a new randomly selected hash function. The bucket in which a tuple would go will be independent of its previous bucket. Thus, for all buffers $i \geq 2$ and location v, whenever a new s_i is picked, $h_{s_i}(v)$ is computed at most once and hence the access patterns at each level appear to be random to the adversary.

9.5.2 Overhead of Oblivious Simulation

Let t be the running time of the program. At the end of simulation, there would be at most $N = O(\log t)$ buffers. For each memory access in the original program, the transformed program scans all the buckets of level 1 buffer and then scans a single bucket of size $O(\log t)$ on every other level, resulting into $O((\log t)^2)$ accesses for each original access. In addition, during simulation level i and level $i-1$ buffers are hashed into level i buffer a total of 4^{n-i+1} times, $i = 2, \ldots n$. Since the joint size of level $i-1$ and level i buffers is $O(4^i)$ buckets of size $O(\log t)$, it takes $O(4^i . \log t . \log(4^i . \log t))$ steps to obliviously hash them. Thus the total number of steps needed to perform all the hashing of all the buffers is equal to :

$$\sum_{i=2}^{N} 4^{N-i+1} . 4^i . \log t . \log(4^i . \log t) = O(t . (\log t)^3)$$

Therefore the overhead in the oblivious simulation of an arbitrary RAM program is a factor of $O((\log t)^3)$ (with a negligible probability of failure of the simulation).

NEEDS REVISION

10 Introduction to Interactive Proofs

10.1 Introduction

A traditional, Euclidean-style proof for an assertion consists of a prover who simply outputs a proof[13]. Someone reading the proof, a verifier, then decides whether or not the proof is correct. The observation has been made that there could be an advantage to letting the verifier *interact* with the prover. This may allow the assertion to be proven faster or with the release of less information than would be the case if the verifier were passive.

Our general framework consists of a prover P who is allowed an arbitrary exponential amount of time, and a verifier V who is allowed only poly-time. Both P and V are allowed to flip coins and they communicate to each other by sending messages. Note that since V is a poly-time machine, only a poly-number of messages will be sent between P and V.

The programs for P and V define the protocol PV. The input, typically an assertion of the form $x \in L$, is presented to both P and V, and P tries to convince V that the assertion is true. If V is convinced, then V accepts.

10.2 Definition of IP [Goldwasser,Micali,Rackoff]

Definition 10.40 An **Interactive Proof** for a language L is a protocol PV for a Prover and a Verifier such that:

- **Completeness:** If $x \in L$ then P has a good chance of convincing V that $x \in L$

$$\forall c > 0 \ \exists N \text{ s.t. } \forall x \in L \text{ where } |x| > N$$

$$\Pr_{coins\ \ of\ \ V,P}[PV(x) \text{ makes } V \text{ accept }] > 1 - \frac{1}{|x|^c}$$

- **Soundness:** If $x \notin L$ then every P' has little chance of convincing V that $x \in L$

$$\forall P' \ \forall c > 0 \ \exists N \text{ s.t. } \forall x \notin L \text{ where } |x| > N$$

$$\Pr_{coinsofV'}[P'V(x) \text{ makes } V \text{ accept }] < \frac{1}{|x|^c}$$

IP is defined to be the class of languages which have Interactive Proofs.

[13] Scribe notes taken by: Vassilis Papavassiliou, October 4,11 1994

10.2.1 IP for Graph Nonisomorphism

Disclaimer: In these notes we will write interactive proofs for two languages: graph-isomorphism (GI) and graph-non-isomorphism (GNI). These languages are chosen because, besides the belief that V (a PPT machine) cannot recognize them, they provide a convenient framework in which to study the notions of interactive proofs and zero-knowledge. In addition, protocols for GI and GNI can be translated into protocols for certain hard number-theory problems. One would not really base a real system on GI or GNI.

Many of these interactive proofs will rely on the ability to produce random permutations of graphs. We do this by creating a random permutation and applying it to the graph.

$x \in GNI$ iff x is a pair of graphs (G_0, G_1) and $G_0 \not\sim G_1$

We abbreviate this as $x = \{G_0 \not\sim G_1\}$

The following protocol is an interactive proof for GNI. V picks either G_0 or G_1 at random and generates a random permutation of that graph. V sends this new graph to P who responds by telling V which graph V originally picked. Repeat this $k = |x|$ times. V accepts if P is right every time. In tabular form:

		$x = \{G_0 \not\sim G_1\}$	
	P	communication	V
1			Generate a random bit b
2		$\leftarrow G' \leftarrow$	Generate a random permutation Π. Let $G' = \Pi(G_b)$
3	Determine b' s.t. $G' \sim G_{b'}$	$\rightarrow b' \rightarrow$	Reject if $b' \neq b$
4			Repeat steps 1-3 k times. Accept if $b' = b$ every time.

This protocol is an interactive proof because:

- **Completeness:** If the graphs are not isomorphic then only one of G_0 or G_1 will be isomorphic to G', so P will always be able to determine b.
- **Soundness:** If the graphs are isomorphic then G' could have come from either G_0 or G_1 with equal probability, so any prover P' can only guess b. P' would have to guess correctly $|x|$ times. This has probability $\frac{1}{2^{|x|}}$.

10.2.2 Protocol (P1): Interactive Proof for Graph Isomorphism

$x \in GI$ iff x is a pair of graphs (G_0, G_1) and $G_0 \sim G_1$
We abbreviate this as $x = \{G_0 \sim G_1\}$

The following protocol is an interactive proof for GI. P generates a random permutation of G_0. P sends this new graph to V who responds with a bit b. P then responds to V's request by sending the permutation which maps the new graph that P generated to G_b. V checks that this permutation is actually an isomorphism between the graphs. Repeat this k times one after the other (i.e. sequentially). V accepts if P was able to send a correct permutation every time. In tabular form:

		$x = \{G_0 \sim G_1\}$	
	P	communication	V
1	Generate a random permutation Π_1. Let $G' = \Pi_1(G_0)$	$\rightarrow G' \rightarrow$	
2		$\leftarrow b \leftarrow$	Generate a random bit b.
3	Determine Π_2 s.t. $G' = \Pi_2(G_b)$	$\rightarrow \Pi_2 \rightarrow$	Reject if $G' \neq \Pi_2(G_b)$
4			Repeat steps 1-3 k times sequentially. Accept if $G' = \Pi_2(G_b)$ every time.

This protocol is an interactive proof because:

- **Completeness:** If the graphs are isomorphic then G' is isomorphic to both G_0 and G_1, so P can always send an isomorphism between G' and G_b.
- **Soundness:** If the graphs are not isomorphic then G' is isomorphic to at most one of G_0 and G_1. Any prover P' would be able to send an isomorphism between G' and G_b only if G' was originally created as a permutation of G_b. Thus P' would have to guess which b V will send. The probability that P' can do this k times is $\frac{1}{2^k}$.

$\boxed{\text{NEEDS REVISION}}$

11 Introduction to Zero Knowledge

The obvious interactive proof protocol for GI is for P to simply send V the isomorphism between G_0 and G_1. This corresponds to the traditional, non-interactive way of proving things[14]. However this protocol has the undesirable feature of revealing to V much information. In particular, V now knows an isomorphism between G_0 and G_1. We desire an interactive proof which still convinces V that the graphs are isomorphic without revealing so much information to V. In fact, we don't want P to reveal *anything* to V beyond that the graphs are isomorphic. Such proofs are called zero-knowledge, introduced by Goldwasser, Micali and Rackoff (GMR-85). (P1) from the previous lecture is one such protocol for GI.

11.1 Motivating story

(This is a story also due to [GMR]) One night in a small town there is a murder. Verry Fire, the local reporter, hears about the murder and wants to write a story for her newspaper. She goes to a pay phone and calls the detective to get the facts of the murder case, but the detective simply tells her "There was a murder" and he hangs up. She calls back several times, but every time she just hears the phrase "There was a murder." Verry already knew that there was a murder, so she certainly didn't need to call the detective to obtain this information. She could have just saved her money and generated this phrase herself. Feeling a little frustrated, she decides to call the police chief. The chief, who enjoys playing games with reporters, flips a coin, and if the coin is heads, the chief says "There was a murder" and hangs up. If the coin is tails, he says "No comment" and hangs up. Verry calls back several times and sometimes she hears the first phrase while other times she hears the second phrase. Her conversation with the chief, however, still hasn't given her any new information for her column. She could have just flipped a coin herself and generated the chief's phrases with the same probability distribution. Verry proceeds to write her column, but she could have written the column without talking to the detective or the chief, because her conversations with them were *zero-knowledge.*

11.1.1 Definition of ZK

Definition 11.41 For a protocol PV, let $PV(x)$ represent *view* of the conversation from verifiers point of view on input x. Specifically, $PV(x)$ consists of:

- The messages sent back and forth between P and V
- The coins of V

In a later section we will justify including the coins of V in this definition. Let $[PV(x)]$ represent the distribution of points $PV(x)$ taken over the coins of P and V. In general for a

[14]Scribe notes taken by: Vassilis Papavassiliou, October 11 1994

machine S, let $[S(x)]$ be the distribution of outputs of S on input x taken over the coins of S.

Definition 11.42 An Interactive Proof PV for a language L is **Zero Knowledge** if

$$\forall V' \quad \exists S_{V'} \in PPT \quad \text{s.t. } \forall x \in L$$

$$[S_{V'}(x)] \simeq [PV'(x)]$$

$S_{V'}$ is a PPT machine which knows V' and which on input $x \in L$ outputs points of the form $PV(x)$ defined above. Intuitively, the existence of $S_{V'}$ shows that V' does not *need* P to generate the output distribution $[PV'(x)]$. V', a PPT machine, could have generated the distribution itself. Therefore P does not transfer any knowledge to V' (beyond the fact that $x \in L$) which V' could not have generated itself.

The quantifier is over all verifiers V', even cheating verifiers. That is, PV' doesn't *have* to be an interactive proof for L. The only goal of V' may be to extract information from P. Even for such cheating verifiers, there must exist a simulator with $\simeq$ output distribution.

The existence of a PPT simulator for a verifier V' means that the interaction of P and V' is zero-knowledge. If the interaction of P with every verifier V' is zero-knowledge, then the protocol PV is zero-knowledge. In this case, note that the protocol PV' for any verifier V' is also zero-knowledge, but remember that this protocol is not necessarily an interactive proof that $x \in L$.

In the definition of ZK, we use a simulator whose distribution on $x \in L$ is $\simeq$ to the distribution of $PV(x)$. There are actually three different definitions of ZK corresponding to the three different definitions of $\simeq$ of distributions:

Definition 11.43 The three variants of ZK are:

- **Perfect ZK** $=$

  The distributions are exactly equal. This is the strictest definition of ZK.

- **Statistical ZK** $\stackrel{s}{=}$

  The distributions are statistically close. Recall that:

  Distributions $\{X_n\}$ and $\{Y_n\}$ are **statistically close iff**

  $$\forall c \ \ \exists N \ \ s.t. \ \ \forall n > N$$

  $$\sum_{\alpha \in \{0,1\}^n} | \Pr_{\{X_n\}}[X_n = \alpha] - \Pr_{\{Y_n\}}[Y_n = \alpha]| < \frac{1}{n^c}$$

  In other words $\forall c \ \ \exists N \ \ s.t. \ \ \forall n > N$ even a machine which is allowed exponential time must take at least n^c samples before it can distinguish $\{X_n\}$ and $\{Y_n\}$. Thus more than a polynomial number of samples are required to distinguish the distributions.

- **Computational ZK** $\stackrel{c}{=}$

  The distributions are computationally indistinguishable to poly-time machines. Recall that:

  Streams $\{X_n\}$ and $\{Y_n\}$ are **poly-time indistinguishable iff**

  $$\forall c \ \ \forall A \in PPT \ \ \exists N \ \ s.t. \ \ \forall n > N$$

  $$| \Pr_{\{\{X_n\}, A's \ coins\}}[A(X_n) = 1] \quad - \quad \Pr_{\{\{Y_n\}, A's \ coins\}}[A(Y_n) = 1] \ | < \frac{1}{n^c}$$

  In other words $\forall c \ \ \exists N \ \ s.t. \ \ \forall n > N$ any polynomial time machine must have running time at least n^c before it can distinguish $\{X_n\}$ and $\{Y_n\}$. Thus no poly-time machine can distinguish the streams.

distributions are equal $\Rightarrow$ statistically close $\Rightarrow$ computationally indistinguishable

The converses may not be true. In particular, if 1-way functions exist, then *computational indistinguishability* $\not\Rightarrow$ *statistical closeness*:
$\{X_{2n}\}$ is the output distribution of a pseudo-random number generator $G : \{0,1\}^n \to \{0,1\}^{2n}$.
$\{Y_{2n}\}$ is the output distribution of a true random number generator for 2n-bit strings.

$\{X_{2n}\}$ contains at most 2^n different strings.

$\{Y_{2n}\}$ contains 2^{2n} different strings.

Therefore these distributions are statistically distinguishable, however since G is a pseudo-random number generator, they are computationally indistinguishable.

11.1.2 Requiring the Simulator to Output the Coins of V

The output of the simulator consists of the messages passed back and forth between P and V' as well as the random bits used by V'. We will now show why it is necessary for the simulator to output the random bits of V'.

