Class CS 6903, Lecture n. 4

- Welcome to Lecture 4!
- In Lectures 1-3 we studied:
  - Classical cryptography, encryption with perfect secrecy
  - Background on algorithms, complexity theory. Modern cryptography: principles, primitives, and a public-key cryptosystem
  - Algorithmic number theory, number theory and cryptographic assumptions, reductions, proofs by reductions, number theory candidates for cryptographic primitives and schemes

Summary of Lecture 4

- Randomness and pseudo-randomness
- Pseudo-random generators
- Pseudo-random functions
- Pseudo-random permutations

Randomness

- Can one generate random bits?
- Using sources generated using physical processes; e.g.:
  - coins, dices
  - clock drift
  - quantum processes
  - computer memory state
- Two main problems:
  - Bias (e.g.: prob[0] = 0.49)
  - Correlation (e.g.: 3rd bit = 1st bit)
- Dealing with bias: Von Neumann's pairing trick to remove bias:
  - Draw 2 bits from the source
  - If "00" or "11", ignore
  - If "01", return: "0"
  - If "10", return: "1"
- Dealing with Correlation: Assume source has some type of correlation; process source output using deterministic functions (e.g., inner products) to obtain almost uncorrelated randomness,

In cryptography we assume that there exists a source of random bits with no bias and no correlation, but acknowledge that this source is expensive and thus try to minimize its use

Pseudo-randomness

Expanding short random string (seed) into much longer (pseudo-random) string

Before Modern Cryptography

- Classical notion: string is pseudo-random if it passes certain statistical tests (e.g., frequency, hypothesis testing, etc.)
- Main application: Monte-Carlo simulation
- Ex.1: Linear Congruential generators
  - Given random a,b,m,x(0), return x(1),..,x(n), where x(i)=a*x(i-1)+b mod m
- Ex.2: Linear Feedback Shift Registers
  - Given n bits x(1),..,x(n), return x(n+1) = xor of some bits among x(1),…,x(n)

Modern Cryptography:

- Following principles of modern cryptography
- Two definitional approaches
- 1) String is pseudo-random if it passes all statistical tests against an efficient (poly-time) adversary
- 2) String is pseudo-random if no efficient (poly-time) adversary can predict the next returned bit from previous ones
- Solutions (to generate pseudo-randomness) are based on cryptographic problems, primitives and assumptions about those

Summary of Lecture 4

- Randomness and pseudo-randomness
- Pseudo-random generators
- Pseudo-random functions
- Pseudo-random permutations

Computational indistinguishability

- We will use the notion of indistinguishability to define pseudo-randomness and security properties of cryptographic primitives and protocols
- How can we say that two entities are indistinguishable? We model that after the Turing Test (for AI)! See https://www.youtube.com/watch?v=sXx-PpEBR7k
- Given random variables X,Y, we say that X and Y are computationally indistinguishable if for any efficient algorithm A, it holds that the difference |p(X)-p(Y)| is negligible, where
  - p(X)=Prob[x←X : A(x)=1]
  - p(Y)=Prob[y←Y: A(y)=1]
- Note:
  - Random variables ~ distributions
  - Single – sample computational indistinguishability
- Given random variables X,Y, we say that X and Y are multiple-sample computationally indistinguishable if for any efficient algorithm A, for any polynomials a,b, it holds that |p(X)-p(Y)| is negligible, where, for any Z,
  - p(Z)=Prob[z(1),..,z(a(n))←Z : A(z(1),..,z(b(n)))=1]
- Theorem: Two random variables are computationally indistinguishable if and only if they are multiple-sample computationally indistinguishable

Pseudo-random Generators

Generator: stretching function (from $\{0,1\}^n$ to $\{0,1\}^m$, $m>=n+1$)

Next-bit test

- Informally: A test that tries to predict (q+1)-th bit given the first q ones
- Formally: generator $G:\{0,1\}^n \rightarrow \{0,1\}^m$ passes all next bit tests if
  - G is efficiently computable
  - For any efficient A, the probability that r = r(m) is <= ½ + negligible, where
    - seed $\leftarrow \{0,1\}^n$;
    - $(r(1),..,r(m)) \leftarrow G(seed)$;
    - $r \leftarrow A(r(1),..,r(m-1))$

