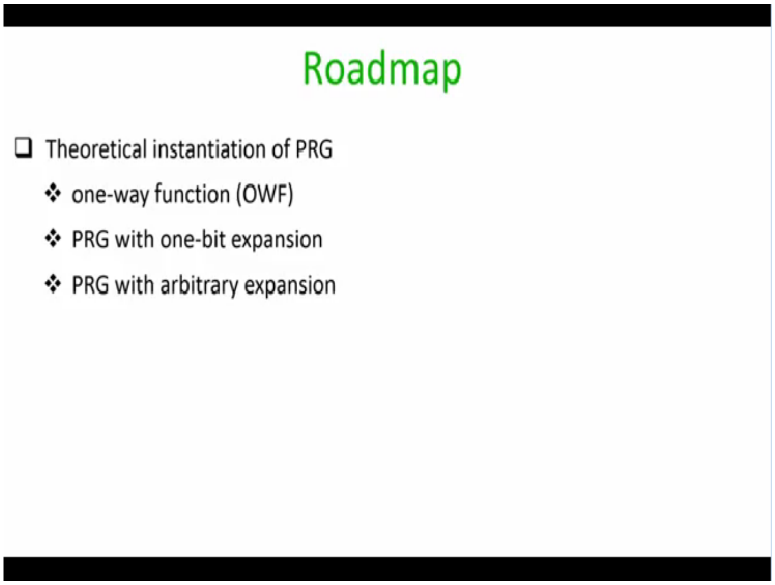
Foundations of Cryptography

Prof. Dr. Ashish Choudhury
(Former) Infosys Foundation Career Development Chair Professor
International Institute of Information Technology-Bangalore

Lecture-11
Provably Secure Instantiation of PRG

Hello everyone, welcome to lecture 10, the plan for this lecture is as follows.

(Refer Slide Time: 00:33)

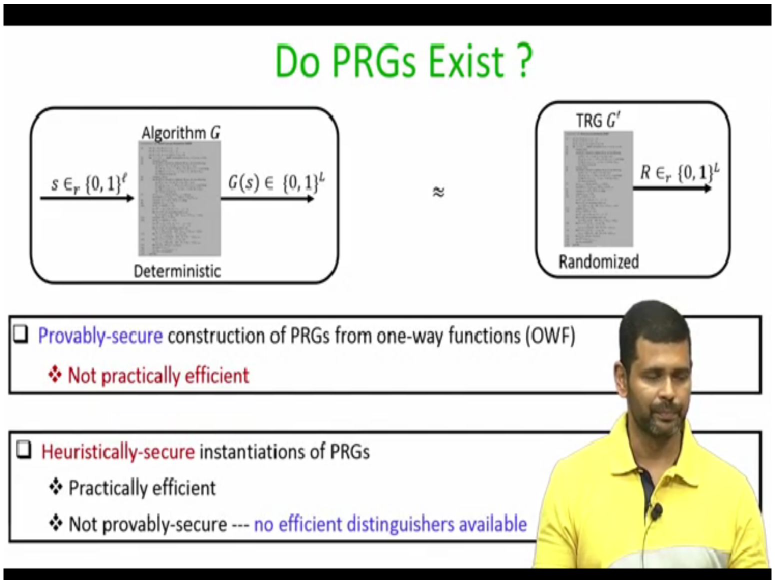


Roadmap

❑ Theoretical instantiation of PRG
- ❖ one-way function (OWF)
- ❖ PRG with one-bit expansion
- ❖ PRG with arbitrary expansion

In this lecture, we will discuss about the theoretical instantiation of pseudo random generators for which we will introduce the concept of one way function. And after introducing the concept of one way function, we will see how to design PRG with one bit expansion and then we will see how to design PRG with arbitrary expansion.

(Refer Slide Time: 00:53)



Do PRGs Exist ?

❑ Provably-secure construction of PRGs from one-way functions (OWF)

❖ Not practically efficient

❑ Heuristically-secure instantiations of PRGs

❖ Practically efficient

❖ Not provably-secure --- no efficient distinguishers available

So, the question that we want to answer is do PRG's exist. So, recall that in the last lecture, we had seen that if you have a pseudo random generator, then using that you can design stream ciphers, which allow you to encrypt long messages using short key. So, the question that we not want to answer is do such PRG's exist or not. And to recall, a PRG is a deterministic algorithm which takes a seed or an input of size *l* bits which we denote by s.

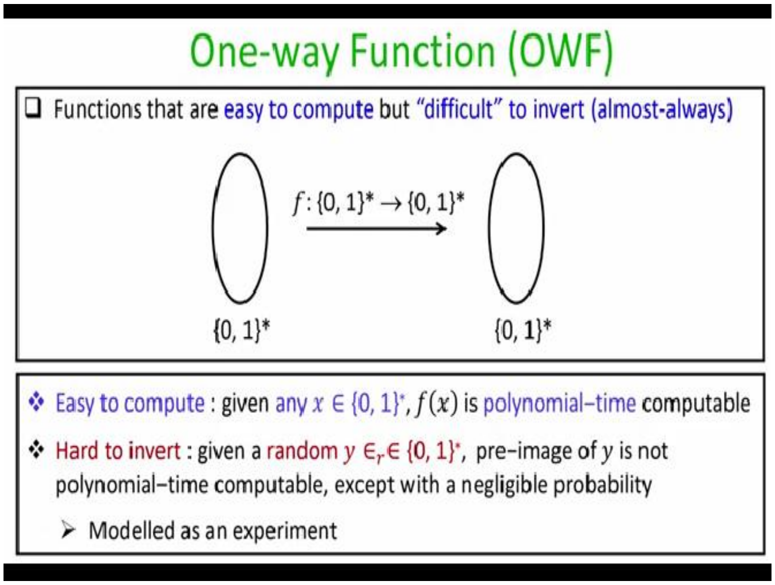
And it expands its output and it produces an output which is significantly larger than the input and the security of PRG means that the output of this deterministic algorithm G should be computationally indistinguishable from the output of a truly random generator of the same size, right. So, it turns out that there are 2 ways to design PRGs. The first way is provably secure constructions of PRGs from one way function.

And when I say provably secure I mean that the constructions that we give using one way function for that we give a proof that indeed the construction is PRG in the sense that they are exist no polynomial time distinguisher which can distinguish outcome of the resultant PRG from the corresponding true random number generator. Unfortunately, we cannot use them in practice because the amount of computation that are involved in the resultant construction are of order of several magnitude.

On the other hand, there are practical instantiations of pseudo random generators, but unfortunately they are not provably secure in the sense we believe that they are heuristically secure because ever since their construction, no weakness have been reported for such pseudo random generators, but the advantage of search heuristically secure pseudo random generators are they are super fast.