Consider the following protocol for graph-isomorphism. This is clearly not a very good way to prove graph-isomorphism, but it illustrates the need for the simulator to output the coins of V'.

First, a description with words: The input is G_0 and G_1. V randomly selects one of the two graphs, say G_0, and sends to P a random permutation of that graph. P randomly selects one of the two graphs, say G_1, and sends back the permutation which maps the graph it received from V to G_1. If P and V happened to choose different graphs (in the above example V chose G_0 and P chose G_1) then V will be able to determine an isomorphism between G_0 and G_1, and so V will accept the graphs as being isomorphic. Repeat everything k times, and if V is never able to determine an isomorphism between G_0 and G_1, then V rejects. In table form, the protocol is:

		$x = \{G_0 \sim G_1\}$	
	P	communication	V
1			Generate a random bit b
2		$\leftarrow G' \leftarrow$	Generate a random permutation Π_1. Let $G' = \Pi_1(G_b)$
3	Generate a random bit c		
4	Determine Π_2 s.t. $G_c = \Pi_2(G')$	$\rightarrow \Pi_2 \rightarrow$	Accept if $G_{\bar{b}} = \Pi_2(\Pi_1(G_b))$. That is, accept if $\Pi_1 \circ \Pi_2$ is an isomorphism between $G_0 \sim G_1$
5			Repeat steps 1-4 k times. Reject if $G_{b'} \neq \Pi_2(\Pi_1(G_b))$ every time.

This protocol is an interactive proof because:

- **Completeness:** If the graphs are isomorphic then V will determine an isomorphism iff $b \neq c$. The probability that all k trials have $b = c$ is $\frac{1}{2^k}$, so the probability that P fails to convince V that the graphs are isomorphic is negligible.
- **Soundness:** If the graphs are not isomorphic then V will never accept since it only accepts if it can determine an isomorphism between G_0 and G_1.

The protocol is not zero-knowledge because V learns an isomorphism between G_0 and G_1. Thus we cannot construct a simulator for V with the appropriate output distribution where the output consists of the messages passed back and forth between P and V as well as the random bits of V. However, if we only required the simulator to output the messages passed back and forth between P and V, and in the definition of zero-knowledge interactions we only required the distributions to be $\simeq$ on points consisting only of the messages passed back and forth, then the interaction of P and V is zero knowledge because, as shown below, we can construct an appropriate simulator for V. Remember that to prove zero-knowledge for a protocol, we would have to show how to simulate any verifier V', not just V. All we will show is that in the particular case of P talking to V, if we don't require the simulator to output the random bits of V, then we will incorrectly conclude that the interaction between P and V is zero-knowledge.

Code for (Pseudo)Simulator S_V

1. FOR i:=1 TO k DO
   (a) pick a random bit b
   (b) pick a random permutation Π. Let $G' = \Pi(G_b)$
   (c) output messages:

$\leftarrow G' \leftarrow$
$\rightarrow \Pi \rightarrow$

   (d) END FOR LOOP
2. END PROGRAM /* The simulator has output k points */

If we consider only the messages passed back and forth between P and V, then the above simulator produces a distribution of points identical to the one generated by the real P talking to V: for $x \in L$, G' is uniformly distributed over all the graphs isomorphic to G_0 (or G_1 since $G_0 \sim G_1$), and Π is a map between G' and either G_0 or G_1 with equal probability for each. This would lead us to believe that the interaction of P and V is zero-knowledge, even though V may learn an isomorphism between G_0 and G_1.

Now consider the messages passed back and forth between P and V *and* the random bits of V. If we have the above simulator also output the bit b from which G' was created (b is suppose to be the random bit of V), then $S_V(x)$ no longer has output distribution identical to $[PV(x)]$ for $x \in L$. In particular, the points in $[PV(x)]$ have the graph G_c to which Π maps G' independent from bit b. The points in $[S_V(x)]$ always have Π mapping G' to G_b. This justifies including the random bits of V in the view of what the verifier sees in the definition of a zero-knowledge interaction.

11.2 Proving a Protocol is Zero Knowledge

Let PV be the protocol (P1) for graph isomorphism which was defined earlier. It was proved previously that PV obeys Completeness and Soundness, so PV is an interactive proof. In this section we will prove that PV is zero-knowledge. We will do this by constructing, for any verifier V', a simulator whose distribution on $x \in GI$ is the same as that for $PV(x)$. The simulator will depend on the notions of saving the state and restarting a Turing Machine.

The verifier is a special kind of Turing Machine. It consists of a finite state control, an input tape, output tape, work tape, random tape, input communication tape and output communication tape. The information on these tapes and the state of the control completely defines the state of the verifier. Thus, we can save the state of the verifier by saving this information. Say we save the state of the verifier at time t. We then put something on the verifier's input communication tape, let the verifier continue running and observe the behavior of the verifier. We can now restore the verifier to the state we saved at time t. If we now put the same thing on the verifier's input communication tape and let the verifier run again, we will observe exactly the same behavior we did before; the Turing Machine has no way of remembering what we had it do before we restored its state.

11.2.1 Story-time

Jay Lentil, host of a popular late-night television show, convinces the great, world-famous magician Flippo to be on his show. To the amazement of all the viewers, Flippo proceeds to flip a normal coin and have it come up heads 100 times in a row. Not to be outdone, Dave Numberman, host of a competing show, tries to find a magician to match this incredible feat. Dave however doesn't find a suitable magician, so he disguises his assistant Paul in a magician's costume. Dave's show is pretaped, so in the studio, he has Paul flip a coin again and again until it has come up heads 100 times. Then the tape is edited to remove all the coin flips that came up tails. When the tape is shown that night, the viewers are amazed to see a man flipping a coin and having it come up heads 100 times in a row!

11.2.2 Construction of a Simulator for (P1)

The previous story showed how it is possible to run an experiment many times and pick out only the successful experiments. If the chance of success is high enough ($\frac{1}{2}$ in the story) then we can get the required number of successful experiments quickly. We will use this for our simulator $S_{V'}$; we will save the state of the verifier V', run an experiment on V', output the results if they are successful, restore the state of V', run another experiment, etc. We continue until we have the required number of successful experiments. For our simulator, a successful experiment corresponds to a point the simulator can output. At the end, the k points that $S_{V'}(x)$ has output have $\simeq$ distribution to $[PV'(x)]$.

Code for Simulator $S_{V'}$

1. pick at random a random tape R for V'
2. FOR i:=1 TO k DO: /* simulate 3-round atomic protocol k times */
   - (a) record state of S'
     /* record configuration of FSM control of V' */
     /* record work tape configuration of V' */
     /* record head positions of work and random tapes readers */
   - (b) set DONE:=FALSE
   - (c) WHILE (not DONE) DO
     - i. pick bit c at random
     - ii. pick permutation Π at random
     - iii. compute $H_i = \Pi_i(G_c)$
     - iv. send H_i to V' and get bit b from V'
     - v. if b=c
       then
       DONE:=TRUE
       output random tape R and messages:

$\rightarrow H_i \rightarrow$
$\leftarrow b \leftarrow$
$\rightarrow \Pi_i \rightarrow$

       else
       reset V' to previously saved state
     - vi. END WHILE LOOP
   - (d) END FOR LOOP
3. END PROGRAM /* The simulator has output k points */

Each experiment is successful half the time. In particular, it is successful when $b = c$. Therefore in $2k$ expected time, $S_{V'}(x)$ outputs k points. If $S_{V'}(x)$ outputs k points, then the distribution of these points is identical to $[PV'(x)]$. This is true because $S_{V'}$ chooses H_i exactly the same way as P does, and V responds with the same b because it has no way of knowing if it is talking to P or $S_{V'}$.

The only complication comes from the fact that there is an exponentially small chance the simulator will fail to terminate and so fail to produce the desired output distribution. Therefore $[S_{V'}]$ is statistically close to $[PV'(x)]$, not exactly equal. To get perfect zero-knowledge, we can run a brute-force, exponential time algorithm for graph isomorphism in parallel to the simulator. If the exhaustive algorithm finishes before the simulator, we use its results instead (i.e. if it gives an isomorphism between G_0 and G_1, use this isomorphism to create the k points). The expected running time of this method is polynomial since the chance that $S_{V'}$ takes a long time is negligible, but there is a small chance it will run in exponential time. Therefore, we have to modify our definition of ZK slightly to allow simulators which are *expected* PPT rather than PPT.

11.2.3 Parallel Version of (P1)

Protocol (P1) runs a 3-round protocol k times, so there are $3k$ total messages sent. Communication is expensive in reality, so we would like a way to minimize the number of rounds required by an interactive proof. This would save us the overhead on each separate message. One thing we could think of for (P1) is to do all k runs in parallel so that we have only 3 (albeit larger) messages sent. The Parallel (P1) protocol is:

		$x = \{G_0 \sim G_1\}$	
	P	communication	V
1	For i=1 to k: Generate a random permutation Π_i^1. Let $G_i' = \Pi_i^1(G_0)$	$\rightarrow G_1', ..., G_k' \rightarrow$	
2		$\leftarrow b_1, ..., b_k \leftarrow$	Generate random bits $b_1, ...b_k$.
3	For i=1 to k: Determine Π_i^2 s.t. $G_i' = \Pi_i^2(G_{b_i})$	$\rightarrow \Pi_1^2, ..., \Pi_k^2 \rightarrow$	Accept iff $\forall i\ G_i' = \Pi_i^2(G_{b_i})$

The above protocol is still an interactive proof for GI because:

- **Completeness:** If the graphs are isomorphic then P will be able to provide an isomorphism between G_i' and G_{b_i} forall i.
- **Soundness:** If the graphs are not isomorphic then V will only accept if some prover P' can guess right all k times. The probability that this happens and P' fools V is $\frac{1}{2^k}$.

Unfortunately, the above protocol is believed not to be zero-knowledge. In particular, the simulator we created for (P1) will not work for Parallel (P1). This is true because a simulator for a parallel verifier produces a successful experiment only if it guess all k bits $b_1, ..., b_k$ correctly *simultaneously*. It can do this with probability only $\frac{1}{2^k}$. In fact, if the above protocol is zero-knowledge, that would imply that $GI \in BPP$, as was shown by [Goldreich, Krawzyk]. However, later on we will see that with appropriate modifications to the protocols we can make a constant-round ZK for GI.

11.2.4 Application: Interactive Passwords

Imagine that you are in Berkeley and that you want to login to a computer in New York. The computer in New York requires a password, so you send your password over the network to New York, however anyone can tap the network cable and learn your password by monitoring the line. This is clearly not acceptable. You need a system for proving to the computer in New York that you really are who you say you are, without sending any information over the network line that can be intercepted:

1. P and V get together and generate "hard" $G_0 \sim G_1$ and Π, the isomorphism between them (warning: in practice, we will not use GI, since we do not know which graphs are hard to find isomorphism for, but rather some algebraic problem, but for now lets assume that we can somehow find a pair of graphs for which is it hard to find an ismoromorphims, to make our example simpler)

2. Whenever P wants to login, P and V run the ZK proof for GI. If P is able to convince V that $G_0 \sim G_1$, then V allows P to login.

Why is this secure? The answer is that ZK protocols can not be repeated by listeners (i.e. it is not transferable!). Why? Because whatever listener heard on the wire during login, he could of generated all by himself, since is there is a simulator for ZK!

NEEDS REVISION

12 Number of rounds for ZK

The topic of today's lecture[15] is the number of rounds of interaction needed for perfect and statistical ZK proofs. We will also consider the special subclass of Interactive proofs (called Arthur-Merlin proofs), in which verifier can not keep secrets from the prover.

12.1 Arthur-Merlin Protocols

Today's topic explores special type of Interactive Proof, called Arthur-Merlin proofs, introduced by Babai and Moran. The name is drawn from the Arthurian legend, where Merlin is the all powerful (exponential time) prover and Arthur is the polynomial time verifier. Arthur is restricted to generating public random coin tosses (i.e. which Merlin can see) as opposed to Interactive Proofs of [GMR], where the verifier can secretly (from the prover) generate the coin tosses. Moreover, the only messages Arthur is allowed to Send to Merlin is results of his coin-tosses. (Recall that in interactive proofs, Verifier can make arbitrary polynomial computations in order to generate questions for the Prover. Notice in the setting where coins are public, there is no need to send anything but the coin-flips, since whatever questions verifier can compute based on his public coin-flips, prover can compute just as well).

Complexity remarks : Goldwasser and Sipser have shown that if there exists an interactive protocol for a language L, then it can be transformed into an AM protocol for a language L. The transformation they present does not (as far as we know) preserve Zero-Knowledge. Also,in the [Shamir,LFKN] proof that $IP = PSAPCE$, it is in fact shown that $AM = PSAPCE$ as far as languages membership is concerned. This does not tell us anything about ZK, though. Fortunately, Impagliazzo and Yung (and [BGGHKMR]) have shown that everything in IP is in computational ZK. As far as perfect and statistical ZK, it was shown by Fortnow, and Aiello and Hastad that only languages in the second level of the polynomial-time hierarchy (in single round AM intersect co-AM) can be proven in statistical ZK.

12.2 Public/Private coins and The number of rounds of interaction

As stated above, the AM protocol is different from IP in that it restricts the verifier to generating public coin tosses, and it restricts verifier's messages to random strings only. Let us revisit the proof for graph non-isomorphism (GNI) to see if this is either or both ZK and AM.