Statistical test

- Informally: A test trying to distinguish random from pseudo-random strings
- Formally: generator $G:\{0,1\}^n \rightarrow \{0,1\}^m$ passes all statistical tests if
  - G is efficiently computable
  - Distributions D(G) and U(m) are computationally indistiguishable, where
    - D(G) is the distribution over $\{0,1\}^m$ obtained by randomly choosing s in $\{0,1\}^n$ and returning G(s)
    - U(m) is uniform distr. over $\{0,1\}^m$
    - (See previous slide for definition of computational indistinguishability)

- Theorem: A generator passes all next-bit tests if and only if it passes all statistical tests
- Note: both linear congruential generators and linear feedback shift registers do not satisfy any of these two definitions (due to a Gaussian elimination attack)

From Pseudo-randomness to Hardness

- Hardness (of inverting one-way functions)
- Pseudo-randomness (of output of pseudo-random generators)
- Theorem: If there exists a pseudo-random generator then there exists a one-way function
- Sketch of proof: we define function F using G; i.e., F(x)=G(x) for all x in $\{0,1\}^n$ (if we prove that F is one-way we are done)
- Assume, towards contradiction, that F is not one-way; i.e., there exists an efficient algorithm A that can invert F with not negligible probability
  - On input y, A returns x' s.t. f(x')=y with not negligible probability
- By definition of F, A inverts G with not negligible probability
- A can be used to construct a statistical test T that G fails or alternatively, an algorithm B (i.e., the test) that distinguishes whether a string y is random or pseudo-random with not negligible probability
  - Test: return "random" if A(y) does not return a valid seed or else "pseudo-random"
- This contradicts the fact that G is a pseudo-random generator

From Hardness to Pseudo-randomness

Theorem: If there exists a one-way function then there exists a pseudo-random generator

- Only consider particular cases here:
  - F is a one-way permutations
  - G only stretches input by 1 bit
- Note: in general we want an arbitrary function F and a G that stretches input by a polynomial number of bits
- Construction (based on hard-core predicate HC for F):
  - G(s)=F(s)|HC(s)
- Intuition:
  - F(s) is pseudo-random (in fact, random) as s is random and F is a permutation
  - HC(s) is pseudo-random as it is unpredictable given F(s)

- Proof sketch: (hybrid technique)
  Define 3 distributions, returning:
  - G(s) = F(s)|HC(s)
  - Hybrid = F(s)|b, b random bit
  - R = r|b, r random n-bit string
- If G is not pseudo-random, there exists an efficient algorithm A distinguishing G(s) from R with not negligible probability; then A either does the same for G(s) and Hybrid or for Hybrid and R
- In the first case, it violates HC properties; the second case cannot happen as Hybrid and R are the same

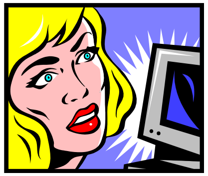
Expanding Pseudo-randomness

Theorem: If there exists a pseudo-random generator G: $\{0,1\}^n \rightarrow \{0,1\}^{n+1}$ then there exists a pseudo-random generator H: $\{0,1\}^n \rightarrow \{0,1\}^m$ for any m=poly(n)

- Idea: cascading m-n applications of G, each generating one new pseudo-random bit
- Construction of H:
  - s(0)=seed
  - For i=1,…,m-n,
    - u(i)=G(s(i-1))
    - Write u(i) as the concatenation of n-bit string s(i) and bit b(i)
  - Return: (s(m-n)|b(1)|…|b(m-n))
- Intuition:
  - Since seed is random, s(0) is pseudorandom
  - s(i) pseudo-random → so is s(i+1)
- Proof sketch: Define distributions:
  - output of H on input random s
  - i=1,…,m-n: Hybrid(i) = on input s, return H's output where first i values u(i) are randomly chosen
  - m-bit random string R
- H not pseudo-random,→ there exists efficient algorithm A distinguishing H(s) from R with not negligible probability → A must distinguish
  - H(s) from Hybrid(1),
  - Hybrid(i) from Hybrid(i+1) or
  - Hybrid(m-n) from R
- In all cases, A violates G's pseudo-randomness