So, that is why in practice we prefer to use such an instantiation. So, the focus for this lecture will be the provably secure construction instantiations of pseudo random generators from one way function. In the next picture we will discuss about the practical instantiations of pseudo random generator.

(Refer Slide Time: 03:00)



One-way Function (OWF)

❑ Functions that are easy to compute but "difficult" to invert (almost-always)

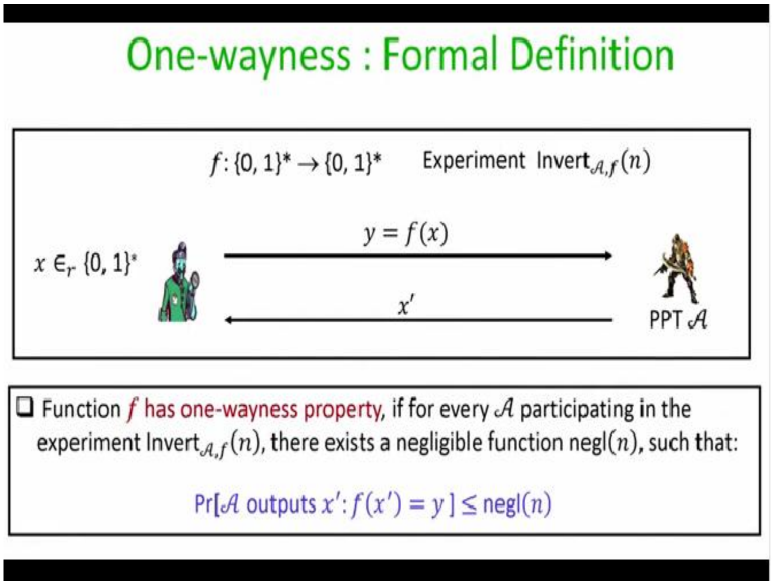
$f: \{0,1\}^* \rightarrow \{0,1\}^*$
$\{0,1\}^*$
$\{0,1\}^*$

- Easy to compute : given any $x \in \{0,1\}^*$, $f(x)$ is polynomial–time computable
- Hard to invert : given a random $y \in_r \in \{0,1\}^*$, pre–image of y is not polynomial–time computable, except with a negligible probability
  - Modelled as an experiment

So let me introduce the concept of one way function or OWF in short, so it is a function from the domain of bit string to the codomain of bit string denoted by f. And, informally, it is a function which is easy to compute, but difficult to invert. More specifically, there are 2 requirements of this function. The first requirement is that it should be easy to compute. Namely, if you are given any input x, computing the value of the function on that input x should be polynomial time, polynomial in the size of the input x.

The second requirement, which is the interesting property from this one way function is the hard to invert property namely, the requirement is that if you are given a random sample y, from the codomain, then finding the preimage or one of the preimages of this given y in polynomial time should be difficult except with a negligible probability. And this property is modeled by an experiment.

(Refer Slide Time: 03:58)



One-wayness : Formal Definition

$f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ Experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$

$x \in_r \{0, 1\}^*$

$y = f(x)$

x'

PPT $\mathcal{A}$

❑ Function f has one-wayness property, if for every $\mathcal{A}$ participating in the experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$, there exists a negligible function negl(n), such that:

$$\Pr[\mathcal{A} \text{ outputs } x' : f(x') = y\,] \leq \mathsf{negl}(n)$$

So in this experiment, we are given the description of a publicly known function f. And the name of the experiment is Invert, played with respect to a polynomial time algorithm whose goal is to invert the output of a function on a random sample with respect to the function f, and n is the security parameter here. So in this experiment, we have 2 entities, namely the verifier and adversary. So the rules of the experiment are as follows.

The verifier here picks a random x from the domain which is an arbitrary bit string and computes the value of the function on that input x, which is given as a challenge for the adversary. So the adversary is not aware of the input x, it only sees the output of the function on the input x, namely y and the goal or the challenge for the adversary is to invert to sample y namely to find at least one of the preimages of this given value y.

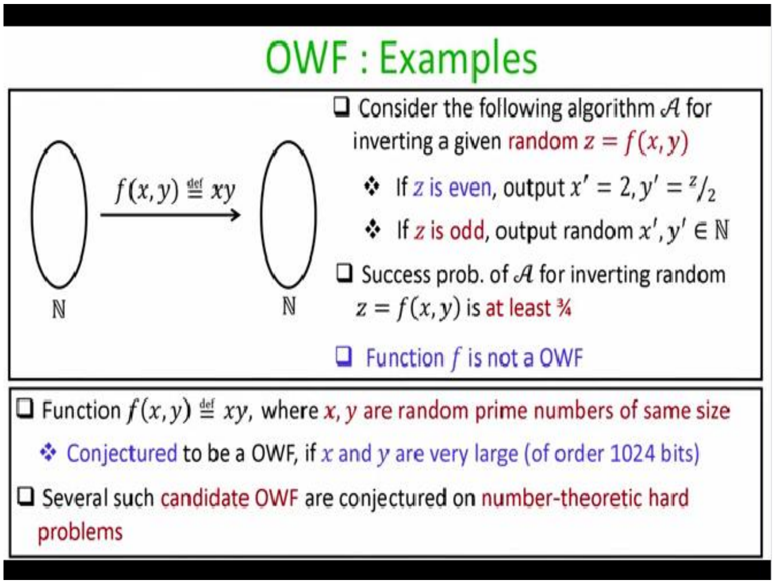
So it submits his response namely x’ and we say that the function f has one wayness property, if for any polynomial time algorithm A participating in this experiment there exist a negligible function say negl(n) such that the probability is that the probability that adversary is indeed able to output x’, which is actually a preimage of the given y is upper bounded by a negligible function.

That means we require that no polynomial time algorithm should be able to invert this random y except with a negligible probability. So, in this definition the requirement is that the success probability of that adversary to invert the random y should be upper bounded by a negligible function not by 0, because they are always exist a guessing strategy by the adversary namely adversary can simply guess a random x’.

And there always exists a nonzero probability that the guessed x’ indeed turns out to be one of the preimages of the given y, that is why in the definition, we do not require that we do not enforce that the condition that it A should be able to invert the sample y should be upper bounded by 0, we give the leverage and the upper bound the success probability of the adversary by a negligible probability.

Also notice that it is not required by the adversary to find the exact x, because the function can be a many to one function. So, it could be possible that the given y which is thrown as a challenge to the adversary has several preimages. The goal of the adversary is to find one of those preimages right.