[15] Scribe notes taken by: Wendy Heffner, October 13, 1994. These notes are based on the previous notes by Sivan Toledo of the same lecture given at MIT in 1992.

P	input: G_0, G_1	V
		Privatly generate a coin flip b
	$\leftarrow \Pi(G_b) = G' \leftarrow$	Generate random permutation Π of graph G_b
Calculate to which graph G' is isomorphic, send back that subscript.	$\rightarrow c \rightarrow$	
		If $c = b$ OK.

In that protocol, the verifier generates a secret bit b and sends to the prover a random permutation $\Pi(G_b) = G'$ isomorphic to one of the two input graphs G_0 or G_1. Then the prover sends back a bit c indicating the graph G_c to which G' is isomorphic. The verifier then checks the value of c to see if matches b. If the graphs G_0 and G_1 are isomorphic, the cheating prover will be caught with probability 1/2. If the graphs are not isomorphic, the verifier will be always convinced.

Clearly, this protocol is not AM. The verifier cannot publicly generate the coin flips in this protocol. In addition, this protocol is not ZK. Suppose the verifier V had a third graph G_{new} that (s)he knew to be isomorphic to either G_0 or G_1. The cheating V could substitute the graph G_{new} for G' in the first step and have the prover P show to which graph G_{new} was isomorphic. Using this technique V, therefore, can gain additional knowledge.

In the future lectures, we will see how to design a perfect ZK protocol for GNI. Could we construct a perfect ZK protocol for GNI which is also AM? The answer depends on the number of rounds of interaction between prover and verifier: Goldreich and Krawczyk have shown that *any* (perfect or statistical) ZK protocol with constant number of rounds which is AM for language L implies that $L \in BPP$. However, if we allow *private* coins, we can design a constant number of rounds perfect ZK protocol for GNI. This we will see next time, this time, we will look at graph-isomorphism problem, and show a constant-round (*private* coins perfect ZK protocol for it, due to [Bellare, Micali, Ostrovsky 1990].

12.3 Private coins, Constant rounds protocol for Graph-Isomorphism

Let's look again at the Graph Isomorphism protocol we devised in the last lecture. In that protocol, the prover sends a random graph C isomorphic to the two input graphs G_0 and G_1, obtained by choosing one of them at random and randomly permuting it. Then the verifier sends a random bit b, and the prover has to show the isomorphism between C and G_b.

P	communication	V
Privatly generate a coin flip x and randomly permute graph G_x.	$\rightarrow \Pi(G_x) = C \rightarrow$	
	$\leftarrow b \leftarrow$	Generate a public coin flip b.
Show isomorphism between G_b and C	$\rightarrow G_b \sim C \rightarrow$	

If the graphs are indeed isomorphic, the verifier always will be convinced. If they are not, the cheating prover will be caught with probability $1/2$. We want to amplify the probability of catching a cheating prover. As we have seen in previous lecture, doing so by repeating the above atomic protocol k times sequentially will amplify the probability to $1-2^{-k}$, and will preserve also the ZK property. But, this requires repeating the atomic protocol sequentially, so the number of rounds raises to $3k$.

Can we squeeze all k repetitions of the atomic protocol into fewer steps? Let us consider what happens when we send all k graphs isomorphic to the input graphs at once, then send all k query bits in one round, and then send all the answers to the queries in one round.

P	communication	V
Privatly generate a coin flips $x_1, \ldots, x_k$ and randomly permute graph $G_{x_1}, \ldots, G_{x_k}$.	$\rightarrow \Pi(G_{x_1}) = C_1, \ldots, \Pi(G_{x_k}) = C_k \rightarrow$	
	$\leftarrow b_1, \ldots, b_k \leftarrow$	Generate a public coin flip $b_1, \ldots, b_k$.
Show isomorphism between each pair G_{b_i} and C_i	$\rightarrow (G_{b_1} \sim C_1), \ldots (G_{b_k} \sim C_k) \rightarrow$	

This protocol is still an interactive proof, meaning that if the graphs are isomorphic the verifier will be convinced, and if they are not he will detect this with probability $1 - 2^{-k}$. Additionally, the protocol is an AM proof, however, the ZK property cannot be established any more.

The problem is that when the simulator sends the k graphs $H_1, \ldots, H_k$ to a verifier, the verifier may send in return query bits that depend on those graphs. This means that if the simulator tries to reset the verifier and send k new graphs that were generated from G_0 and G_1 according to the query bits, the verifier might ask different queries. Depending on luck is also a bad strategy here. While the verifier has no way to know from which of the input graph was each H_i generated, the chances that the query bits will match the generation pattern of the H_i's is only 2^{-k}. So it will take the simulator exponential (in k) expected time to generate a valid conversation. Note that even knowing the code of the verifier is not enough. The verifier might choose the query bits according to some hard to invert hash functions of the H_i's.

What should we do? Intuitively, we would like to modify the protocol such that the verifier will have to commit to its query bits before seeing the H_i's. Of course the commitment should not reveal the bits to the verifier in any way, otherwise he might generate the graphs H_i accordingly, and the protocol will no longer be an interactive proof. Having such a bit commitment mechanism, the idea is that a simulator will be able to get the committed bits, then send some $H_1, \ldots, H_k$. Then the verifier de-commits, or reveals its bits. Now the simulator rewinds the verifier to the state it was in just before receiving the H_i's, and sends another sequence of k graphs, which were generated according to the bits (which must be the same, since the verifier committed to them), and generate a conversation.

We face however a major problem. When we used encryption, we used it to hide information from the verifier. This worked because the verifier has only polynomial time to try to decipher the messages. But our prover has infinite computational power, so the verifier cannot hide anything by encryption. The solution is to modify a bit our requirements from the bit commitment protocol. We will devise a mechanism that will ensure that the prover has no way at all to know the query bits, since this is essential for maintaining the IP properties. We will however let the verifier cheat sometimes, that is to ask a different query than the one he committed to. This has no relevance to the IP properties, but it might affect the ZK property. However, we will make sure that if the verifier changes his bits, then he already knows the isomorphism between G_0 and G_1, so he doesn't gain any knowledge from the protocol.

P	communication	V
Generate 2 random graphs A_0, A_1 by permuting G_0 twice.	$\rightarrow A_0, A_1 \rightarrow$	
	$\leftarrow \Pi_1(A_{b_1}) \ldots \Pi_k(A_{b_k}) \leftarrow$	Generate k random bits $b_1 \ldots b_k$ and k random permutation $\Pi_1 \ldots \Pi_k$.
Generate k random graphs $H_1 \ldots H_k$ by permuting G_0	$\rightarrow H_1 \ldots H_k \rightarrow$	
P checks that all (b_i, Π_i) for all $i = 1, \ldots k$ are valid. If this is *not* the case, P stops the conversation.	$\leftarrow (b_1, \Pi_1) \ldots (b_k, \Pi_k) \leftarrow$	(Send the query bits, with "proofs" that these were indeed committed.)
(Send a proof that P could not decipher the committed bits.)	$\rightarrow G_0 \approx A_0 \approx A_1 \rightarrow$	
(Send the actual proofs.)	$\rightarrow H_1 \approx G_{b_1} \ldots H_k \approx G_{b_k} \rightarrow$	

It is easy to see that the protocol is indeed an IP protocol. Since V accepts only if in the step before last P proves that A_0 and A_1 are isomorphic, V can be sure that P could not know what are $b_1, \ldots, b_k$. So in fact in this sense the protocol is similar to the previous (non ZK) one, since the query bits are sent all after P sends $H_1, \ldots, H_k$. To prove that the protocol is indeed a ZK protocol we have to show a simulator for any verifier. The simulator works in phases, where each phase is divided into two sub-phases. If a phase succeeds, the simulator generates a conversation, and is done. Since the probability of success will be shown to be constant, the expected number of phases of the simulator will be constant. In the first sub-phase, the simulator tries to simulate a conversation under the assumption that the verifier he has is honest, hence the simulator is in the *honest* mode. He generates A_0 and A_1 from G_0, sends them to the verifier, then gets the k permuted copies of them (the commitments), and sends the k random graphs, generated arbitrarily from G_0 or G_1. Then the verifier de-commits and reveals the bits. The simulator then rewinds the verifier to the state just before the k random graphs were sent, and now sends k permuted graphs, but that were generated from G_0 or G_1 according to the query bits. If the verifier is an honest one, and actually de-commits the same bits, we are done, since the simulator can run the protocol to completion. If, however, the verifier sends different bits, we are out of luck. Note that the simulator cannot actually declare the verifier a cheater, since this is not something that a prover could find out in a real conversation. In that case the simulator moves to the second subphase, the *cheating* mode. Now the simulation starts by generating A_0 from G_0 and A_1 from G_1. Note that the verifier cannot distinguish between these modes, since he always sees two random isomorphic graphs. The simulation proceeds as before, the verifier is rewinded, and given new H_i's. If he is honest now, and de-commits to the same bits, we are again out of luck, since the simulator will not be able to demonstrate the isomorphism between A_1 and G_0. But if the verifier cheats again, we can complete the simulation. Consider the bit b_i that the verifier changed. In the first try, he demonstrated that some graph is isomorphic to A_0 say. In the second, he demonstrated that the same graph is isomorphic to A_1. By doing so, he gave the simulator the isomorphism between A_0 and A_1, and therefore between G_0 and G_1. So now there is no problem to answer all the queries, and to show that A_1 is isomorphic to G_0.

The crucial point is that because the verifier cannot distinguish between the honest and cheating mode, the probability that he will cheat in the honest mode and be honest in the cheating mode is at most $1/4$. So the expected number of phases is constant.

The protocol which appears in [Bellare,Micali,Ostrovsky 1990], is a perfect ZK protocol. The way we presented the correctness proof however, only shows that the protocol is a statistical ZK protocol, since a verifier might behave in some deterministic way (that is, be honest or cheat) for some A_0 and A_1's, so it is not clear that the distribution will be completely identical. The protocol is a perfect ZK protocol, however, but the proof of this fact is somewhat more complicated. A similar protocol can be devised for Quadratic Residuosity (and any other random self-reducible problem).

$\boxed{\text{NEEDS REVISION}}$

13 Number-theoretic ZK protocols and Compound ZK protocols

13.1 Some Number Theory

In this lecture[16], we consider Perfect ZK proofs for a number theoretic language. Before doing that, we need some facts from number theory. Let's start with some definitions.

Definition 13.44 $Z_N{}^* = \{x | 1 \le x \le N, gcd(N, x) = 1\}$.

Definition 13.45 $x \in QR(Z_N{}^*)$ if

- $\exists w \in Z_N{}^*$ such that $w^2 = x$ mod N, and
- $(\frac{x}{N}) = 1$ where $(\frac{x}{N})$ is the Jacobi symbol.

For the definition of the Jacobi symbol, see p. 30 of Dana Angluin's lecture note. We make the following assumption:
If N is a product of two large primes and $(\frac{x}{N}) = 1$, then it is hard to decide if x is a QR or not.
Assumption :

Fact 13.46 $(\frac{x}{N})$ can be computed in polynomial time in given N even if you do not know the factorization of N.

Fact 13.47 Suppose $N = P_1P_2$ (a product of two different primes). Given $x \in QR(Z_N{}^*)$, there are four different square roots of x mod N, say $y, -y, z, -z$. Then we have $gcd(y + z, N) = P_1$ or P_2. In particular, if you know these four square roots, it is easy to factor N.

Proof: Since $y^2 \equiv x \bmod N$ and $z^2 \equiv x \bmod N$, $y^2 \equiv z^2 \bmod N$. So there is K such that $y^2 - z^2 = KN$. Since $y^2 - z^2 = (y + z)(y - z)$, we have $P_1|y + z$ or $P_1|y - z$. Also, $P_2|y + z$ or $P_2|y - z$. From these, it is easy to see $P_1|y + z$ xor (exclusive or) $P_2|y + z$. Thus, we have $gcd(y + z, N) = P_1$ or P_2. ■

On the other hand,

Fact 13.48 if you know the factorization of N, then it is easy to check whether x is a QR or not.

Fact 13.49 If both q and x are squares, then qx is a square. If q is a square and x is not a square, then qx is not a square.

[16]Scribe notes taken by: Shuzo Takahashi, October 18, 1994

13.2 A Perfect ZK Proof for QR in $3k$ rounds [GMR]

Now we describe an IP protocol for QR. In this protocol, an input is a pair (N, x), and the prover needs to convince the verifier that $x \in QR(Z_N{}^*)$.

Protocol 1

P	communication	V
Generate a square q at random.	$\rightarrow q \rightarrow$	
	$\leftarrow b \leftarrow$	Generate a random bit b.
Send $\sqrt{q}$ (if $b = 0$) and send $\sqrt{qx}$ (if $b = 1$).	$\rightarrow \sqrt{q}$ or $\sqrt{qx} \rightarrow$	

Repeat the above k-times sequentially.

It is easy to see that this is an IP proof. Notice the similarity between the protocol for the graph isomorphism and Protocol 1. Actually, this protocol can be translated into the graph isomorphism protocol. Now we show that Protocol 1 is a statistical ZK proof (it can also be shown that the above protocol is perfect ZK by running, in parallel to the simulator below, an "exponential search" simulator). The following is a statistical ZK simulator:

Description of a Simulator

(1) Set the state of the verifier as usual.