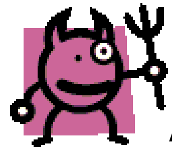
Using PRGs for Private-Key Encryption

Alice



Private key: k

Adversary



Bob

Private key: k

Sets c=G(k) xor m

c

Computes: m'=G(k) xor c

- One-time pad definition: Enc(k,m) = k xor m, Dec(k,c) = k xor c
- Properties: perfect secrecy (+), key length = message length (-)
- Using pseudo-random generator G:
  - Enc(k,m) = G(k) xor m
  - Dec(k,c) = c xor G(k)
- Properties: key length << message length (+), computational secrecy (-)
- This "computational secrecy" notion put forward in modern cryptography is a very acceptable compromise in practice

Question set 10

- Let $G:\{0,1\}^n \rightarrow \{0,1\}^m$ be a pseudo-random generator. Which distribution is G computationally indistinguishable from?

- Let G be a pseudo-random generator. Is G a one-way function?
- Let F be a one-way function. Is F a pseudo-random generator?

- Fill a 3x3 table whose entries indicate which of the statements

  "if there exists A then there exists B"

  is true or unknown or false, where A and B are taken from set {one-way functions, one-way permutations, pseudo-random generators}

Summary of Lecture 4

- Randomness and pseudo-randomness
- Pseudo-random generators
- Pseudo-random functions
- Pseudo-random permutations

Pseudo-random functions

Random Functions

- Definition: function takes as input x and returns uniformly and independently distributed value r=r(x)
- Note:
  - It is a function: when called twice on the same input x, returns the same output
  - Could be a great tool when randomness is necessary
  - Does not have short description

Pseudo-random functions:

- Goal: achieve effect similar to random functions with respect to polynomial time observers
- Input: short random key k + additional input x
- Output: value y that "looks" pseudo-random (assuming k is random and remains secret)
- Adversary should be allowed to see polynomially many values y's for different x's but same, secret k
- PRFs are defined as functions that are "multiple-sample computationally indistinguishable" from a random function by any efficient "oracle adversary"

Pseudo-random Function: definition

- The adversary attacking PRFs will have access to a function, called oracle, $O:\{0,1\}^n \rightarrow \{0,1\}^n$ used as a black-box fashion.
- An efficient algorithm A is an oracle adversary if it is given access to oracle O and can repeat the following for polynomially many times:
  - On input x(1),y(1),..,x(i),y(i), compute x(i+1)
  - Call oracle O on input x(i+1)
  - Set y(i+1) be the response obtained from O
- Adversary A given access to oracle O is also denoted A^O
- For any n, let R be the set of all functions $r : \{0,1\}^n \rightarrow \{0,1\}^n$, and let $f : \{0,1\}^n * \{0,1\}^n \rightarrow \{0,1\}^n$ be a function. Consider the following probabilistic experiment INIT:
  - Uniformly choose r from R
  - Uniformly choose s from $\{0,1\}^n$ for each n
  - Set f=f(s,.)
- f is a pseudo-random function if for any efficient oracle adversary, the difference |p(real)-p(rand)| is negligible, where
  - p(real)=prob[INIT;O$\leftarrow$f(s,.): A^O=1]
  - p(rand)=prob[INIT;O$\leftarrow$r : A^O=1]

