Pseudorandomness (IV)

601.642/442: Modern Cryptography

Fall 2020

Pseudorandomness: Next-Bit Test

- Here is another interesting way to talk about pseudorandomness

Pseudorandomness: Next-Bit Test

- Here is another interesting way to talk about pseudorandomness
- A pseudorandom string should pass all efficient tests that a (truly) random string would pass

Pseudorandomness: Next-Bit Test

- Here is another interesting way to talk about pseudorandomness
- A pseudorandom string should pass all efficient tests that a (truly) random string would pass
- **Next Bit Test**: for a truly random sequence of bits, it is not possible to predict the "next bit" in the sequence with probability better than 1/2 even given all previous bits of the sequence so far

Pseudorandomness: Next-Bit Test

- Here is another interesting way to talk about pseudorandomness
- A pseudorandom string should pass all efficient tests that a (truly) random string would pass
- **Next Bit Test**: for a truly random sequence of bits, it is not possible to predict the "next bit" in the sequence with probability better than 1/2 even given all previous bits of the sequence so far
- A sequence of bits *passes the next bit test* if no efficient adversary can predict "the next bit" in the sequence with probability better than 1/2 even given all previous bits of the sequence so far

Next-bit Unpredictability

Definition (Next-bit Unpredictability)

An ensemble of distributions $\{X_n\}$ over $\{0,1\}^{\ell(n)}$ is next-bit unpredictable if, for all $0 \leqslant i < \ell(n)$ and n.u. PPT $\mathcal{A}$, $\exists$ negligible function $\nu(\cdot)$ s.t.:

$$\Pr[t = t_1 \ldots t_{\ell(n)} \leftarrow X_n \colon \mathcal{A}(t_1 \ldots t_i) = t_{i+1}] \leqslant \frac{1}{2} + \nu(n)$$

Next-bit Unpredictability

Definition (Next-bit Unpredictability)

An ensemble of distributions $\{X_n\}$ over $\{0,1\}^{\ell(n)}$ is next-bit unpredictable if, for all $0 \leqslant i < \ell(n)$ and n.u. PPT $\mathcal{A}$, $\exists$ negligible function $\nu(\cdot)$ s.t.:

$$\Pr[t = t_1 \dots t_{\ell(n)} \leftarrow X_n \colon \mathcal{A}(t_1 \dots t_i) = t_{i+1}] \leqslant \frac{1}{2} + \nu(n)$$

Theorem (Completeness of Next-bit Test)

If $\{X_n\}$ is next-bit unpredictable then $\{X_n\}$ is pseudorandom.

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{ x \leftarrow X_n, u \leftarrow U_n : x_1 \ldots x_i u_{i+1} \ldots u_{\ell(n)} \right\}$$

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \leftarrow X_n, u \leftarrow U_n : x_1 \ldots x_i u_{i+1} \ldots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \leftarrow X_n, u \leftarrow U_n : x_1 \ldots x_i u_{i+1} \ldots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \leftarrow X_n, u \leftarrow U_n : x_1 \dots x_i u_{i+1} \dots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n
- Suppose $H_n^{(\ell(n))}$ is next-bit unpredictable but not pseudorandom

Next-bit Unpredictability $\Longrightarrow$ Pseudorandomness

$$H_n^{(i)} := \left\{ x \leftarrow X_n, u \leftarrow U_n : x_1 \dots x_i u_{i+1} \dots u_{\ell(n)} \right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n
- Suppose $H_n^{(\ell(n))}$ is next-bit unpredictable but not pseudorandom
- $H_n^{(0)} \not\approx H_n^{(\ell(n))} \implies \exists\ i \in [\ell(n) - 1]$ s.t. $H_n^{(i)} \not\approx H_n^{(i+1)}$

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \leftarrow X_n, u \leftarrow U_n : x_1 \ldots x_i u_{i+1} \ldots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n
- Suppose $H_n^{(\ell(n))}$ is next-bit unpredictable but not pseudorandom
- $H_n^{(0)} \not\approx H_n^{(\ell(n))} \implies \exists\ i \in [\ell(n) - 1]$ s.t. $H_n^{(i)} \not\approx H_n^{(i+1)}$
- Now, next bit unpredictability is violated