(2) Pick a bit b' and $l \in Z_N{}^*$ at random. Then if $b' = 0$, set $q = l^2 \bmod N$ and if $b' = 1$ then $q = l^2 x^{-1} \bmod N$.

(3) Then send q to the verifier.

(4) If $b = b'$ (where b is a random bit generated by the verifier), then we can supply $\sqrt{q}$ or $\sqrt{qx}$ (depending on $b = 0$ or $b = 1$) to the verifier. Otherwise, reset the state to (1) and repeat (2)-(4).

The distribution created by taking only the successful repetitions is equal to the distribution of the prover-verifier conversation in Protocol 1.

Remark: the above protocol is also perfect ZK, where the above simulator is augmented by adding low-probability exponential-time search, as before.

13.3 A Perfect ZK proof for QR in 5 rounds

In this section, we give a perfect ZK proof for QR which takes only 5 rounds. (it is a simplification of a [BMO-90] protocol for this problem.) Again, the input is (N, x) and the prover needs to convince the verifier that $x \in QR(Z_N{}^*)$.

Protocol 2

P	communication	V
Randomly generate $s \in Z_N{}^*$, and calculate $z = s^2$.	$\rightarrow z \rightarrow$	
	$\leftarrow y_1, ..., y_k$ (commitment) $\leftarrow$	Randomly generate bits $b_1, ..., b_k$ and $r_1, ..., r_k \in Z_N{}^*$, and calculate $y_i = z^{b_i} r_i^2$.
Randomly generate squares $q_1, ..., q_k$.	$\rightarrow q_1, ..., q_k \rightarrow$	
P checks that $y_i = z^{b_i} r_i^2$ for all $i = 1, \ldots k$. If this is *not* the case, P stops the conversation.	$\leftarrow r_1, ..., r_k, b_1, ..., b_k \leftarrow$	De-commit $y_1, ..., y_k$, i.e., show $r_1, ..., r_k$ and $b_1, ..., b_k$ to the prover.
Send a proof that the prover couldn't decipher the committed bits.	$\rightarrow \sqrt{z} \rightarrow$	
Send $\sqrt{q_i}$ or $\sqrt{q_i x}$ for each i depending on $b_i = 0$ or $b_i = 1$.	$\rightarrow \sqrt{q_i}$ or $\sqrt{q_i x} \rightarrow$	

We can show that Protocol 2 is a perfect ZK proof. The proof is similar to the [BMO-90] five-round graph isomorphism simulator..

13.4 Another Example of Perfect ZK Proof: Proving an "OR" of GI

Let $L = \{\langle (G_0, G_1), (C_0, C_1) \rangle |$ either $G_0 \sim G_1$ or $C_0 \sim C_1\}$. In the following protocol, the prover needs to convince the verifier that at least one of two pairs of graphs is isomorphic.

Protocol 3

P	communication	V
Randomly generate bits b_1 and b_2, and graphs G' and C' such that $G' \sim G_{b_1}$ and $C' \sim C_{b_2}$.	$\rightarrow G', C' \rightarrow$	
	$\leftarrow b \leftarrow$	Randomly generate a bit b.
Choose $b_1{}'$ and $b_2{}'$ such that $b = b_1{}' \oplus b_2{}'$, and such that $G' \sim G_{b_1{}'}$ or $C' \sim C_{b_2{}'}$.	$\rightarrow b_1{}', b_2{}', G' \sim G_{b_1{}'}, C' \sim C_{b_2{}'} \rightarrow$	

Repeat the above k-times sequentially

Now we claim:

Claim 13.50 The above protocol is an IP proof for the language L.

Proof: If $x \notin L$, that is, $G_0 \not\sim G_1$ and $C_0 \not\sim C_1$, then the prover cannot find b_1', b_2' described in Protocol 3 at least half the time. So the verifier will reject with probability $\geq \frac{1}{2}$. On the other hand, if $x \in L$, say $G_0 \sim G_1$, then the prover can change b_1 to b_1' (if it is necessary) and take $b_2{}' = b_2$ so that $b = b_1{}' \oplus b_2{}'$ for b which is sent by the verifier. So the prover can always convince the verifier. ∎

Next we claim:

Claim 13.51 The protocol is a statistical ZK proof

Proof: The following is a simulator:

Simulator

(1) Record the state of the verifier.

(2) Pick b_1, b_2 at random and generate G', C' such that $G' \sim G_{b_1}$ and $C' \sim C_{b_2}$.

(3) Send G' and C' to the verifier.

(4) If $b = b_1 \oplus b_2$ (where b is a random bit generated by the verifier), then we can supply b_1, b_2, G_{b_1}, and C_{b_2} to the verifier. Otherwise reset the state of the verifier to (1) and repeat (2) - (4).

The distribution created by taking only the successful repetition is statistically close (i.e. with exponentially small probability the protocol is always guessing wrong, and then we can not proceed) to the distribution of the original prover-verifier conversation in the protocol. Thus, Protocol 3 is a statistical ZK proof. ∎

Remark: the above protocol is also perfect ZK, where the above simulator is augmented by adding low-probability exponential-time search, as before.

NEEDS REVISION

14 Perfect Zero-Knowledge Graph Non-Isomorphism

In this lecture we consduer ZK protocol for GNI, due to [GMW][17].

Let $L = \{(G_0, G_1) | G_0 \not\sim G_1\}$. Now, the prover's task is to convince the verifier that two graphs G_0 and G_1 are not isomorphic. The following is an IP protocol for L.

Protocol 4

P	communication	V
	$\leftarrow G' \leftarrow$	Randomly generate a bit b and G' such that $G' \sim G_b$.
Find b such that $G' \sim G_b$.	$\rightarrow b \rightarrow$	

Repeat the above k-times.

Now we claim:

Claim 14.52 Protocol 4 is an IP proof for the language L.

Proof: If $G_0 \not\sim G_1$, then the prover can always tell b which is sent by the verifier. On the other hand, If $G_0 \sim G_1$, then the prover can send b at most half the time. ■

However, Protocol 4 is **not** a ZK proof (of any type)! Suppose the verifier has a graph C such that $G_0 \sim C$ or $G_1 \sim C$; but the verifier does not know which is the case. If the verifier sends C to the prover and the prover answers faithfully, then the verifier knows which one of G_0 and G_1 is isomorphic to C. This is extra information. Also, notice that an obvious simulation does not work because a verifier V' might have a biased coin. A ZK proof for the graph non-isomorphism exists, to design it, let us first consider a modification of the above protocol:
Here's another interactive protocol for graph non-isomorphism (GNI) – P wants to prove to V that $G_0 \ncong G_1$.

Prover	Communication	Verifier
		Flip a coin $b \in \{0, 1\}$
	$\leftarrow C_0, C_1 \leftarrow$	Generate random permutations $C_0 \cong G_b$ and $C_1 \cong G_{\bar{b}}$
Guess bit b	$\rightarrow b \rightarrow$	

(1) This is an interactive protocol:

- If $G_0 \ncong G_1$, then $C_0 \ncong C_1$, so P can always distinguish between them and thus guess the bit.

[17]Scribe notes taken by: Shuzo Takahashi, October 18, 1994; Sanjoy Dasgupta October 27. 1994

- If $G_0 \cong G_1$, then $C_0 \cong C_1$, so no prover can distinguish them; thus any prover has at most a $1/2$ chance of fooling V. This can be reduced to $1/2^k$ by repeating the process k times.

(2) However, the protocol is not clearly zero-knowledge: we run into difficulties constructing a simulator for it.

The basic problem is that V may not know the bit b before the prover tells him what it is, so that he actually gains information. To overcome this, we force V to start by proving to P that he knows b (so that he can't possibly have gained anything from P's answer). Here is the expanded protocol:

Prover	Communication	Verifier
		Flip a coin $b \in \{0, 1\}$
	$\leftarrow C_0, C_1 \leftarrow$	Generate random permutations $C_0 \cong G_b$ and $C_1 \cong G_{\bar{b}}$
		V proves that he knows b:
	$\leftarrow D_1, \ldots, D_{2k} \leftarrow$	Generate graphs $D_1, \ldots, D_{2k}$, $\{D_{2i-1}, D_{2i}\} \cong \{C_0, C_1\}$
Flip coins $q_1, \ldots, q_k$	$\rightarrow q_1, \ldots, q_k \rightarrow$	Find k pairs of isomorphisms $\pi_1, \ldots, \pi_k$, such that $\pi_i : \{D_{2i-1}, D_{2i}\} \rightarrow \{G_0, G_1\}$ if $q_i = 0$, and $\pi_i : \{D_{2i-1}, D_{2i}\} \rightarrow \{C_0, C_1\}$ if $q_i = 1$
Check the π_i	$\leftarrow \pi_1, \ldots, \pi_k \leftarrow$	
Find bit b	$\rightarrow b \rightarrow$	

Notation:
$\{A, B\} \cong \{C, D\}$ means that $(A \cong C \wedge B \cong D)$ or $(A \cong D \wedge B \cong C)$.
$\pi : \{A, B\} \rightarrow \{C, D\}$ means that $\{A, B\} \cong \{C, D\}$ and π consists of the two relevant mappings.

To get an idea of what is going on in the "proof of knowledge" phase, consider the pair (D_1, D_2). P will ask V to either demonstrate a mapping $\{D_1, D_2\} \rightarrow \{G_0, G_1\}$ or $\{D_1, D_2\} \rightarrow \{C_0, C_1\}$. Since V doesn't know beforehand which mapping he will be asked for, he must know both of them (otherwise, he'll fail with probability $1/2$). If he knows both of them, then he automatically knows a mapping $\{G_0, G_1\} \rightarrow \{C_0, C_1\}$ and he therefore knows b. So the chance that he doesn't know b but manages to give the right answers (for all k pairs) anyway is $1/2^k$. Here's a statistical ZK simulator $S_{\hat{V}}$ for this protocol:

1. Get (C_0, C_1) and $(D_1, D_2), \ldots, (D_{2k-1}, D_{2k})$ from $\hat{V}$.
2. Record $\hat{V}$'s state (the usual).

3. Send random bits $q_1, \ldots, q_k$, and get responses $\pi_1, \ldots, \pi_k$.
4. Reset $\hat{V}$ to its old state (in 2).
5. Send new random bits $q'_1, \ldots, q'_k$, and get responses $\pi'_1, \ldots, \pi'_k$. With high probability $(1 - 1/2^k)$, $q_i \neq q'_i$ for some i. For this i, we have $\pi_i : \{D_{2i-1}, D_{2i}\} \to \{G_0, G_1\}$ and $\pi'_i : \{D_{2i-1}, D_{2i}\} \to \{C_0, C_1\}$. Therefore, we know b.

The distribution created by successful runs of this simulator is statistically equivalent to that created by the prover-verifier conversation. It is not necessarily perfect ZK because of the tiny chance of failure. To make the simulator perfect ZK, we can augment it with a low-probability exponential-time search.

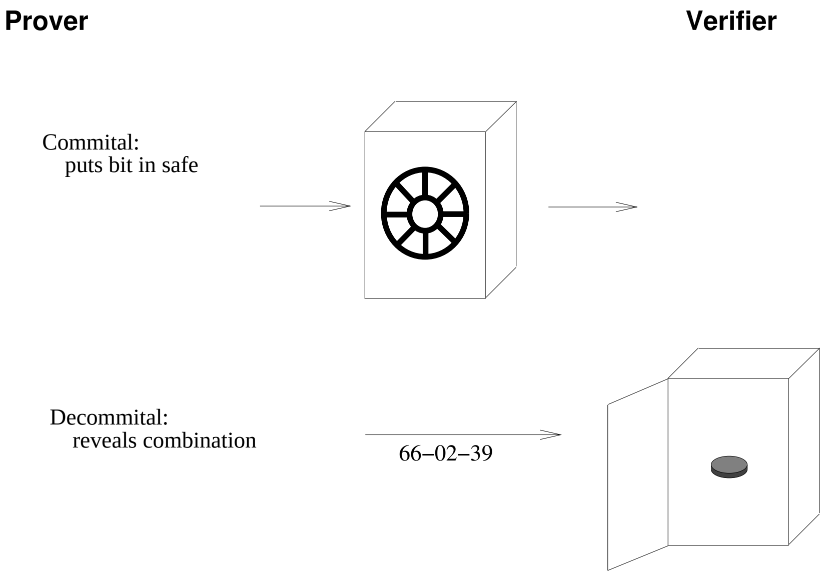
NEEDS REVISION

15 ZK for ALL of NP

15.1 ZK Proofs for all of NP under physical assumptions

In our protocolgraph non-isomorphism, the verifier had to convince the prover that he knew a certain bit. Now we'll look at a situation where the prover must commit to a bit and then reveal it later to the verifier. For the time being, let us implement this using a physical assumption – safes. Later, we'll show to how simulate this using one-way permutations. Figure 1 shows the protocol.

Figure 1: Bit commitment using a safe:



We'll use safes to exhibit a ZK protocol for Graph 3-Colourability (G3C), an NP-complete problem – see Figure 2. It is statistical zero-knowledge, since the conversations can be simulated as shown in Figure 3.