(Refer Slide Time: 06:25)



$f(x,y) \overset{\text{def}}{=} xy$
$\mathbb{N}$
$\mathbb{N}$

OWF : Examples

- ❑ Consider the following algorithm $\mathcal{A}$ for inverting a given random $z = f(x,y)$
  - ❖ If z is even, output $x' = 2, y' = z/2$
  - ❖ If z is odd, output random $x', y' \in \mathbb{N}$
- ❑ Success prob. of $\mathcal{A}$ for inverting random $z = f(x,y)$ is at least ¾
- ❑ Function f is not a OWF

- ❑ Function $f(x,y) \overset{\text{def}}{=} xy$, where x, y are random prime numbers of same size
  - ❖ Conjectured to be a OWF, if x and y are very large (of order 1024 bits)
- ❑ Several such candidate OWF are conjectured on number-theoretic hard problems

So let us see an example to make our understanding clear, consider the function f which is a 2 input function and the function is from the set of natural numbers to the set of natural numbers. So, the function takes 2 inputs from the set of natural numbers and output of the function is basically the product of the 2 inputs. Now, the question is this function a one way function that means if someone gives you the value of x dot y, your x and y are not known to you. Will you be able to invert or find one of the preimages namely a pair of x, y such that whose product is equal to the given product in polynomial amount of time, and it turns out that they are exist, indeed, there indeed exists a simple algorithm to find one of the preimages and the algorithm is as follows. So you are given a random sample z, which is basically the output of the function on a pair of unknown inputs x, y. And your inversion algorithm is as follows.

The algorithm checks whether z is even or not, if z is even then the algorithm outputs the following pair of preimages, the first preimage x' is 2 and a second preimage y' is z/2. So, that is a pair of inputs which would actually give you the output z, whereas if the sample z which is given to you is odd then you just randomly output x', y' on the set of natural numbers.

So, it turns out that the success probability of this inversion algorithm is actually 3 by 4. Because since the function is a 2 input function, the possible values of x, y could be (odd, odd), (odd, even), (even, odd) and (even, even). And it turns out that out of this 4 combinations, for 3 of the combinations, your z will always be even. And if z is even and indeed the inversion algorithm that is used here, namely outputting 2 and z/2 as the possible preimage is a successful algorithm.

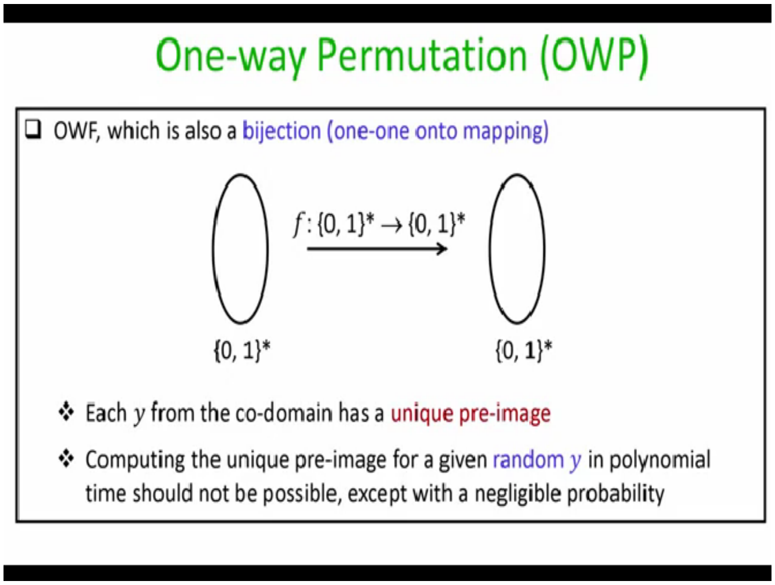
The only problematic case is when the sample x, y is actually an odd, odd pair in which your z may not be z will not be an even number, in which case you might be outputting an incorrect x', y'. So, it turns out that in three fourths of the instances, the adversary or this algorithm will be able to successfully invert the given value z. Hence, this function f is not a one way function because the success probability of inversion is 3/4, which is a significant amount of probability right.

On the other hand, let us slightly modify this function. So, my function f(x, y) is still the product of x and y but instead of saying that x and y are random natural numbers, put the restriction here that x and y are random prime numbers of roughly same size. And it turns out that this function is now conjectured to be a one way function if x and y are large prime numbers say of order 1024 bits.

So I stress here that it is just conjectured that this function f(x, y) where x and y are arbitrary large prime numbers is a one way function because till now, we do not have any polynomial time algorithm to invert this function. It is not proved formally that indeed this function is a one way function. And later on, when we will be discussing about the public key cryptography, and we will be discussing number theory where we will see several such candidate one way functions, which are conjecture on several hard so called number theoretic hard problems.

So for the moment, we will believe that this function f(x, y) where x and y are arbitrary large numbers and the function is a product of x and y is a candidate one way function.

(Refer Slide Time: 10:02)



One-way Permutation (OWP)

❑ OWF, which is also a bijection (one-one onto mapping)

$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$

$\{0, 1\}^*$ $\{0, 1\}^*$

❖ Each y from the co-domain has a unique pre-image

❖ Computing the unique pre-image for a given random y in polynomial time should not be possible, except with a negligible probability

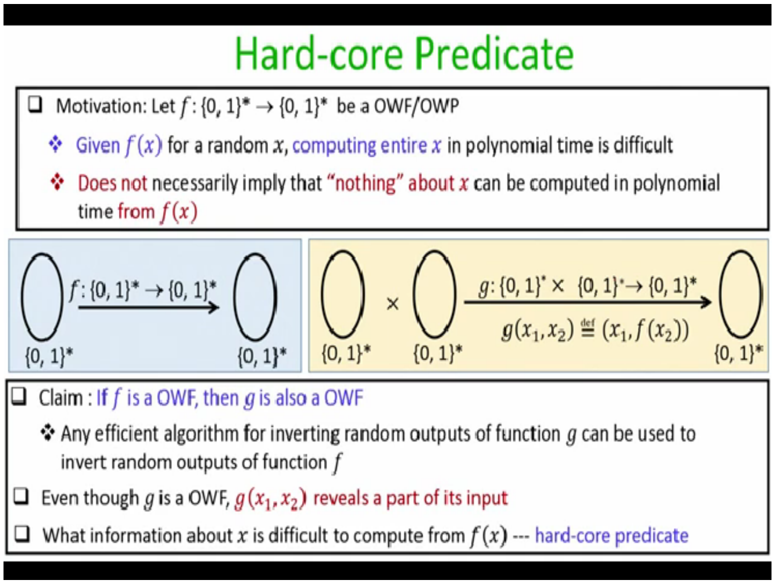
So now let us define another related concept namely that of one way permutation or OWP. And basically a one way permutation is a one way function, which also happens to be a bijection. Namely, the mapping is now a one-one onto mapping. That means each y from the codomain will have a unique preimage. And the requirement or the hard to invert part for one way permutation is that if you are given a random y from the codomain, the challenge for the adversary is to find out a unique preimage x which would have actually given that y in polynomial amount of time.