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \leftarrow X_n, u \leftarrow U_n : x_1 \ldots x_i u_{i+1} \ldots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n
- Suppose $H_n^{(\ell(n))}$ is next-bit unpredictable but not pseudorandom
- $H_n^{(0)} \not\approx H_n^{(\ell(n))} \implies \exists\, i \in [\ell(n) - 1]$ s.t. $H_n^{(i)} \not\approx H_n^{(i+1)}$
- Now, next bit unpredictability is violated
- Exercise: Do the full formal proof

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way <u>permutation</u> (OWP) over $\{0, 1\}^n$

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way <u>permutation</u> (OWP) over $\{0,1\}^n$
  - Domain and Range are of same size, i.e., $|f(s)| = |s| = n$

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s)\|h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way permutation (OWP) over $\{0,1\}^n$
  - Domain and Range are of same size, i.e., $|f(s)| = |s| = n$
  - $f(s)$ is uniformly distributed over $\{0,1\}^n$ if s is

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way <u>permutation</u> (OWP) over $\{0,1\}^n$
  - Domain and Range are of same size, i.e., $|f(s)| = |s| = n$
  - $f(s)$ is uniformly distributed over $\{0,1\}^n$ if s is

    $\forall y : \Pr[f(s) = y] = \Pr[s = f^{-1}(y)] = 2^{-n}$

PRG with 1-bit stretch

- Hardcore predicate: It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way <u>permutation</u> (OWP) over $\{0,1\}^n$
  - Domain and Range are of same size, i.e., $|f(s)| = |s| = n$
  - $f(s)$ is uniformly distributed over $\{0,1\}^n$ if s is

    $\forall y : \Pr[f(s) = y] = \Pr[s = f^{-1}(y)] = 2^{-n}$

    $\Rightarrow f(s)$ is uniformly distributed

PRG with 1-bit stretch

- Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be a **OWP**
- Let $h : \{0,1\}^* \rightarrow \{0,1\}$ be a hardcore predicate for f
- **Construction:** $G(s) = f(s) \parallel h(s)$

PRG with 1-bit stretch

- Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be a **OWP**
- Let $h : \{0,1\}^* \rightarrow \{0,1\}$ be a hardcore predicate for f
- **Construction:** $G(s) = f(s) \| h(s)$

Theorem (PRG based on OWP)

G is a pseudorandom generator with 1-bit stretch.

- Think: Proof?

PRG with 1-bit stretch

- Let $f : \{0,1\}^* \to \{0,1\}^*$ be a **OWP**
- Let $h : \{0,1\}^* \to \{0,1\}$ be a hardcore predicate for f
- **Construction:** $G(s) = f(s) \parallel h(s)$

Theorem (PRG based on OWP)

G is a pseudorandom generator with 1-bit stretch.

- Think: Proof?
- Proof Idea: Use next-bit unpredictability. Since first n bits of the output are uniformly distributed (since f is a permutation), any adversary for next-bit unpredictability with non-negligible advantage $\frac{1}{p(n)}$ must be predicting the $(n+1)$th bit with advantage $\frac{1}{p(n)}$. Build an adversary for hard-core predicate to get a contradiction.

Pseudorandom Functions

Going beyond Poly Stretch

- PRGs can only generate polynomially long pseudorandom strings

Going beyond Poly Stretch

- PRGs can only generate polynomially long pseudorandom strings
- Think: How to efficiently generate exponentially long pseudorandom strings?

Going beyond Poly Stretch

- PRGs can only generate polynomially long pseudorandom strings
- Think: How to efficiently generate exponentially long pseudorandom strings?

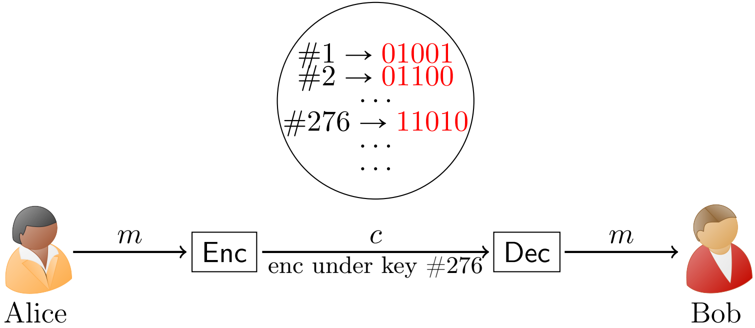
Going beyond Poly Stretch

- PRGs can only generate polynomially long pseudorandom strings
- Think: How to efficiently generate exponentially long pseudorandom strings?