Figure 2: ZK proof for G3C:

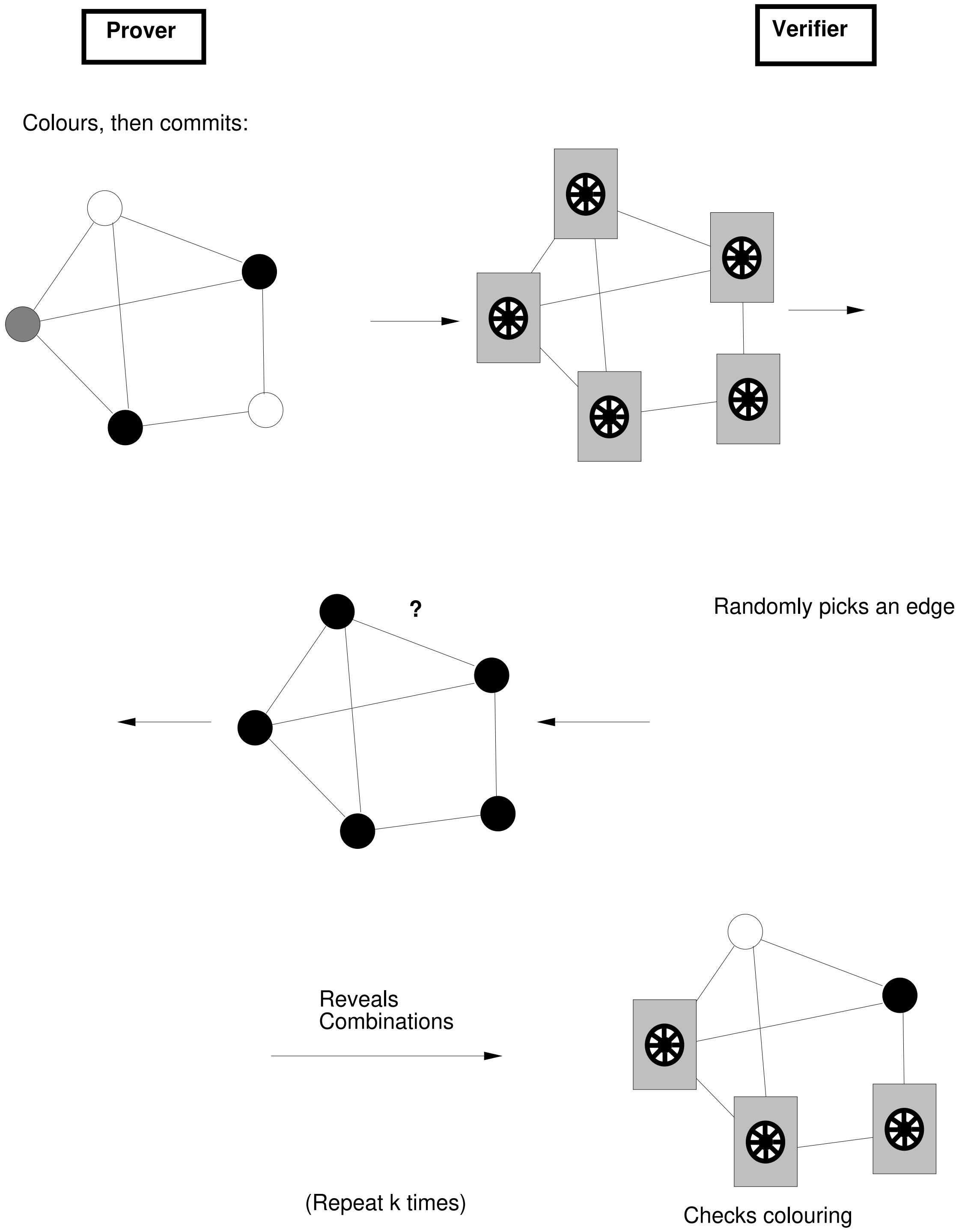
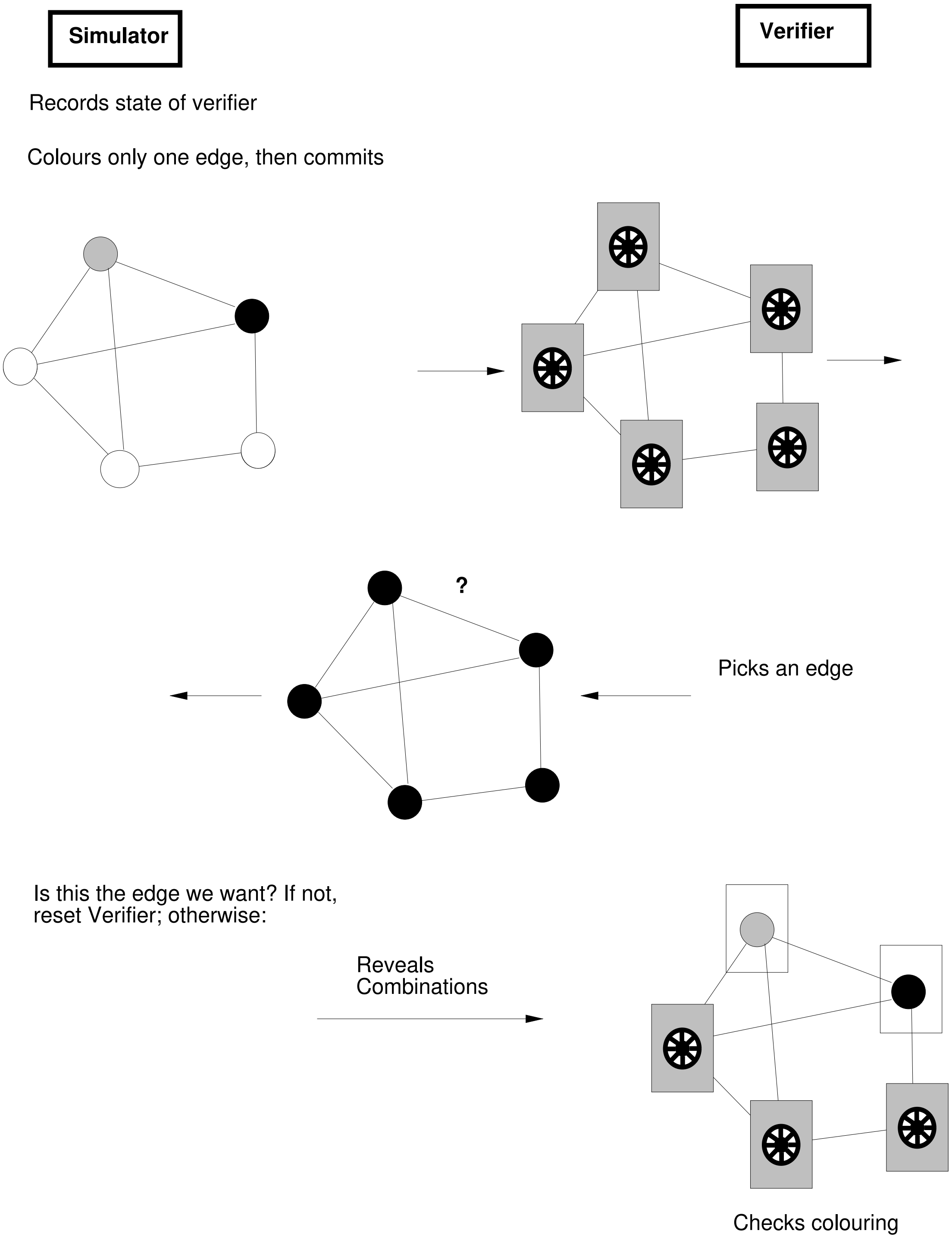


Figure 3: ZK simulator for G3C:



15.2 ZK proofs for NP using one-way permutations

15.2.1 Implementing safes

We'll show how to simulate a safe using a one-way permutation f:

Commital:

- P wants to commit to bit b. He randomly chooses p, x such that $|p| = |x|$.
- P sends $p, f(x), (x \cdot p) \oplus b$ to V.

Decommital:

- P sends x, b to V.

Why does this system work?

(1) It is perfectly binding – because f is a permutation, P is fully committed to x. He is thus committed to $(x \cdot p)$ and therefore to b.

(2) If the verifier (who operates in probabilistic polynomial time) can deduce any information about b, then he can invert f, as we showed in our proofs about hard-core bits.

It makes a big difference whether f is a uniform or non-uniform one-way permutation, as we'll see below.

15.2.2 ZK protocol for Graph 3-Colourability

Here is a more formal statement of the ZK protocol for G3C:

Prover	Communication	Verifier
C = 3-colouring of graph G		
Repeat k times:		
Permute(C)	$\rightarrow$ Commit to C $\rightarrow$	
	$\leftarrow u, v \leftarrow$	Pick an edge $e = (u, v) \in G$
	$\rightarrow$ Decommit colouring of u, v $\rightarrow$	Check colouring of u, v

Intuition for proof: If graph G has $|E|$ edges, then the probability that P doesn't know a coloring but manages to fool V is at most $(1 - \frac{1}{|E|})^k$. So for $k \sim \ell|E|$, this probability is less than $1/e^{\ell}$. The simulator $S_{\hat{V}}$ follows the usual pattern:

Do k times:

1. Pick an edge $e = (u, v)$ in G and color u, v differently. Assign all other vertices the same color.
2. Commit to the coloring of G.

3. Record the state of $\hat{V}$.
4. If $\hat{V}$ asks for an edge other than e, reset its state, and repeat step (4). Otherwise, decommit vertices u, v.

As before, this is a statistical ZK simulator, because it has a (small) chance of failure, but it can be made into a perfect ZK simulator by adding a low-probability exponential search.

The proof of the protocol is rather involved, and we'll touch on some of the important issues below.

15.2.3 Uniform vs. Non-Uniform Bit Commitment

One possible problem with the protocol is that we've assumed that our bit commitment scheme is secure in this context. Consider, for instance, what happens if it is based on a permutation f which is uniform one-way but not non-uniform one-way. That is, f (and thus the bit commitment) can be cracked with non-negligible probability by a family of poly-sized circuits (one circuit for each input size) or equivalently, by a polynomial time Turing machine which receives some additional "advice" as input (this advice being the description of the relevant circuit). Well, it might just happen that the advice that the verifier needs to break the bit commitment scheme is precisely the statement "G is 3-colourable"! Who knows? – this statement might, by some strange mapping, be exactly equivalent to the description of a circuit that can invert the function f. Thus it is crucial that f be non-uniform one-way.

15.2.4 ZK subroutines

Here's another difficulty – our interactive protocol for G3C consists of a loop which is repeated many times. We can show that a single iteration of this loop is zero-knowledge, but it is somewhat more complicated to show that the entire protocol is zero-knowledge, since with each successive iteration, the verifier has more information (specifically, the conversation from previous rounds). To handle this, we introduce a new term:

Definition An interactive protocol (P, V) for a language L is **auxiliary-input zero-knowledge** if $\forall$ verifiers $\hat{V}$ $\exists$ simulator $S_{\hat{V}} \in PPT$ such that:

$$\forall h,\ \forall x \in L, \quad P\hat{V}(x, h) \simeq S_{\hat{V}}(x, h)$$

Here, h corresponds to the past history between the prover and verifier. If a protocol is auxiliary-input zero-knowledge, then we'll see in a later lecture that it can be used as a subroutine without any problems. That is, it can be used in another program which calls it a polynomial number of times, and these multiple calls taken together will be auxiliary-input zero-knowledge.

15.2.5 Main proof

The core of the proof consists of the following:

Claim Say the bit-commitment scheme is based on a permutation f. If for infinitely many graphs, the verifier can often (with non-negligible probability) decide if a committed (hidden) graph coloring is correct, then f is not non-uniform one-way.

Proof Intuition. Let's first restate the claim. Without loss of generality, assume there's a infinite sequence of graphs $G_1, G_2, \ldots$ where the size of G_i is i, on which the verifier V can often distinguish a hidden correct coloring from a hidden incorrect coloring. For each graph G_i, fix (1) a correct 3-coloring C_i and (2) an incorrect coloring C_i' in which the vertices of one edge are colored the same as in C_i and the remaining vertices all have the same color. Let our bit-commitment relation (a hard-core predicate of f) be $g(\cdot)$. With a slight abuse of notation, let $g^*(C_i)$ denote a committal of coloring C_i. So we know that $\forall i, V$ can often distinguish between $g^*(C_i)$ and $g^*(C_i')$. We'll use this fact to construct a family of poly-sized circuits P_i, each of which can invert $g(\cdot)$ on outputs of length i, with non-negligible probability.

Look at a specific pair (C_i, C_i'). Consider a sequence of colorings $C_i' = D_1, D_2, D_3, \ldots, D_k = C_i$ such that D_{j+1} is the same as D_j except for one vertex v_j, whose coloring is changed to that in C_i. Using our old hybrid argument, we can show that there is some j for which V can tell apart D_j and D_{j+1} with non-negligible probability. So here's our circuit P_i: it has G_i, D_j, D_{j+1}, and v_j hard-wired into it, and on input $I = g(b) = (f(x), (x \cdot p) \oplus b, p)$, it asks V if there's a difference between (1) D_{j+1} and (2) D_j with I substituted for $g(\text{coloring of } v_j)$. The circuit is right whenever V is.