So the difference here is for the case of one way function, the random challenge y could have several preimages and your goal was to find one of those preimages. In the case of one way permutation, random sample y which is given to you will have only a unique preimage. And your goal is to find that unique preimage and polynomial amount of time. Notice that both in the case of one way function as well as in one way permutation, we put the restriction that the inversion algorithm should take polynomial amount of time.

Because if we do not put any restriction on the amount of time, which is given to the inversion algorithm, then there are always exists a brute force algorithm namely, the algorithm can try over all possible candidate x from the set of candidate x and definitely it will hit upon one x which will be give the random y which is thrown as the challenge, but the running time of this brute force algorithm will be exponentially or it will be proportional to the size of the domain.

And the size of the domain is exponentially large than the inversion algorithms running time is also exponential. So, that is why in the experiment, where we model the inversion part both for the one way function as well as one way permutation we put the restriction that running time of that adversary should be polynomial time.

(Refer Slide Time: 11:52)



Hard-core Predicate

❑ Motivation: Let $f: \{0,1\}^* \to \{0,1\}^*$ be a OWF/OWP

❖ Given $f(x)$ for a random x, computing entire x in polynomial time is difficult

❖ Does not necessarily imply that "nothing" about x can be computed in polynomial time from $f(x)$

$f: \{0,1\}^* \to \{0,1\}^*$
{0, 1}*
{0, 1}*
{0, 1}*
×
{0, 1}*
$g: \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$
$g(x_1, x_2) \overset{\text{def}}{=} (x_1, f(x_2))$
{0, 1}*

❑ Claim : If f is a OWF, then g is also a OWF

❖ Any efficient algorithm for inverting random outputs of function g can be used to invert random outputs of function f

❑ Even though g is a OWF, $g(x_1, x_2)$ reveals a part of its input

❑ What information about x is difficult to compute from $f(x)$ --- hard-core predicate

So, now let us discuss another interesting related concept namely that of hard-core predicate which will be useful when we will design our candidate pseudo random generator from one way function and the motivation for this hard-core predicate is as follows. So imagine you are given a function f, which is a one way function or a one way permutation. And if you are given the value of function on a random input x, then definitely computing the entire x in polynomial amount of time is difficult.

Because that is what is the property of a one way function or one way permutation, but that does not necessarily mean that you cannot compute anything about x in polynomial time, there might be some information about x, which you can compute from the value of f(x) in polynomial amount of time. So basically hard-core predicate models what can be computed about x from a f(x) in polynomial amount of time and what cannot be computed.

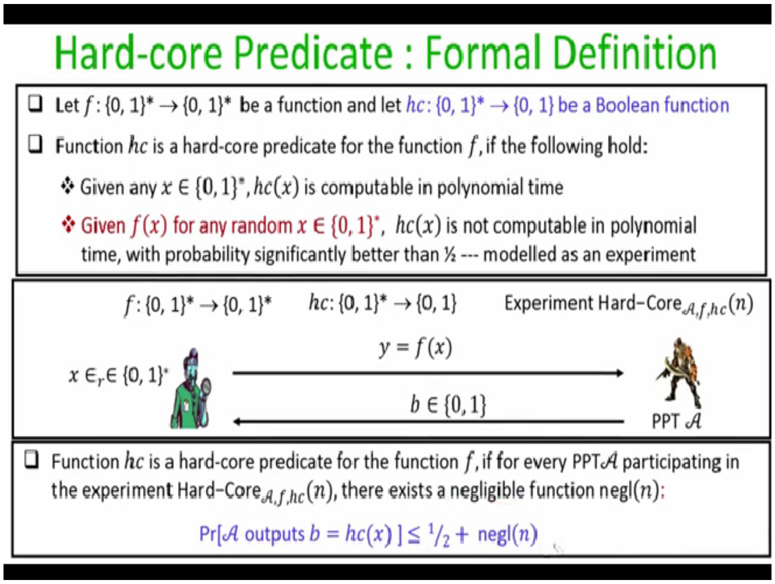
To make my point more clear, consider the function $f: \{0,1\} * \to \{0,1\} *$ and say this function f is a one way function and using this function we design another function g, which is now a 2 input function and the function g on the input pair x_1, x_2 is defined as the output consisting of x_1 as it is, and the second part of the output of this function g is actually the value of the function f on the second part of the input, namely x_2 , $g(x_1, x_2) := (x_1, f(x_2))$.

That is the way I am defining my function g. And my claim is that if the size of x_1 and x_2 are roughly same, namely some polynomial function in the security parameter, then if the function f with which we start is a one way function, then the constructed function g, which is constructed like this is also a one way function right. So basically, the idea here is that if g is not a one way function, that means if someone gives you the value of $g(x_1, x_2)$, and you can actually find out x_1, x_2 in polynomial amount of time. Then intuitively using that algorithm, you can also compute the value of x_2 using just given $f(x_2)$, so we can formally established his fact using a reduction argument. So I am not going into the reduction argument. But basically the idea here is that if f is one way function and the function g, which is a 2 input function constructed like this is also a one way function provided both x_1 and x_2 are roughly same size.

Now, if you see the function g, it turns out that even though it is a one way function, by seeing the output of this function g on a random pair of input, you actually end up learning the one half of the input, right. Because the first half of the output is actually the first half of the input as it is. In that sense, even though this function g is a one way function, it is not the case that if I give you the value of this function g on an arbitrary pair of input, you cannot compute back the entire input in polynomial amount of time.

There is something which you can compute, there is something which you cannot compute in polynomial amount of time. So this notion of hard-core predicate precisely models, what is polynomial time computable from the value of the function f(x) on random input x, right.

(Refer Slide Time: 15:01)



Hard-core Predicate : Formal Definition

- ❑ Let $f: \{0,1\}^* \rightarrow \{0,1\}^*$ be a function and let $hc: \{0,1\}^* \rightarrow \{0,1\}$ be a Boolean function
- ❑ Function hc is a hard-core predicate for the function f, if the following hold:
  - ❖ Given any $x \in \{0,1\}^*$, $hc(x)$ is computable in polynomial time
  - ❖ Given $f(x)$ for any random $x \in \{0,1\}^*$, $hc(x)$ is not computable in polynomial time, with probability significantly better than ½ --- modelled as an experiment

$f: \{0,1\}^* \rightarrow \{0,1\}^*$ $hc: \{0,1\}^* \rightarrow \{0,1\}$ Experiment $\text{Hard-Core}_{\mathcal{A},f,hc}(n)$

$x \in_r \in \{0,1\}^*$

$y = f(x)$

$b \in \{0,1\}$

PPT $\mathcal{A}$

- ❑ Function hc is a hard-core predicate for the function f, if for every PPT $\mathcal{A}$ participating in the experiment $\text{Hard-Core}_{\mathcal{A},f,hc}(n)$, there exists a negligible function $\text{negl}(n)$:

$$\Pr[\mathcal{A} \text{ outputs } b = hc(x)] \leq {}^{1}/_{2} + \text{negl}(n)$$

So let us see the formal definition of this hard-core predicate. So, imagine you are given a function from a domain of bit string to the codomain of bit string and I stress that the function may not be a one way function it could be a normal function. So, we are given a function f and corresponding to the function f we define another function *hc* which is a Boolean function that means, it takes an arbitrary binary string as an input and it produces a Boolean output 0, 1.