Idea: Functions that index exponentially long pseudorandom strings

Motivation

- Imagine if Alice and Bob had an exponential amount of shared randomness - not just a short key.
- They could split it up into λ-bit chunks and use each one as a one-time pad whenever they want to send an encrypted message of length λ.



- Although Alice publicly announces which location/chunk was used as each OTP key, Eve doesn't know the value at that location.

Random Functions

- How do we define a random function?

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;
  - against each input, second column has the function value

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;
  - against each input, second column has the function value
  - i.e., each row is of the form $(x, F(x))$

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;
  - against each input, second column has the function value
  - i.e., each row is of the form $(x, F(x))$
- The size of the table for $F = 2^n \times n = n2^n$

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \rightarrow \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;
  - against each input, second column has the function value
  - i.e., each row is of the form $(x, F(x))$
- The size of the table for $F = 2^n \times n = n2^n$
- Total number of functions mapping n bits to n bits $= 2^{n2^n}$

Random Functions

There are two ways to define a random function:

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F
  - M keeps a table T that is initially empty.

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F
  - M keeps a table T that is initially empty.
  - on input x, M has not seen x before, choose a random string y and add the entry (x, y) to the table T

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F
  - M keeps a table T that is initially empty.
  - on input x, M has not seen x before, choose a random string y and add the entry (x, y) to the table T
  - otherwise, if x is already in the table, M picks the entry corresponding to x from T, and outputs that

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected uniformly at random from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F
  - M keeps a table T that is initially empty.
  - on input x, M has not seen x before, choose a random string y and add the entry (x, y) to the table T
  - otherwise, if x is already in the table, M picks the entry corresponding to x from T, and outputs that
- M's output distribution identical to that of F.

Random Functions

- Truly random functions are huge random objects

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently
- With the second method, we can support **polynomial** calls to the function efficiently because M will only need polynomial space and time to store and query T

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently
- With the second method, we can support **polynomial** calls to the function efficiently because M will only need polynomial space and time to store and query T
- Can we use some crypto magic to make a function F' so that:

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently
- With the second method, we can support **polynomial** calls to the function efficiently because M will only need polynomial space and time to store and query T
- Can we use some crypto magic to make a function F' so that:
  - it "looks like" a random function

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently
- With the second method, we can support **polynomial** calls to the function efficiently because M will only need polynomial space and time to store and query T
- Can we use some crypto magic to make a function F' so that:
  - it "looks like" a random function
  - but actually needs much fewer bits to describe/store/query?

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does “looks like” mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the description of either a random function or a PRF?

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the description of either a random function or a PRF?
- **Main Issue:** A random function is of exponential size

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the description of either a random function or a PRF?
- **Main Issue:** A random function is of exponential size
  - D can't even read the input efficiently

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the description of either a random function or a PRF?
- **Main Issue:** A random function is of exponential size
  - D can't even read the input efficiently
  - D can tell by looking at the size

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the description of either a random function or a PRF?
- **Main Issue:** A random function is of exponential size
  - D can't even read the input efficiently
  - D can tell by looking at the size
- **Idea**: D can only query the function on inputs of its choice, and see the output.

Pseudorandom Functions

- Keep the description of PRF **secret** from D?

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF
  - D send queries $x_1, x_2, \ldots$ to Ch, one-by-one

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF
  - D send queries $x_1, x_2, \ldots$ to Ch, one-by-one
  - Ch answers by correctly replying $F(x_1), F(x_2), \ldots$

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF
  - D send queries $x_1, x_2, \ldots$ to Ch, one-by-one
  - Ch answers by correctly replying $F(x_1), F(x_2), \ldots$
  - Finally, D outputs his guess b' (of F being random or PRF)

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF
  - D send queries $x_1, x_2, \ldots$ to Ch, one-by-one
  - Ch answers by correctly replying $F(x_1), F(x_2), \ldots$
  - Finally, D outputs his guess b' (of F being random or PRF)
  - D wins if $b' = b$

Pseudorandom Functions

- Keep the description of PRF **secret** from D?
  - Security by obscurity not a good idea (Kerckoff's priniciple)
- Solution: PRF will be a keyed function. Only the key will be secret, and the PRF evaluation algorithm will be public
- **Security via a Game based definition**
  - Players: a **challenger** Ch and D. Ch is randomized and efficient
  - Game starts by Ch choosing a random bit b. If $b = 0$, Ch implements a random function, otherwise it implements a PRF
  - D send queries $x_1, x_2, \ldots$ to Ch, one-by-one
  - Ch answers by correctly replying $F(x_1), F(x_2), \ldots$
  - Finally, D outputs his guess b' (of F being random or PRF)
  - D wins if $b' = b$
- PRF Security: No D can win with probability better than $1/2$.