The proof is slightly more complicated than this (for instance, the coloring of v_k actually uses two bits since there are three possible colors), but these are the main ideas behind it.

$\boxed{\text{NEEDS REVISION}}$

16 ZK: a fine print

Herein[18] we will cover the the following:

- Uniform versus non-uniform zero-knowledge: a danger
- Auxiliary-input zero-knowledge
- A compiler for honest zero-knowledge $\Rightarrow$ general zero-knowledge

16.1 Uniform versus non-uniform zero-knowledge

Thus far, we have implemented a secure bit-commitment mechanism for zero-knowledge proofs using one-way functions. A question remains as to whether we need uniform or non-uniform one-way functions for the proofs to actually be zero-knowledge.

It turns out we *must* assume our one-way function f is *non-uniformly* secure, i.e., that it is secure not only against polynomial-time adversaries, but also a family of circuits, one per input length.

Recall that a uniform one-way function cannot be broken by a polynomial time adversary, but can be broken by a family of circuits or a polynomial-time adversary with advice.

A non-uniform one-way function can't be broken by either.

As an example, in a zero-knowledge proof for graph-coloring, the description of the circuit that breaks the particular uniform f, or the advice necessary to break it, might be embedded in the (potentially very bizarre) graph itself.

Remark: Note that for all G of a certain input length, no advice will help.

16.2 Auxiliary-input zero-knowledge

Usually, a zero-knowledge proof is a subroutine in some larger application. Thus, much additional information might be available to the verifier. For example, the verifier might know how to color half of a certain graph. We now discuss how a verifier, V, gets no *additional* information, even given some auxiliary input.

Definition 16.53 An interactive proof PV for a language L is auxiliary-input zero-knowledge if:

$$\forall \hat{V}\ \exists S_{\hat{V}} \in \mathsf{PPT}\ \text{s.t.}\ \forall h\ \forall x \in L$$

$$[P\hat{V}(x,h)] \cong [S_{\hat{V}}(x,h)]$$

i.e. the distributions are indistinguishable.

[18] Scribe notes taken by: Todd Hodes, Ocotber 27, 1994

Here, the simulator has *two* inputs, where h is an arbitrary string: the auxiliary input.

The benefit of auxiliary-input zero-knowledge proofs is that they can be *combined*, and they stay zero-knowledge. For example, lets look at the special case where a single protocol is repeated multiple times. We first prove that the protocol is "good," i.e. that the probability of error is negligible, and then prove it is auxiliary-input zero- knowledge. Thus, assume we have a protocol PV that shows $x \in L$ with probability of $\frac{1}{2}$. Running this protocol Q times gives us a new protocol, $P_Q V_Q$.

Claim 16.54 If PV is auxiliary-input zero-knowledge, then $P_Q V_Q$ is also auxiliary-input zero-knowledge.

Proof:
Suppose $\exists$ a distinguisher that can distinguish distributions $P_1V_1, P_2V_2, P_3V_3, \ldots$ from the distributions $S_1, S_2, S_3, \ldots$, where the simulator S_n has the previous partial history as auxiliary input.

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

$$\begin{aligned} P_0 &= (P_1V_1)(P_2V_2)(P_3V_3)\ldots(P_kV_k) \\ P_1 &= S_1(P_2V_2)(P_3V_3)\ldots(P_kV_k) \\ P_2 &= S_1S_2(P_3V_3)\ldots(P_kV_k) \\ &\vdots \\ P_k &= S_1S_2S_3\ldots S_k \end{aligned}$$

We know $P_0 - P_k > \frac{1}{2^n}$, and therefore $\exists j$ such that $P_j - P_{j+1} > \frac{1}{k2^n}$, another $\frac{1}{poly}$ fraction. Consider a distribution:

$$P(\boxed{z}) = S_1S_2S_3\ldots S_j\boxed{z}(P_{j+1}V_{j+1})\ldots(P_kV_k)$$

If z is a sample from S_n then $P(z) = P_j$, and if z is a sample from P_nV_n then $P(z) = P_{j+1}$. Assuming we find $j+1$ and fix the other values correctly, such a distinguisher could be used to distinguish this *single sample,* $\boxed{z}$, which is a contradiction. ∎

16.3 A compiler for honest zero-knowledge ⇒ general zero-knowledge

It would be nice if we knew that a verifier following a zero-knowledge interactive proof protocol wouldn't "cheat" and try to obtain additional information from the prover. Since this is unlikely, we will now look at a procedure to take a honest ZK protocol and convert it into one where it is assured that no information is revealed. In other words, we will show how to convert an interactive proof which is zero-knowledge for an honest verifier into an interactive proof which is zero-knowledge for *any* verifier.

The first step in the procedure is to design a protocol where V must follow the instructions exactly, an *honest* zero-knowledge protocol. The second step is to "compile" this into a new protocol which is guaranteed to divulge no information, regardless of the actions of the verifier.

First, we describe a procedure developed by [Blum and Micali]: **flipping coins into a well**:

A person P_1 stands far enough away from a well so as not to be able to look into it. Another person, P_2, stands at the edge of the well. P_1 then flips coins into the well. P_2 can discern the result of the coin tosses by looking into the well, and also prevent P_1 from seeing them.

This can be formalized as follows:

		Tossing Coins Into a Well	
	P_1	communication	P_2
1		$\leftarrow$ commit(b_1) $\leftarrow$	Generate a random bit b_1
2	Generate a random bit b_2	$\rightarrow b_2 \rightarrow$	$r = b_1 \oplus b_2$

If multiple bits are needed, they can all be done in parallel, still using only two communications.

This procedure has the following properties:

1. P_2 cannot control the outcome of the coin flip
2. P_1 doesn't know the outcome of the coin flip

Note that V will never have to decommit her bit if they use a "good" commitment scheme.

Given this procedure, if we let $P_1 = P$ and $P_2 = V$, *Prover* can pick the coins for V. We must now enforce that V acts honestly. One way of doing this is by demanding that each time V sends a message m to P, V also sends a zero-knowledge proof of the following NP statement:

> "According to my committed secret random tape and previous random history, m is the message I was supposed to send."

However, how does a verifier commit to a prover, if the prover has arbitrary computational power? This is the topic of our next lecture.

NEEDS REVISION

17 Bit-Commitment (BC): two variants

17.1 Introduction

These notes[19] explain two versions of Bit Commitment and the construction of Bit Commitment protocols based on cryptographic protocols.

Let's recall the problem of Bit Commitment (BC). There are two communicating parties, a sender S, and a receiver R. BC takes place in two stages. First, in the *commit* stage, a bit b is committed to, then in the *reveal* stage the bit is revealed. In order to make this protocol effective we want it to possess the following two properties:

- R has no knowledge of the value of b until S wishes him to know it.
- S cannot change the value of his commitment, i.e., *decommit* to a different value after commiting to it.

There are two properties of a BC protocol:

1. Security: The complexity of R knowing the value of b, i.e., how well b is "hidden".
2. Binding: The complexity of S being able to "cheat" (change the value of his commitment without R detecting it), i.e., how "binding" is the commitment to the sender.

We will see two versions:

1. Computationally Secure/Perfectly Binding.
2. Perfectly Secure/Computationally Binding.

17.2 Computationally Secure/Perfectly Binding BC (CS/PB)

Computationally Secure/Perfectly Binding BC has the following properties:

1. After commitment b is well defined, i.e., sender will *never* be able to cheat and decommit both a 0 and a 1.
2. b is hidden only computationally.

Note: S can have arbitrary complexity but R must only be of polynomial-time complexity.

The following is an example of a CS/PB BC protocol. Let f be a one-way permutation. Let $HCB(x)$ be a Hard Core Bit of a string x generated using f, then the following is a CS/PB BC protocol of bit b:

[19]Scribe notes taken by: Elan Amir, October 25, 27 1994

S	communication	R
Generate a random string x. Let $c = b \oplus HCB(x)$.		
Commitment	$\rightarrow f(x), c \rightarrow$	
Decommitment	$\rightarrow x, b \rightarrow$	Verify $c = b \oplus HCB(x)$.

Let's examine the protocol in detail and see why it is in fact CS/PB. In order to cheat the sender has to be able to find a value of x with the property that

$$HCB(f^{-1}(f(x))) = \{0, 1\}$$

But f is a permutation so this is impossible. Therefore the protocol is Perfectly Binding. On the receiver's side, in order for the receiver to determine b from the information he is given, he needs to determine the value of $HCB(x)$ which is computationally difficult for a polynomial-time receiver. Therefore, the protocol is Computationally Secure.

17.2.1 Extending CS/PB BC Protocol Construction to all One-Way Functions

We have seen a CS/PB BC protocol that requires the use of one-way permutations. We now show that such a protocol can be devised using *any* one-way function f.

The first step of the construction uses the fact established in [ill] that any one-way function can be used to build a Pseudo-Random Number Generator (PRG). [naor] completes the construction with the result that any PRG can be used to construct a CS/PB BC protocol. We now prove this result.

Let $G : \{0,1\}^n \rightarrow \{0,1\}^{3n}$ be a PRG. Let $C(g, r) = c, \quad g, r, c \in \{0,1\}^{3n}$ where the bits of c are defined as follows:

$$c_i = \begin{cases} g_i \oplus b & \text{if } r_i = 1 \\ g_i & \text{if } r_i = 0 \end{cases}$$

Now consider the following BC protocol of bit b.

S	communication	R
	$\leftarrow r \leftarrow$	Choose a random $3n$-bit string r.
Choose a random n-bit seed s.		
Let $g = G(s), c = C(g, r)$		
Commitment	$\rightarrow c \rightarrow$	
Decommitment	$\rightarrow b, s \rightarrow$	Verify c.

Let's examine the properties of this protocol. First, we claim that the sequence c is still pseudo-random. To see this, observe that if $b = 0$, then c is just the output of a PRG. If $b = 1$, then c is the output of a PRG with a random set of flipped bits. The latter case is pseudo-random since if it was not, i.e., we could construct a distinguisher to distringuish between c and a truly random value, we could use this distinguisher to distinguish all pseudo-random numbers, which contradicts the assumption of PRG existence. Since c is pseudo-random, b is computationally hidden, and the protocol is Computationally Secure.

Next, for a given r, consider what the sender needs to do in order to cheat. He must find two seeds s_1 and s_2 such that $G(s_1)$ agrees with $G(s_2)$ on all bit positions where $r_i = 0$ and disagrees with $G(s_2)$ on all bit positions where $r_i = 1$. A simple counting argument will show that given a random r the probability of the existence of such a pair is exponentially small. We observe that each pair of seeds corresponds to a single choice of r (since the bits of r are defined according to the corresponding bits of s_1 and s_2 as described above). Next, notice that there exist 2^{2n} pairs of n-bit seeds. The correspondence between seed pairs and r implies that there exist at most 2^{2n} values of r for which there exist a pair of seeds s_1 and s_2 that can be used to cheat. However, there exist 2^{3n} $3n$-bit strings for r so that the probability that given a random r there exist a pair of seeds s_1 and s_2 that can be used to cheat is $2^{2n}/2^{3n} = 2^{-n}$.

Therefore we have have shown that any one-way function can be used to construct a CS/PB BC protocol.

17.3 Perfectly Secure/Computationally Binding BC (PS/CB)

Perfectly Secure/Computationally Binding BC has the following properties:

1. After commitment, b is perfectly hidden, i.e.,

$$\forall \text{conversations } c \text{ of commitment}, P(c|b=0) = P(c|b=1)$$

   where the probabilities are over the coin-flips of S and R.

2. A polynomial-time sender can not cheat, i.e., decommit to both a 0 and a 1.

Here we note that S must be of polynomial-time complexity while R can have arbitrary complexity.

Before moving on to an example of a PS/CB BC protocol we introduce the notion of claw-free functions. Informally, two one way permutations f_0, f_1 are **claw-free** if it is computationally intractable to find in polynomial-time two values x_0, x_1 such that $f_0(x_0) = f_1(x_1)$.

We are now ready for a PS/CB BC protocol. Let f_0, f_1 be two claw-free one-way permutations. Then the following is a PS/CB BC protocol of bit b:

S	communication	R
Generate a random string x. Let $z = f_b(x)$.		
Commitment	$\rightarrow z \rightarrow$	
Decommitment	$\rightarrow x, b \rightarrow$	Verify $z = f_b(x)$.

Why is this PS/CB? In order for the sender to cheat, he has to find two values x_0, x_1 such that $f_0(x_0) = f_1(x_1)$, that way when he decommits he can send x_b. But this property contradicts the fact that $f_{0,1}$ are claw-free for a polynomial-time sender, so this protocol is Computationally Binding. Note that the alternative of finding a value of z such that $f_{0,1}(x) = z$ is impossible since $f_{0,1}$ are permutations. On the receiver's side, since $f_{0,1}$ are permutations, there exist x_0, x_1 such that $f_0(x_0) = f_1(x_1) = z$, therefore only the knowledge of z does not enable the receiver to obtain any information on b since the sender could have computed z with $b = 0$ or $b = 1$ with equal probability. Therefore the protocol is Perfectly Secure.

So we see that the above protocol is PS/CB providing that we can find claw-free functions. We now show a Number Theoretic construction of a PS/CB BC protocol which uses a pair of such functions.

Let b be a bit to be committed. Then the following protocol is PS/CB.