And then we see that this function *hc* associated with a function f is a hard-core predicate if the following 2 conditions hold. So, the first condition is that if you are given random input text and computing the value of this function, hard-core predicate namely *hc* of f should be polynomial time computable that means you should be able to compute the value of *hc* of x in polynomial time polynomial in the size of the input x.

The second interesting property which actually makes hard-core predicate very interesting is the following that if you are given the value of f(x) for a randomly chosen x, then it is not possible to compute the value of *hc*(x) with probability significantly better than half in polynomial amount of time. That means the challenge here is that if I give you f(x) and I do not to tell you the value of x, where x is randomly chosen, then even after learning f(x), we should not be able to compute the value of the function *hc* of x in polynomial amount of time, except with probability, negligible better than half. If that is the case, then we say that the function *hc*(x), *hc* is actually the hard-core predicate associated with the function f. So this requirement of inability to compute the value of *hc*(x) with probability significantly better than half by a polynomial time algorithm is modeled by an experiment which we call as hard-core experiment.

And now this experiment is designed with respect to a function f which is publicly known, which is a candidate function which could be a candidate one way function or one way permutation or it could just be a function and associated with the function f we have a candidate hard-core predicate, which is a Boolean function and experiment involves 2 entities. So the rule of the experiment is as follows.

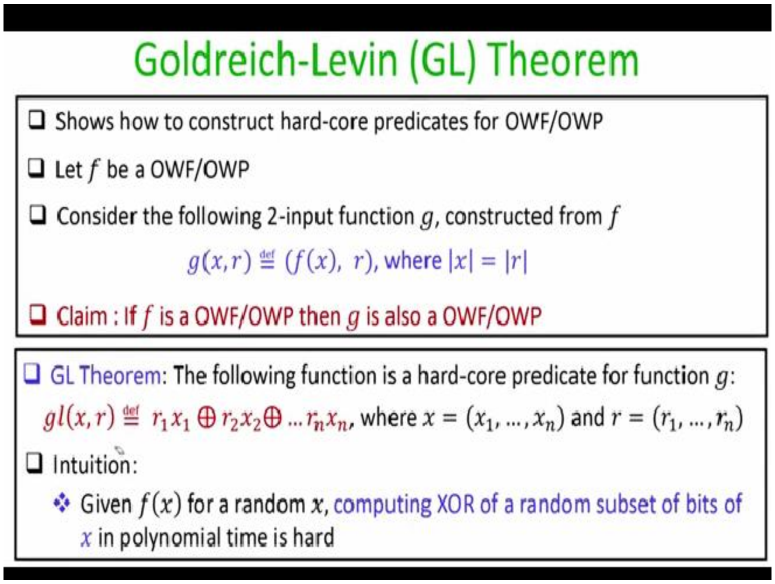
The verifier here picks a random x from the domain, computes the value of the function on that input x. And the corresponding output y is given as a challenge to the adversary. And the goal of the adversary or the challenge for the adversary is after seeing the value of y it has to compute the value of *hc*(x) without knowing the x. So, say *hc*(x) is going to be a bit it submits the adversary outputs a bit which could be either 0 or 1.

And we say that adversary has won the game if actually b = *hc*(x). So, that means, we say that the function *hc* associated with the function f is a hard-core predicate, if for any polynomial time algorithm participating in this experiment, the probability that adversary outputs b where b is actually equal to *hc* of x is upper bounded by half plus some negligible probability or negligible function in the security parameter.

If that is the case, then we say that the function hc is a hard-core predicate associated with the function f. So, notice there is in definition, we bounded success probability of the adversary to be half plus negligible, we do not require that the success probability of the adversary should be 0, because there is always a guessing strategy or guessing adversary who can just guess a bit and there always exist a probability of 1/2 that the guest is indeed the value of *hc*(x) correctly.

Apart from that we are also willing to give that adversary an extra negligible advantage and successfully outputting the hard-core predicate, because we are in a computationally secure world right. So that is the definition of hard-core predicate.

(Refer Slide Time: 18:53)



Goldreich-Levin (GL) Theorem

❑ Shows how to construct hard-core predicates for OWF/OWP

❑ Let f be a OWF/OWP

❑ Consider the following 2-input function g, constructed from f

$$g(x,r) \overset{\text{def}}{=} (f(x),\ r), \text{ where } |x| = |r|$$

❑ Claim : If f is a OWF/OWP then g is also a OWF/OWP

❑ GL Theorem: The following function is a hard-core predicate for function g:

$gl(x,r) \overset{\text{def}}{=} r_1x_1 \oplus r_2x_2 \oplus \dots r_nx_n$, where $x = (x_1, \dots, x_n)$ and $r = (r_1, \dots, r_n)$

❑ Intuition:

❖ Given $f(x)$ for a random x, computing XOR of a random subset of bits of x in polynomial time is hard

So now the question is, if we are given a candidate one way function or one way permutation does there exist an associated hard-core predicate or not. And it turns out that interestingly the answer is yes. And that is because of a very fundamental theorem and cryptography, which is also called as Goldreich-Levin theorem. And Goldreich-Levin theorem basically shows you how to construct a hard-core predicate associated with any given one way function, or any given one way permutation.

So we will not be going into the proof of the Goldreich-Levin theorem, we will just roughly see the theorem statement and intuition of the security. So the theorem statement is as follows. So imagine you are given a candidate one way function say f or a candidate one way permutation. Now using this candidate function f, we construct a 2 input function g, and the construction of g is as follows. So it passes the input as x, r.

And the output of g is defined as follows. So we evaluate the function f on the first part of the input of g. That constitutes the first part of the output of g, and the second part of the input of g is copied as it is in the output of the g. And here the size of the 2 inputs of the g, namely x and r are of same order. So that is the way we define our function g. Now we can prove formally that if the function f with which we start with is a one way function or one way permutation then the function g which we have constructed like (f(x), r) is also a one way function or one way permutation respectively.