Pseudorandom Functions: Definition

Definition (Pseudorandom Functions)

A family $\{F_k\}_{k\in\{0,1\}^n}$ of functions, where : $F_k : \{0,1\}^n \to \{0,1\}^n$ for all k, is pseudorandom if:

Pseudorandom Functions: Definition

Definition (Pseudorandom Functions)

A family $\{F_k\}_{k \in \{0,1\}^n}$ of functions, where : $F_k : \{0,1\}^n \rightarrow \{0,1\}^n$ for all k, is pseudorandom if:

- **Easy to compute:** there is an efficient algorithm M such that $\forall k, x : M(k,x) = F_k(x)$.

Pseudorandom Functions: Definition

Definition (Pseudorandom Functions)

A family $\{F_k\}_{k \in \{0,1\}^n}$ of functions, where : $F_k : \{0,1\}^n \to \{0,1\}^n$ for all k, is pseudorandom if:

- **Easy to compute:** there is an efficient algorithm M such that $\forall k, x : M(k, x) = F_k(x)$.
- **Hard to distinguish:** for every non-uniform PPT D there exists a negligible function ν such that $\forall n \in \mathbb{N}$:

Pseudorandom Functions: Definition

Definition (Pseudorandom Functions)

A family $\{F_k\}_{k \in \{0,1\}^n}$ of functions, where : $F_k : \{0,1\}^n \rightarrow \{0,1\}^n$ for all k, is pseudorandom if:

- **Easy to compute:** there is an efficient algorithm M such that $\forall k, x : M(k, x) = F_k(x)$.
- **Hard to distinguish:** for every non-uniform PPT D there exists a negligible function ν such that $\forall n \in \mathbb{N}$:

$$|\Pr[D \text{ wins } \mathsf{GuessGame}] - 1/2| \leqslant \nu(n).$$

where $\mathsf{GuessGame}$ is defined below

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)
  - (keep a table T for previous answers)

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)
  - (keep a table T for previous answers)
  - if x is in T: return $T[x]$.
  - else: choose $y \leftarrow \{0,1\}^n$, $T[x] = y$, return y.

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)
  - (keep a table T for previous answers)
  - if x is in T: return $T[x]$.
  - else: choose $y \leftarrow \{0,1\}^n$, $T[x] = y$, return y.
- Game stops when D halts. D outputs a bit b'

D **wins GuessGame if** $b' = b$**.**

Pseudorandom Functions: Game Based Definition

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)
  - (keep a table T for previous answers)
  - if x is in T: return $T[x]$.
  - else: choose $y \leftarrow \{0,1\}^n$, $T[x] = y$, return y.
- Game stops when D halts. D outputs a bit b'

D **wins** GuessGame **if** $b' = b$**.**

Remark: note that for any b only one of the two functions is ever used.

Pseudorandom Functions (contd.)

- Think: How can we construct a PRF?

Pseudorandom Functions (contd.)

- Think: How can we construct a PRF?
- Use PRG?

Pseudorandom Functions (contd.)

- Think: How can we construct a PRF?
- Use PRG?
- **Simpler problem:** build PRF for just 1-bit inputs using PRG

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \rightarrow \{0,1\}^n$

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \to \{0,1\}^n$
- G is length doubling, so let

$$G(s) = y_0 \| y_1$$

where $|y_0| = |y_1| = n$

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \to \{0,1\}^n$
- G is length doubling, so let

$$G(s) = y_0 \| y_1$$

  where $|y_0| = |y_1| = n$
- PRF: Set $k = s$ and,

$$F_k(0) = y_0, \; F_k(1) = y_1$$

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \rightarrow \{0,1\}^n$
- G is length doubling, so let

$$G(s) = y_0 \| y_1$$

  where $|y_0| = |y_1| = n$
- PRF: Set $k = s$ and,

$$F_k(0) = y_0, \; F_k(1) = y_1$$

- Think: What about n-bit inputs?