S	communication	R
		Generate 2 large primes p_0, p_1. Let $n = p_0 p_1$ Generate a random string w. Let $s = w^2 \bmod n$ Generate k random values v_i, $1 \le i \le k$. Let $t_i = v_i^2 \bmod n$
	$\leftarrow n, s, t_1, t_2, \ldots, t_k \leftarrow$	Proof the s is a square
Choose k bit values $c_i, 1 \le i \le k$	$\rightarrow c_1, c_2, \ldots, c_k \rightarrow$	
	$\leftarrow \sqrt{t_1 s^{c_1}}, \ldots, \sqrt{t_k s^{c_k}} \pmod{n} \leftarrow$	End proof that s is a square.
Commitment. Choose a large number r. Define $M_b(r) = s^b r^2 \bmod n$	$\rightarrow M_b(r) \rightarrow$	
Decommitment	$\rightarrow (b, r) \rightarrow$	Verify $M_b(r) = s^b r^2$.

The proof rests assumes the fact that factoring cannot be be done in polynomial time. First, let's examine the proof of the fact that s is a square. Note that if s is not a square then if $b_i = 1$, $\sqrt{t_i s^{c_i}}$ will not be defined unless t_i is not a square and the product $t_i s^{c_i}$ is a square. In that case, when $b_i = 0$, $\sqrt{t_i}$ will not be a square. So we see that the probability of R being able to cheat on the fact that s is a square is 2^{-k}.

Next, notice that since s is square, both values that the sender could send are squares, i.e., in the quadratic residue of n, and therefore have the same distribution. As a result, the receiver learns no information on the value of b from the commitment which implies that the protocol is Perfectly Secure. From the sender's point of view, in order for him to cheat, he must be able to factor s, since then he would be able to send the pair $(b, r\sqrt{s})$ or the pair (b, r), depending on what he chose to decommit to. However, since we assume that factoring s is computationally impossible for a polynomial-time machine, the fact that the sender has only polynomial complexity assures that the protocol is Computationally Binding. Note that the above discussion shows that M_b is a pair of claw-free functions.

17.3.1 Construction of a PS/CB BC using any One-Way Permutation

We now show a PS/CB BC protocol that removes the restriction of claw-free functions. The following protocol is presented in [novy]. The sender wishes to commit to a bit b. Let $B(x, y) = x \cdot y \pmod{2}$, $x, y \in \{0,1\}^n$. Let f be a one-way permutation on $\{0,1\}^n$.

The following is the commitment stage:

1. S generates a random n-bit string x and computes $y = f(x)$.

2. R selects a linearly independent set of n-bit vectors $\{h_1, h_2, \ldots, h_{n-1}\}$.

3. For j from 1 to $n-1$

   - R sends h_j to S.
   - S sends $c_j = B(h_j, y)$ to R.

4. The $n-1$ iterations of the above step define $n-1$ linearly independent equations of the form $c_j = B(h_j, y)$. Therefore there are exactly two n-bit vectors y_0, y_1 that are solutions to these equations. Define y_0 to be the lexicographically smaller of the two vectors. Both S and R compute y_0 and y_1 (this does not place any restrictions on the complexity of S or R since a set of linear equations can be solved in polynomial time). Observe now that y is equal to either y_0 or y_1. Let
$$c = \begin{cases} 0 & \text{if } y = y_b \\ 1 & \text{if } y = y_{1-b} \end{cases}$$

5. S computes c and sends it to R.

To decommit S sends b and x to R who verifies the computation of c and that in fact $y = f(x)$ solves the set of equations.

The above protocol is clearly perfectly secure. How do we show that it is computationally binding? Observe that in order to cheat the sender must be able to know $f^{-1}(y_0)$ *and* $f^{-1}(y_1)$ since then he will be able to send the appropriate inverse corresponding to the bit value he wishes to decommit to according to the definition of c. However, we now show that this is impossible. In particular, we show that if the sender can invert both solutions to the equations, we can use that sender as a subroutine to an algorithm which can invert a one-way function. The construction of such an algorithm is as follows.

Define the I_f to be the following algorithm which "communicates" with a cheating sender $\hat{S}$. The input to I_f is an n-bit string w. $B(x, y)$ is defined as before.

The core of the algorithm is the following loop:

1. Record the entire state of $\hat{S}$.

2. Pick h_1 at random and send it to $\hat{S}$.

3. Upon receipt of c_1 from $\hat{S}$ check if $B(h_1, w) = c_1$ If so, proceed to h_2 (linearly independent of h_1), otherwise reset $\hat{S}$ to the state recorded in (1) and goto (2), i.e., choose a new h_1.

I_f continues this loop until either it succeeds $n-1$ times, i.e., it accepts $h_1, h_2, \ldots, h_{n-1}$, or it fails n times, i.e., it needs to choose a new h_i in step (3) n times over the runtime of the algorithm (not necessarily consecutively).

Here is an idea of what happens (the actual proof is more complicated): We now examine the state of I_f when it terminates. In the first case, we have obtained $n-1$ linear equations to which w is a solution and which we know that $\hat{S}$ holds the inverse to, since it computed its c_i's using the same check we performed in step (3). Therefore, at the decommital stage $\hat{S}$ will send us $f^{-1}(w)$. In a sense we have "forced" $\hat{S}$ into inverting our w by constraining the equations to contain w as a solution, thereby obtaining the inverse by the assumption that $\hat{S}$ could invert both solutions. In the latter case, we observe that if $\hat{S}$ has avoided w n times this means that we already have n equations which are sufficient to *solve for* w. This means that $\hat{S}$ has already guessed w, which ((can be shown to happen over the choice of w) with negligible probability. We see therefore that if S can cheat, we can invert a one-way function. Therefore, the protocol is Computationally Binding.

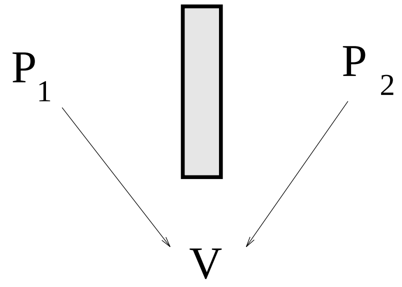
NEEDS REVISION

18 Two-prover Model

18.0.2 Setup

All of our previous [20] discussions of zero-knowledge proofs have been based on the notion of "secure envelopes" or "locked safes," where we can place a bit into a container in a *commitment* stage and then only open the envelope in a *decommitment* stage. We will now examine a **two-prover model** for zero-knowledge proofs, first described by [Goldwasser, Benor, Wigderson, and Kilian].

The basic framework for the two-prover model consists of two (or more) provers (P_1, P_2) who are unable to communicate with each other, but who can communicate with a Verifier (V) using a prescribed protocol. Graphically, the model looks like the following:



We could imagine many ways of implementing this. For example:

- The two provers could be devices (such as ATM cards) that are inserted into the verifying device (such as an ATM machine). By having the verifier physically isolate the two provers, it can be sure no communication occurs.
- The two provers could simply have a wall between them. We illustrated this last version for the sake of simplicity.

18.0.3 Implementation of Bit-Commitment (BC)

So far, we have shown that there exist zero-knowledge proofs for all languages in NP if we have bit-commitment. So, in order to implement zero-knowledge proofs in the two-prover model, we can simply show how to implement bit-commitment, and all the other machinery follows from there.

The nice properties of the two-prover model is that it model is "perfect:" no cheating is possible, and everything is hidden information-theoretically. Also note that there are no cryptographic assumptions here, only physical ones.

First we'll describe a slightly broken protocol, *Weak Two-Prover Bit-Commitment*, where we allow the provers to cheat with a probability of $\frac{1}{2}$. Then we'll describe how to fix it to

[20]Scribe notes taken by: Todd Hodes, November 1 1994

obtain *Strong Two-Prover Bit-Commitment*, where the provers chances of cheating can be made arbitrarily small (negligible).

18.0.4 Weak Bit-Commitment

The following is a protocol for *Weak Two-Party Bit-Commitment.*

The provers P_1 and P_2 have a bit b they want to commit. They choose two bits b_0 and b_1 such that $b_0 \oplus b_1 = b$. V then picks a bit i, sends it to P_1, and asks P_1 to return b_i. This is the *commitment* stage. V then asks for b_0 and b_1 from P_2, and verifies the reply. This is the *decommitment* stage.

In tabular form:

Assume P_1 and P_2 have chosen b, a bit to commit, and a set of bits b_0 and b_1 such that $b_0 \oplus b_1 = b$.

			Weak Two-Party Bit-Commitment		
	P_1	communication	V	communication	P_2
1		$\leftarrow i \leftarrow$	Generate a random bit i		
2		$\rightarrow b_i \rightarrow$	(commitment ends)		
3			de-commit phase	$\leftarrow b_0$ and $b_1 \leftarrow$	
4			$b = b_0 \oplus b_1$		

We see that the provers can cheat by changing either b_0 or b_1, but not both. The cheating will go unnoticed with probability $\frac{1}{2}$.

18.0.5 Strong Bit-Commitment

We now show how to implement *Strong Two-Party Bit-Commitment* using the weak protocol as a subroutine. We simply repeat the procedure for weak bit-commitment k times, and the probability of tricking the verifier decreases to $\frac{1}{2^k}$. This probability can be made arbitrarily small by varying k.

The protocol is illustrated as follows:

Assume P_1 and P_2 have chosen b, the bit to commit, and k pairs of bits (b_0, b_1), (c_0, c_1), $(d_0, d_1), \ldots$ such that $b = b_0 \oplus b_1 = c_0 \oplus c_1 = d_0 \oplus d_1 = \ldots$.

			Strong Two-Party Bit-Commitment		
	P_1	communication	V	communication	P_2
1		$\leftarrow i_0, i_1, i_2, \ldots \leftarrow$	Generate a random vector of k bits $i_0, i_1, i_2, \ldots$		
2		$\rightarrow b_{i_0}, c_{i_1}, d_{i_2}, \ldots \rightarrow$			
3				$\leftarrow (b_0, b_1), (c_0, c_1), (d_0, d_1), \ldots \leftarrow$	
4			$b = b_0 \oplus b_1 = c_0 \oplus c_1 = d_0 \oplus d_1 = \ldots$		

Again, the provers can cheat by changing either b_0 or b_1, and either c_0 or c_1, and either d_0 or d_1, etc. But this time, the cheating will go unnoticed with a probability of only $\frac{1}{2^k}$.

18.0.6 Some extensions

It was mentioned earlier that IP = PSPACE. Incidentally, MIP = NEXP. That is, the set of languages with multiple-prover interactive proofs is the same as the set of languages where membership can be determined in non-deterministic exponential time (!). This is clearly a *huge* class of languages.

Additionally, multiple-prover interactive proofs with more than two provers can be simulated by two-prover interactive proofs. Showing how this simulation is done is left as an exercise.

Finally, a quick note of the importance of the class MIP. Multiple-prover interactive proofs are quite useful for two important reasons:

- The model gives a method to obtain strong complexity results, like PCP, but this is outside the scope of this course
- They provide information-theoretical security
- They translate into identification schemes

One example of the usefulness of MIP is a mentioned above two-card ATM scheme, where no information is revealed if the cards are not able to communicate.

NEEDS REVISION

19 Introduction to Oblivious Transfer

19.1 Oblivious Transfer

In this lecture we introduce the oblivious transfer primitive[21]. The concept of **oblivious transfer** was developed by [Rabin, 79] for digital signatures.

Here is a story to explain the concept: Imagine I have a very smart parrot that is old and about to die. Also imagine that I have a bit b I want to commit to my friend. I whisper the value of the bit to the parrot. The parrot flies over to my friend, and with probability $\frac{1}{2}$ tells my friend the value of b, or with probability $\frac{1}{2}$ dies and tells my friend nothing.

The sender (me) has no idea if the output was either the bit b or nothing. The receiver (my friend) does know if she gets the bit or not.

Thus, if we indicate the case where the receiver gets the bit as b, and the case where she gets nothing as $\#$, then

$\Pr(b) = \Pr(\#) = \frac{1}{2}$.

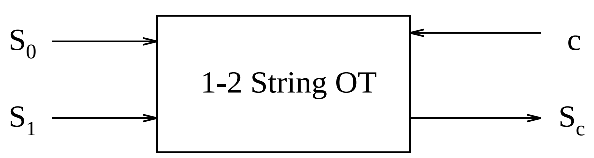
In a sense, one can simply think of OT as a noisy channel.

This game is useful in that it can be used to play many other games; it can be thought of to be "complete" in this sense. We will now describe some other such games.

19.1.1 One-out-of-two string oblivious transfer

This game comes from [Even, Goldreich, Lempel] and can be abbreviated 1-2-String OT.

Imagine a traitor, T, who is trying to sell one of two secrets to some enemy of our country. The enemy, E, doesn't want T to know which secret he or she has purchased. If we call the two secrets S_0 and S_1, and the selection bit c, then the situation looks like this:

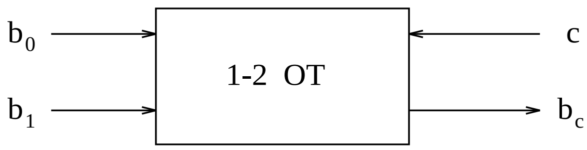


More formally, a sender inputs two strings, S_0 and S_1, while a receiver inputs a bit c, and gets as output S_c. At the end of the game, the sender doesn't know anything about the selection bit c, and the receiver doesn't know anything about message S_{1-c}.