And the proof is by a reduction argument. Namely, we can show that if you have a polynomial time algorithm to invert the output of the function g that you have constructed on a random pair of input x, r. That means if you have an algorithm who without knowing the random pair of input x, r, but just seeing the value of g(x, r) can give you back x, r in polynomial amount of time. Then using the same inversion algorithm we can construct another polynomial time algorithm, which when given the value of f(x) on a random x without knowing the value of x can actually give you back the value of x, which is a contradiction to the fact that f is a one way function or a one way permutation.

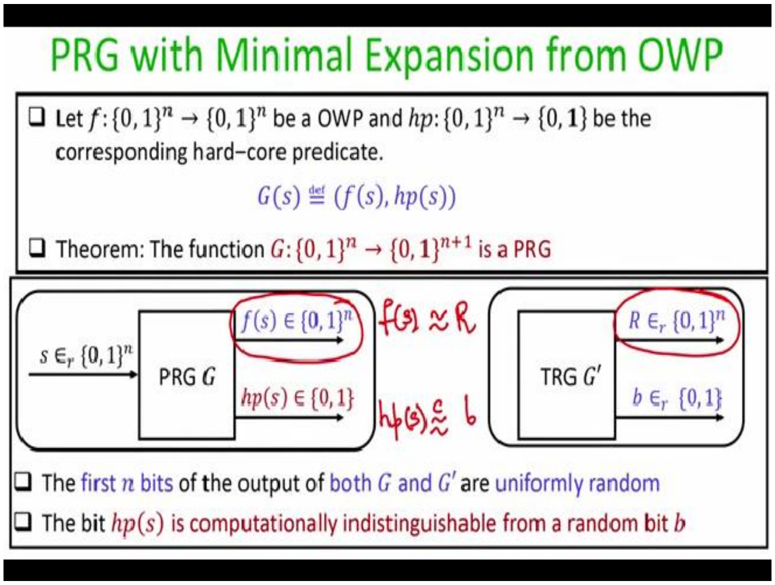
So that is a straightforward reduction argument using which we can prove this theorem. So, now we will focus on the function g. And what Goldreich-Levin theorem shows you basically that if this function g is a one way function, then we can associate a corresponding hard-core predicte with this function g. Namely, the hard-core predicate is the Boolean function, which basically takes random linear combination of the bits of the input x, right.

So what we consider here is that your r is interpreted as a bit string of length n, and your x is also considered as a bit string of length n, these are the 2 inputs for the function g, and associated hard-core predicate, which is associated with the function g is denoted by the function *gl*. And the function *gl* is basically random linear combination of the bits of the x, where the combiners are defined by the bit representation of the input r.

And what the Goldreich-Levin theorem claims basically that this function *gl*(x, r) is the hard-core predicate associated with the function g. That means, if someone gives you the value of the function g(x, r), for a random x and r, then, except with probability half plus negligible no polynomial time algorithm can compute the value of *gl*(x, r). And the intuition behind this the proof of the Goldreich-Levin theorem is that even though you are given f(x) since you do not know the value of x, basically the challenge for computing the Goldreich *gl*, the output of the function *gl*(x, r) is basically to compute the XOR of the random subsets of the bit of x but since the adversary will not know the exact bits of the x, it will be difficult for the adversary to compute that XOR of the random subsets of the bits of x. That is basically the intuition of this theorem. However, it turns out that the proof of this theorem is indeed challenging and involved.

So, the students who are interested to go through the proof of this Goldreich-Levin theorem they can refer to the book by Katz and Lindel, where they have given the detailed proof, but for our discussion, we will assume that the function *gl*(x, r) is associated hard-core predicate which is associated with our candidate one way function g(x, r) right.

(Refer Slide Time: 23:36)



PRG with Minimal Expansion from OWP

❑ Let $f:\{0,1\}^n \rightarrow \{0,1\}^n$ be a OWP and $hp:\{0,1\}^n \rightarrow \{0,\mathbf{1}\}$ be the corresponding hard-core predicate.

$$G(s) \overset{\text{def}}{=} (f(s), hp(s))$$

❑ Theorem: The function $G:\{0,1\}^n \rightarrow \{0,\mathbf{1}\}^{n+1}$ is a PRG

❑ The first n bits of the output of both G and G' are uniformly random

❑ The bit $hp(s)$ is computationally indistinguishable from a random bit b

So, assuming we have now a hard-core predicate, and we have a candidate one way permutation let us see how we can construct PRG with minimal expansion, and what we mean by minimal expansion is that we will assume that we are given a candidate one way permutations from the set of n bit strings to the set of n bit strings. That means the function f takes an input which is a string of length n and it produces output which is also a string of length n.

And it is a one way permutation. And associated with the function f, we have a hard-core predicate. Now given this we will construct a pseudo random generator where the construction of the pseudo random generator is as follows. So we call the algorithm as G which takes an input s, which is a random input of size n bits, and it is going to produce an output of n + 1 bits. So the first n bits of the output is nothing but the value of the function f for the input s.

And the last output bit of the function G is basically the value of your hard-core predicate on the input s. And the theorem statement that we are now going to prove is that if the function f is a one way permutation, and *hp* is the associated hard-core predicate, then the construction G that we have given here is actually a pseudo random generator, which expands its input by one bit.

That is why it is actually a PRG with minimal expansion because that is a minimum amount expansion which you expect from a PRG. Because if the output of PRG is same as input of PRG, then that is not a PRG as per our definition of PRG, because the first requirement from the PRG definition is that the size of the output should be more than the size of the input.

So minimal such condition is when the size of the output is one more than the size of the input. And that is why this PRG is a PRG with minimal expansion. So pictorially, this is your algorithm G, you take the input of size n, which is a uniformly random bit string, and you produce a bit string of length n + 1. And we want to prove that this algorithm G is a pseudo random generator.

Namely, we want to prove that the output of this algorithm G is indistinguishable from the output of a true random number generator say G', which also outputs strings of length n + 1 where the the algorithm G' differs from the algorithm G in the sense that all the n + 1 output bits of G' are uniformly random and independent of each other, whereas the outputs of algorithm G actually depend upon the input s.