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \rightarrow \{0,1\}^n$
- G is length doubling, so let

$$G(s) = y_0 \| y_1$$

  where $|y_0| = |y_1| = n$
- PRF: Set $k = s$ and,

$$F_k(0) = y_0, \ F_k(1) = y_1$$

- Think: What about n-bit inputs?
  - Idea for 1-bit case: "double and choose"

From PRG to PRF with 1-bit input

- Let G be a length doubling PRG
- Want: $\{F_k\}$ such that $F_k : \{0,1\} \to \{0,1\}^n$
- G is length doubling, so let

$$G(s) = y_0 \| y_1$$

  where $|y_0| = |y_1| = n$
- PRF: Set $k = s$ and,

$$F_k(0) = y_0, \; F_k(1) = y_1$$

- Think: What about n-bit inputs?
  - Idea for 1-bit case: "double and choose"
  - For general case: Apply the "double and choose" idea repeatedly!

PRF from PRG

Theorem (Goldreich-Goldwasser-Micali (GGM))

If pseudorandom generators exist then pseudorandom functions exist

PRF from PRG

Theorem (Goldreich-Goldwasser-Micali (GGM))

If pseudorandom generators exist then pseudorandom functions exist

- **Notation:** define G_0 and G_1 as

$$G(s) = G_0(s) \| G_1(s)$$

i.e., G_0 chooses left half of G and G_1 chooses right half

PRF from PRG

Theorem (Goldreich-Goldwasser-Micali (GGM))

If pseudorandom generators exist then pseudorandom functions exist

- **Notation:** define G_0 and G_1 as

$$G(s) = G_0(s) \| G_1(s)$$

  i.e., G_0 chooses left half of G and G_1 chooses right half

- Construction for n-bit inputs $x = x_1 x_2 \ldots x_n$

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\ldots\big(G_{x_1}(k)\big)_{..}\big)$$

PRF from PRG (contd.)

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\ldots\big(G_{x_1}(k)\big)_{..}\big)$$

- We can represent F_k as a binary tree of size 2^n

PRF from PRG (contd.)

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\dots\big(G_{x_1}(k)\big)_{..}\big)$$

- We can represent F_k as a binary tree of size 2^n
- The root corresponds to k

PRF from PRG (contd.)

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\ldots\big(G_{x_1}(k)\big)_{..}\big)$$

- We can represent F_k as a binary tree of size 2^n
- The root corresponds to k
- Left and right child on level 1 and 2 are:

$$k_0 = G_0(k) \text{ and } k_1 = G_1(k)$$

PRF from PRG (contd.)

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\dots\big(G_{x_1}(k)\big)_{..}\big)$$

- We can represent F_k as a binary tree of size 2^n
- The root corresponds to k
- Left and right child on level 1 and 2 are:

$$k_0 = G_0(k) \text{ and } k_1 = G_1(k)$$

- Second level children:

$$k_{00} = G_0(k_0),\ k_{01} = G_1(k_0),\ k_{10} = G_0(k_1),\ k_{11} = G_1(k_1)$$

PRF from PRG (contd.)

$$F_k(x) = G_{x_n}\big(G_{x_{n-1}}\big(\ldots\big(G_{x_1}(k)\big)_{..}\big)$$

- We can represent F_k as a binary tree of size 2^n
- The root corresponds to k
- Left and right child on level 1 and 2 are:

$$k_0 = G_0(k) \text{ and } k_1 = G_1(k)$$

- Second level children:

$$k_{00} = G_0(k_0),\ k_{01} = G_1(k_0),\ k_{10} = G_0(k_1),\ k_{11} = G_1(k_1)$$

- At level ℓ, 2^ℓ nodes, one for each path, denoted by $k_{x_1 \ldots x_\ell}$

Proof Strategy

- Let's use Hybrid Arguments!

Proof Strategy

- Let's use Hybrid Arguments!
- Problem: If we replace each node in the tree one-by-one with random, then exponentially many hybrids. Hybrid lemma doesn't apply!

Proof Strategy

- Let's use Hybrid Arguments!
- Problem: If we replace each node in the tree one-by-one with random, then exponentially many hybrids. Hybrid lemma doesn't apply!
- **Observation:** Efficient adversary can only make polynomial queries

Proof Strategy

- Let's use Hybrid Arguments!
- Problem: If we replace each node in the tree one-by-one with random, then exponentially many hybrids. Hybrid lemma doesn't apply!
- **Observation:** Efficient adversary can only make polynomial queries
- Thus, only need to change polynomial number of nodes in the tree

Proof Strategy (contd.)