19.1.2 One-out-of-two oblivious transfer

This game is nearly identical to 1-2-String OT, but instead of having T input strings S_0 and S_1, he or she inputs single bits b_0 and b_1. The output in this case is the single bit b_c. 1-2 OT looks like so:

[21]Scribe notes taken by: Todd Hodes, November 1, 1994



Again, the sender doesn't know anything about the selection bit c, and the receiver doesn't know anything about message S_{1-c}.

It has been shown by [Crepeau] that OT $\Leftrightarrow$ 1-2 OT. The constructions for both directions are as follows:

Claim 19.55 1-2 OT $\Rightarrow$ OT

Proof: We prove by construction.

1. sender picks bit i at random and sets $S_i = b$ and $S_{1-1} = 0$.
2. Run the 1-2 OT protocol with inputs S_0, S_1.
3. Have the sender tell the receiver which S_i had the actual message.
4. Thus, with probability $\frac{1}{2}$, R gets the bit b, and with probability $\frac{1}{2}$, R gets nothing. This is exactly the outcome of OT.

■

Now for the other direction...

Claim 19.56 OT $\Rightarrow$ 1-2 OT

Proof: Again, we prove by construction.

1. S generates a random string of bits $c_1 \ldots c_{poly}$, and sends it to the receiver over the OT channel.
2. R gets something of the form $c_1 c_2 \# \# c_5 \# \ldots c_{poly}$ at the other end of the OT channel
3. R selects two disjoint subsets of $\frac{1}{3}$ of the indices from $1 \ldots$ *poly*; one of these sets should only contain indices of bits that were successfully received.
4. R sends both sets of indices to S in the clear.
5. S then sends $S_0 \oplus$(all the bits in the first subset), and $S_1 \oplus$(all the bits in the second subset.)
6. Using the Chernoff Bound, the probability of successfully getting either more than $\frac{2}{3}$ or less than $\frac{1}{3}$ bits can be made negligible:

   $$Pr[|\sum X_i - E(X)| \geq \frac{E(X)}{3}] < e^{-\frac{E(X)}{18}}$$

   Thus, with overwhelming probability, R can construct one subset with all of the bits known, but can't construct a second subset with all bits known. In this case, just a single message will get through, which is exactly the outcome of 1-2 OT.

■

NEEDS REVISION

20 More on Oblivious Transfer, Connetction to ZK

20.1 OT $\Longrightarrow$ BC

Recall[22] that in Oblivious Transfer (OT), S has a bit that is sent to R. R either receives that bit, or receives the # sign, which is interpreted as knowledge that a bit was sent, but no knowledge of what that bit was. Each possible outcome occurs with probability $\frac{1}{2}$, and S has no knowledge of what R actually received. Recall also that we showed OT to be equivalent to 1-2-OT, where S has two bits b_0 and b_1, and R has a bit c. R receives b_c, but S has no knowledge of which bit R received.

We will show that given a black box for performing OT, we can perform Bit Commitment (BC) [kilian]. Say that S wishes to commit one bit b to R. S chooses $b_1, b_2, b_3 \ldots b_n$ such that $b = b_1 \oplus b_2 \oplus b_3 \oplus \ldots b_n$. Then, S sends $b_1, b_2, b_3 \ldots b_n$ to R through the OT channel. With probability $1 - \frac{1}{2}^n$, at least one of the bits b_i does not go through, and thus the bit b will be information theoretically hidden.

Decommitment is performed by S sending all the bits $b_1, b_2, b_3 \ldots b_n$ in the clear (i.e. through a channel such that they all get through). Sending an incorrect value for b requires changing at least one of the bits $b_1 \ldots b_n$. But, since S does not know which bits actually were correctly received by R, changing any bit b_i in the decommitment stage will result in being caught with probability $\frac{1}{2}$. This can be amplified by performing the commitment many times, where the value of the committed bit has to be the same every time.

20.2 OT + PRG $\Longrightarrow$ Communication efficient ZK

We first define what communication efficient ZK is (due to [kilian,micali,ostrovsky]). This is motivated by the story of the traveling Theoretical Computer Scientist who proves theorems during his travels. He is only able to send postcards home. In order to ensure that he gets credit for his work, he sends Zero Knowledge proofs on these postcards, but since he does not receive replies, the Zero Knowledge proofs cannot be interactive, and hence the term communication efficient.

So, say there exists a Prover P, and a Verifier V. They are allowed k rounds of communication in a pre-processing stage. During the this stage, neither the prover nor the verifier knows what the prover will be proving, (which corresponds to the traveler not knowing what he will prove before he leaves for his trip). After the pre-processing stage, all further communications from P to V will not be interactive. We will show an implementation where P is able to send a polynomial number of Theorems, of the form (Theorem 1, Proof 1).

To help explain this technique, we first give a ZK proof for the existence of a Hamiltonian Cycle on graph G, due to Manuel Blum:

Blum's protocol:

[22] Scribe notes taken by: Micah Adler, November 3, 1994

		$x = \{G \text{ is Hamilonian}\}$	
	P	communication	V
1	Generate Π, a random permutation of G	$\rightarrow$ Commitment of Π $\rightarrow$	
2	Find $C(\Pi(E))$, a cycle on the edges of $\Pi(G)$	$\rightarrow$ Commitment of $C(\Pi(E))$ $\rightarrow$	
3		$\leftarrow b \leftarrow$	Generate b, a random bit
4	if $b = 0$	$\rightarrow$ Decommitment of Π $\rightarrow$	
5	if $b = 1$	$\rightarrow$ Decommitment of $C(\Pi(G))$ $\rightarrow$	

Each iteration of this technique can only succeed with probability $\frac{1}{2}$ if the graph is not Hamiltonian, and thus the probability of being able to cheat can be made very small by repeated iterations.

In order to give a communication efficient ZK proof for this problem, we will use 1-2-String OT, and Pseudo Random Generation (PRG). Recall that in PRG, function f takes a string s and returns a string $f(s)$ that has length polynomial in s, such that $f(s)$ is difficult to distinguish from a truly random string.

The pre-processing phase proceeds as follows: P selects pairs of strings $(a_0, a_1), (b_0, b_1), \ldots (z_0, z_1)$, and V selects bits $b_1, b_2, \ldots b_{26}$. P and V then use 1-2-String OT to transmit $a_{b_1}, b_{b_2}, \ldots z_{b_{26}}$ to V. Thus, V will know exactly one string of each pair, but P will not know which one. These strings can then be used as the seeds to a PRG, f. Then, for each pair (i_0, i_1), P can send two messages M_0 and M_1, as $M_0 \oplus f(i_0)$ and $M_1 \oplus f(i_1)$, where $M_0 \oplus f(i_0)$ represents the strings M_o and $f(i_0)$ bitwise XORed together. V, who also has access to the function f will be able to decode exactly one of M_0 and M_1, but P does not know which one.

A communication efficient ZK proof for Hamiltonian Cycle then works as follows. P sends, in the clear, a commitment of Π, a permutation on G and a commitment of $C(\Pi(G))$, a Hamiltonian cycle on the edges $\Pi(G)$. Then, P sends Π XORed with $f_{a_0}(theorem - number)$, and $C(\Pi(G))$ XORed with $f_{a_1}(theorem - number)$. Thus, since V will be able to decode exactly one of them, but P does not know which one, P has only a $\frac{1}{2}$ probability of being able to cheat if the graph is not Hamiltonian. But, since we had several such pairs of strings, this can be boosted by repeating this process for the other pairs of strings. Since the pseudo random functions can be used polynomially many times in the length of the original string, we can perform this same procedure for a polynomial number of proofs.

NEEDS REVISION

21 Completeness Theorem for multi-party computation

21.1 Completeness Theorem for the Honest But Curious model

In the honest but curios model[23] several parties participate in a communication protocol. The parties are assumed to follow any directions exactly, but will make any use of information that they can, include sharing information with other parties.

A completeness theorem is a method of computing any poly-time function between two or more parties subjetc to a privacy constraint. Let us first consider two-party setting. Say party A has input x, and party B has input y, and they wish to compute $f(x,y)$. The goal is for both parties to see the value $f(x,y)$, but for neither to gain any additional information about the input held by the other party. An example of this is two millionaires trying to find out who is richer, neither of which wishes to let the other know how much money they have. In the multiparty case, an example is seeing election results without revealing individual votes.

We will assume that f is any function computable by a polynomially sized circuit. First, note that any circuit can be converted to a similarly sized circuit consisting of only XOR gates and AND gates. For example, NOT(x) could be converted to XOR(x,1).

Now, assume that P_1 has input $a_1 a_2 \ldots a_m$, P_2 has input $b_1 b_2 \ldots b_m$, and so on up to P_n. Now, $\forall i, 1 \leq i \leq m$, P_1 chooses a representation of $a_i = x_{i1} \oplus x_{i2} \oplus x_{i3} \ldots x_{in}$. Then, P_1 sends x_{i2} to P_2, x_{i3} to P_3, and so on. Every other party also sends the corresponding bits to every other party. Thus, every party has a part of every bit, but the bit is information theoretically hidden from every proper subset of the parties.

We next describe a simple example of the two party case. Say party A has the bit a and party B has the bit b, and they wish to compute $a \oplus b$. In this example, at the end of the algorithm, both parties will be able to deduce the other persons bit, but once we can compute both the XOR and the AND, we will be able to build every circuit. A chooses a_1 and a_2 such that $a = a_1 \oplus a_2$ and sends a_2 to B. Also, B chooses b_1 and b_2 such that $b = b_1 \oplus b_2$ and sends b_1 to A. A calculates $c_1 = a_1 \oplus b_1$ and tells c_1 to B. At the same time, B calculates $c_2 = a_2 \oplus b_2$ and tells c_2 to A. Thus, both parties will know $a \oplus b$.

[23]Scribe notes taken by: Micah Adler, November 3, 1994

		Computing $a \cdot b$	
	A	communication	B
1	A knows a		B knows b
2	Chooses a_1, a_2 s.t. $a_1 \oplus a_2 = a$		Chooses b_1, b_2 s.t. $b_1 \oplus b_2 = b$
3		$\rightarrow a_2 \rightarrow$	
4		$\leftarrow b_1 \leftarrow$	
5	Calculates $c_1 = a_1 \oplus b_1$		Calculates $c_2 = a_2 \oplus b_2$
6		$\rightarrow c_1 \rightarrow$	
7		$\leftarrow c_2 \leftarrow$	
4	A knows $c_1 \oplus c_2 = a \oplus b$		C knows $c_1 \oplus c_2 = a \oplus b$

We next show how to compute an AND gate. First note that we can convert our circuit to binary arithmetic: $\mathrm{XOR}(a_1, b_1) = a_1 + b_1 \bmod 2$ and $\mathrm{AND}(a_1, b_1) = a_1 \cdot b_1$. Again, A holds a_1 and b_1, and B holds a_2 and b_2. They both wish to calculate $(a_1 \oplus a_2) \cdot (b_1 \oplus b_2) = a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2$. The first two terms are calculated by A and the second two by B. We describe here A's calculation; B's is similar.

Calculating $a_1 \cdot b_1$ is easy, since both bits are known to A. To calculate $a_1 \cdot b_2$, we will use 1-2-OT. First, B picks r to be a random bit. Then, B sets $v_0 = 0 \cdot b_2 - r$ and $v_1 = 1 \cdot b_2 - r$. Then, A chooses one of v_0 and v_1 based on a_1 using 1-2-OT. Thus, A receives the bit v_{a_1} without giving B any information. So, A knows the bit $a_1 \cdot b_2 - r$, and B knows the bit r, which can then be later added (XORed) together to obtain the bit $a_1 \cdot b_2$. But, since A does not know the value of r, A has not received any information.

		Computing $a_1 \cdot b_2$	
	A	communication	B
1	A knows a_1		B knows b_2
2			Pick r, a random bit, and let $v_0 = 0 \cdot b_2 - r$, and $v_1 = 1 \cdot b_2 - r$
3	Using 1-2-String OT	$\leftarrow v_{a_1} \rightarrow$	Using 1-2-String OT
4	A knows $a_1 \cdot b_2 - r$		B knows r

The multi-party case is similar. In other words, when we wish to know

$$c = a \cdot b = \sum_{i=1}^{n} a_i \cdot \sum_{i=1}^{n} b_i =$$

$$\sum_{1 \le i = j \le n} a_i \cdot b_j + \sum_{1 \le i \ne j \le n} a_i b_j,$$

then the first set of terms can all be computed locally, and the second set can all be done with the OT trick.

We now have the ability to play poker over the phone, since shuffling can be simulated as a pseudo random function of seeds given by different players, and playing a hand can be done by computing other functions.

21.2 Cheating parties

(to be completed)

$\boxed{\text{NEEDS REVISION}}$

22 Digital Singnatures

presented definition of existential, adaptive, chosen message attack security and [Naor,Yung] signature scheme. The scribe must be totally re-written and is deleted.

$\boxed{\text{NEEDS REVISION}}$

23 Full Information Model

Presented [Ostrovsky,Rajagopalan,Vazirani] paper, scribe should be completely re-written, and is deleted.