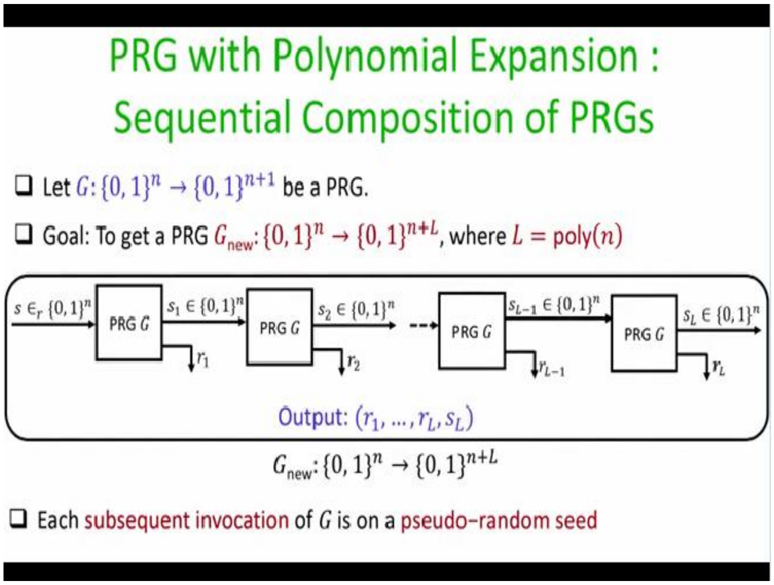
And intuitively if we see the difference between the output of PRG and output of the algorithm G' is actually in the last bit, because if we compare the first n output bits of the algorithm G, and the first n output bits of the algorithm G', the distribution is actually the same, namely the uniform distribution over the set of n bit strings. This is because if the seed s to the algorithm G uniformly random, then the function output f(s) is also going to be uniformly random that means, this output f(s) is actually indistinguishable from the output R. The difference is only in the last part of the output of G, and the last bit of the output of G' for the case of G' the last output bit is a uniformly random bit, but for the algorithm G the last bit is actually the value of the hard-core predicate on the input s.

And it turns out that from the definition of hard-core predicate the output bit *hp*(s) is actually computationally indistinguishable from the uniformly random bit that means no polynomial time algorithm can distinguish between the uniformly random bit b and the bit *hp*(s), because that is what is the definition of hard-core predicate. And that is precisely the argument which we are going to use to prove that the construction G that we are given is actually a pseudo random generator.

We can actually formally establish that if the algorithm G which you have constructed is not a pseudo random number generator, then there exists an algorithm polynomial time algorithm, which when given the value of f(s) without knowing the value of s can actually compute the value of *hp*(s) with probability significantly better than half and that will be a contradiction to the assumption that the function *hp* is the associated hard-core predicate corresponding to the function f.

So that formal argument we can establish though a reduction, I am leaving that reduction and the formal details to you as an assignment.

(Refer Slide Time: 28:07)



PRG with Polynomial Expansion : Sequential Composition of PRGs

❑ Let $G: \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ be a PRG.

❑ Goal: To get a PRG $G_{\text{new}}: \{0,1\}^n \rightarrow \{0,1\}^{n+L}$, where $L = \text{poly}(n)$

$G_{\text{new}}: \{0,1\}^n \rightarrow \{0,1\}^{n+L}$

❑ Each subsequent invocation of G is on a pseudo-random seed

So, that is a way we can design a PRG which whose output is one more than the size of input, but in practice we might be interested to construct pseudo random generator where the size of the output is significantly large compared to the size of the input. So, now, what we are going to do is using the previous construction, where the output of the PRG is one more than the size of the input, we are going to construct another pseudo random generator whose output is polynomially large compared to the size of input and this will be done by doing a sequential composition of pseudo random generators where we will be executing several instance of the pseudo random generator that we have constructed in the last slide, in polynomial number of times in sequence one after the other.

And this will be different from the polynomial composition of pseudo random generators which we had seen in one of our earlier lectures. So, now, in this construction we are assuming that we are given a pseudo random generator, which takes a n bit seed uniformly random n bit input and gives the output of length n + 1. And for example, you can take the construction G to be the same that we have constructed in our last slide using the one way function and hard-core predicate.

Now, using this function G our goal is to construct a new PRG which takes an input of size n and produces an output of length (n + L) bits, where L is some polynomial function in your security parameter. And this is how we will construct a new pseudo random generator G_{new} by sequentially composing the function G polynomial number of time. So, what we do here is we invoke an instance of the algorithm G on the input s.

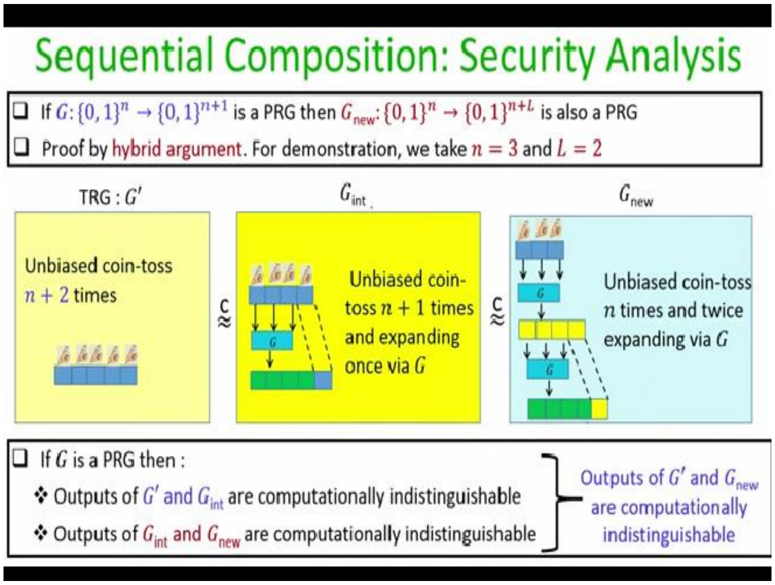
And the input s will be the seed for our algorithm G_{new} which we are constructing. So, the input s for the G_{new} is actually passed as an input for the algorithm G and as a result, G produces an output of size n + 1 bits. So we parse that input as r_1, s_1 where r_1 is a single bit and the remaining n bits is denoted by s_1. Now what we do is we actually invoke another invocation of the PRG with a seed value s_1.

So seed value s_1 is a string of length n bits. So that fits the semantic of my algorithm G. And I invoke the second invocation of the algorithm G using s_1 as seed, which produces an output of size n + 1 bit, which again is parsed as a bit r_2 concatenated with the remaining part, which is a string of length n bits and now again, I keep on doing this process repeatedly, one after the other L number of times, and the final invocation of the PRG will be on the output produced by the second last output of the PRG namely s_{L-1}.

And it produces an output of length L + 1 bits which again we pass as r_L, s_L. So, the overall output of this algorithm G_{new} will be nothing but the concatenation of r_L, r_2, r_{L-1} , r_L along with s_L. So, what basically we are doing here is we are actually invoking the algorithm G, the existing algorithm G, L number of times and in each invocation we are actually generating one pseudo random output and a pseudo random seed, where the pseudo random seed is actually used for the next invocation of the PRG and this process is repeated L number of times. So, the claim here is that if the existing algorithm G is a pseudo random generator, then the new algorithm or the new construction that we have designed G_{new} is also a pseudo random generator provided L is some polynomial function in the security parameter right.