Two layers of hybrids:

- First, define hybrids over the n levels in the tree. For every i, H_i is such that the nodes up to level i are random, but the nodes below are pseudorandom.

Proof Strategy (contd.)

Two layers of hybrids:

- First, define hybrids over the n levels in the tree. For every i, H_i is such that the nodes up to level i are random, but the nodes below are pseudorandom.
- If H_1 and H_n are distinguishable with non-negligible advantage, then use hybrid lemma to find level i s.t. H_i and H_{i+1} are also distinguishable with non-negligible advantage

Proof Strategy (contd.)

Two layers of hybrids:

- First, define hybrids over the n levels in the tree. For every i, H_i is such that the nodes up to level i are random, but the nodes below are pseudorandom.
- If H_1 and H_n are distinguishable with non-negligible advantage, then use hybrid lemma to find level i s.t. H_i and H_{i+1} are also distinguishable with non-negligible advantage
- Now, hybrid over the nodes in level $i+1$ that are "affected" by adversary's queries, replacing each node one by one with random

Proof Strategy (contd.)

Two layers of hybrids:

- First, define hybrids over the n levels in the tree. For every i, H_i is such that the nodes up to level i are random, but the nodes below are pseudorandom.
- If H_1 and H_n are distinguishable with non-negligible advantage, then use hybrid lemma to find level i s.t. H_i and H_{i+1} are also distinguishable with non-negligible advantage
- Now, hybrid over the nodes in level $i+1$ that are "affected" by adversary's queries, replacing each node one by one with random
- Use hybrid lemma again to identify one node that is changed from pseudorandom to random and break PRG's security to get a contradiction

Proof Details

- Must make sure that all hybrids are implementable in polynomial time

Proof Details

- Must make sure that all hybrids are implementable in polynomial time
- Will use two key points to ensure this:

Proof Details

- Must make sure that all hybrids are implementable in polynomial time
- Will use two key points to ensure this:
  1. Adversary only makes polynomial number of queries

Proof Details

- Must make sure that all hybrids are implementable in polynomial time
- Will use two key points to ensure this:
  1. Adversary only makes polynomial number of queries
  2. A random function can be efficiently implemented (using second method) if the number of queries are polynomial

Proof Details

- Must make sure that all hybrids are implementable in polynomial time
- Will use two key points to ensure this:
  1. Adversary only makes polynomial number of queries
  2. A random function can be efficiently implemented (using second method) if the number of queries are polynomial
- Think: Formal proof?

Concluding Remarks

- **Efficient PRFs from concrete assumptions:**
  [Naor-Reingold97], [Banerjee-Peikert-Rosen12]

Concluding Remarks

- **Efficient PRFs from concrete assumptions:** [Naor-Reingold97], [Banerjee-Peikert-Rosen12]
- **Constrained PRFs:** PRFs with "punctured" keys that are disabled on certain inputs [Boneh-Waters13, Kiayias-Papadopoulos-Triandopoulos-Zacharias13, Boyle-Goldwasser-Ivan14, Sahai-Waters14]

Concluding Remarks

- **Efficient PRFs from concrete assumptions:** [Naor-Reingold97], [Banerjee-Peikert-Rosen12]
- **Constrained PRFs:** PRFs with "punctured" keys that are disabled on certain inputs [Boneh-Waters13, Kiayias-Papadopoulos-Triandopoulos-Zacharias13, Boyle-Goldwasser-Ivan14, Sahai-Waters14]
- **Related-key Security:** Evaluation of $F_s(x)$ does not help in predicting $F_{s'}(x)$ [Bellare-Cash10]

Concluding Remarks

- **Efficient PRFs from concrete assumptions:** [Naor-Reingold97], [Banerjee-Peikert-Rosen12]
- **Constrained PRFs:** PRFs with "punctured" keys that are disabled on certain inputs [Boneh-Waters13, Kiayias-Papadopoulos-Triandopoulos-Zacharias13, Boyle-Goldwasser-Ivan14, Sahai-Waters14]
- **Related-key Security:** Evaluation of $F_s(x)$ does not help in predicting $F_{s'}(x)$ [Bellare-Cash10]
- **Key-homomorphic PRFs:** Given $f_s(x)$ and $f_{s'}(x)$, compute $f_{g(s,s')}(x)$ [Boneh-Lewi-Montgomery-Raghunathan13]