And now, you can see that why this construction is called as a sequential composition, here the invocations of G are not independent of each other, they actually depend on the previous output, means the next invocation of the G actually depends upon the output of the previous invocation of G. In that sense, it is a sequential composition of the pseudo random generators.

(Refer Slide Time: 32:09)



Sequential Composition: Security Analysis

❑ If $G: \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ is a PRG then $G_{\text{new}}: \{0,1\}^n \rightarrow \{0,1\}^{n+L}$ is also a PRG

❑ Proof by hybrid argument. For demonstration, we take $n = 3$ and $L = 2$

TRG : G'

Unbiased coin-toss $n + 2$ times

$\overset{c}{\approx}$

G_{int}

Unbiased coin-toss $n + 1$ times and expanding once via G

$\overset{c}{\approx}$

G_{new}

Unbiased coin-toss n times and twice expanding via G

❑ If G is a PRG then :

❖ Outputs of G' and G_{int} are computationally indistinguishable

❖ Outputs of G_{int} and G_{new} are computationally indistinguishable

Outputs of G' and G_{new} are computationally indistinguishable

So, the theorem statement that we want to prove is the following. And the proof will be using the hybrid argument because if you see pictorially they are a polynomial number of invocations of G which are involved here. And we cannot directly reduce the security of the algorithm G new to the security of the algorithm G. Because there are polynomial number of invocations.

So to get around this we actually introduce hybrid argument and for the sake of demonstration we assume that n = 3 and L = 2. Namely, I assume that the G_{new} construction that we are giving takes a seed of length 3 bits and it produces an output of 5 bits. So what we are going to do in the hybrid argument is as follows. We are going to introduce polynomial number of intermediate hybrids.

So we start with the first algorithm G' and output of the algorithm G' basically is a collection of n + 2 uniformly random bits. So since n = 3, and L = 2 here, basically, the algorithm outputs a string of length 5, where all the bits are uniformly random. And that is precisely the output that a true random number generator of length 5 would actually look like. That is the construction of your algorithm G'.

Whereas the construction that we had given, namely the sequential construction will look something as this. We start with a seed of length 3 bits, right. That is your seed. And we invoke the algorithm G and produce a pseudo random output of size 4 bits. Now, what we do is we take the first output bit to be the overall output of the algorithm G_{new} and the remaining 3 bits are actually used as the seed for the next invocation of G.

So actually, in this pictorial representation I am doing it other way around, I am setting the last bit of the first invocation of G to be the overall output, or the overall last bit of the algorithm G_{new}. And the first 3 output bits of the first invocation of G is used as the seed for the next invocation of G. And it does not make any difference here, you can use any way, any representation for your convenience here.

So the second invocation of G is actually now on a pseudo random string of length 3 bits, and it produces an output of length 4 bit, which is actually concatenated with the first bit, the last bit which was generated by the previous invocation of G. And that is how you would have actually obtained a pseudo random string of length 5 bits. That is the construction of your algorithm G_{new}.

And our goal is to prove that no polynomial time distinguisher can distinguish apart a random sample which is produced by G' from a random sample produced by G_{new}, that means if it is thrown a uniformly random sample, which is either generated by G' or by G_{new} without actually knowing the way by which it is generated, then the distinguisher cannot find out in polynomial amount of time whether the sample that is given to him is actually generated as per G' or as per G_{new}.

To prove that what we are going to do here is we are going to introduce an intermediate hybrid or an intermediate construction, which we call us G_{int}. And what this construction G_{int} does is it takes an input of size 4 bits, which are uniformly random. And what it does is on the first 3 bits of the input it invokes and invocation of G and it produces a pseudo random output of length 4 bits.

And the last bit of the input of G_{int} is copied as it is in the overall output. And that is the way the algorithm G_{int} produces a string of length 5 bits. So you can see that you have now got 3 different constructions. The construction of a true random number generator which outputs a string of length 5 where all the 5 bits are uniformly random. The construction G_{new} which is construction that we have proposed which produces a pseudo random string of length 5 bits by invoking by using 2 invocations of G.

And we have an intermediate algorithm G_{int}, which also produces a pseudo random string of length 5 bits, but by just using one invocation of algorithm. Now, in the hybrid argument, what we are going to prove is we can prove that if the construction G, which expands a 3 bit uniformly random string to a 4 bit pseudo random string is a PRG, then the construction G' and the construction G_{int} are indistinguishable.

That means no polynomial time distinguisher can distinguish apart a sample produced by G' from a sample produced by G_{int}. That we can establish by reducing the distinguisher who can distinguish between G' and G_{int} to a distinguisher who can actually distinguish between an output of the algorithm G from a truly random number generator which produces output of length 4 bits right.

So I am not going into the details of the proof, I hope that you will be able to complete that proof. In the same way we can prove that if the algorithm G is actually a pseudo random generator, then no polynomial time distinguisher can distinguish apart a random sample produced by G_{int} from a random sample produced by G_{new}. And again, the proof will be by reduction right.

That means if you have a polynomial time distinguisher who can distinguish apart a sample of G_{int} from a sample of G_{new} then we can show via reduction that the same distinguisher can be used to distinguish apart the output of the algorithm G from the output of a true random number generator which produces true random strings of length n + 1 bits, which is a contradiction.

So now since we have 2 intermediate hybrids and the algorithm G' is indistinguishable from the algorithm G_{int} and algorithm G_{int} is indistinguishable from algorithm G_{new} it follows that overall the algorithm G' is also indistinguishable from the algorithm G_{new}. So that is the way we can prove by hybrid argument that G' and G_{new} are computationally indistinguishable.

For the general case where L is some polynomial function of the security parameter, we have to introduce L number of hybrids. And what we can do is we can prove that each pair of consecutive hybrids are computationally indistinguishable by reducing its security to the security of your underlying construction G. And since there are a polynomial number of hybrids, we end up establishing that algorithm G' is computationally indistinguishable from the algorithm G_{new}.

So, I leave the details of the formal proof to you as an exercise. So, that brings me to the end of this lecture. So just to summarize, in this lecture, we have seen provably secure constructions of pseudo random generators from one way functions. So we have proved that if one way function exists, then corresponding to that one way function, we can construct hard-core predicate and using that hard-core predicate, and one way function we can actually construct pseudo random generator which actually expands input by 1 bit. And then using this construction, namely the pseudo random generator with 1 bit expansion polynomial amount of time in sequence, we can actually produce a pseudo random generator, whose output is polynomially large compared to its input, and the construction is provably secure.

Namely, we can prove that no polynomial time distinguisher exists for our construction, otherwise, it reduces to the security of the underlying one way function and the hard-core predicate. I hope you enjoyed this lecture. Thank you.