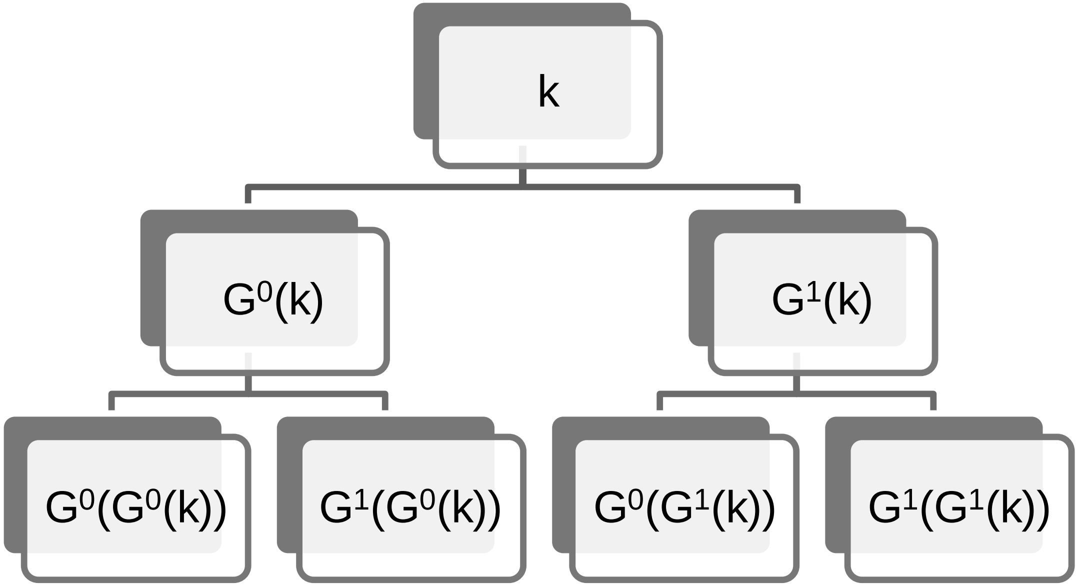
PRFs and PRGs: comparisons and theorem

- PRFs more powerful than PRGs:
  - PRGs return a polynomially long pseudo-random string
  - PRFs return an exponentially long string with
    - efficient direct access (through input x)
    - pseudo-randomness for each polynomially long substring
- PRGs more efficient and easier to construct than PRFs
- Theorem: A pseudo-random function exists if and only if a pseudo-random generator exists
- Corollary: A pseudo-random function exists if and only if a one-way function exists
- Corollary: Pseudo-random generators and functions exists if
  - Factoring is hard
  - The RSA problem is hard
- PRGs from PRFs, proof sketch: note that G defined as $G(s) = F(s,0)|F(s,1)|\ldots$ is a pseudo-random generator (that is, set any distinct values for the input x until the total output is longer than $|s|$)

PRFs from PRGs

- Extending any generator stretching its seed into a polynomially-longer pseudo-random string to a function returning an exponentially-long pseudo-random string with efficient direct access (via input x)

- Construction: Tree-based applications of 2n-bit stretching pseudo-random generator, and left/right choices according to 0/1 value of bits in x
- Let $G:\{0,1\}^n \rightarrow \{0,1\}^{2n}$ be a pseudo-random generator
  - Let G^0 be function returning first n bits returned by G
  - Let G^1 be function returning second n bits returned by G
- Define function f(s,x) = y, where
  - seed s is the key for f,
  - $y=(G^{x(n)}(\ldots(G^{x(2)}(G^{x(1)}(s)))\ldots))$,
  - x=x(1),..,x(n) and each x(i) is a bit



Construction example for x such that |x|=2

Every input value x is associated with a path from the root to a unique leaf, whose value is used as the function's output on input x

PRFs from PRGs: proof sketch

- Intuition: an appropriate generalization of the proof for the PRG output expansion theorem
- Define distributions:
  - Answer A's queries using output of function f(k,.)
  - i=1,…,|x|: answer A's queries using output of Hybrid(i) function, defined as function f(k,.) , with the difference that $G^{x(1)},\ldots,G^{x(i)}$ functions are now computed as random functions
  - Answer A's queries using output of a single random function R
- If f(k,.) is not a pseudo-random function, there exists an efficient oracle algorithm A distinguishing f(k,.) from R with not negligible probability; then A must distinguish with not negligible probability at least one of the following:
  - f(k,.) from Hybrid(1)
  - Hybrid(i) from Hybrid(i+1) for some i in {1,..,|x|-1}
  - Hybrid(n) from R
- In all cases (but last one), it violates G's pseudo-randomness
- Last case is not possible as two functions are identical

Summary of Lecture 4

- Randomness and pseudo-randomness
- Pseudo-random generators
- Pseudo-random functions
- Pseudo-random permutations

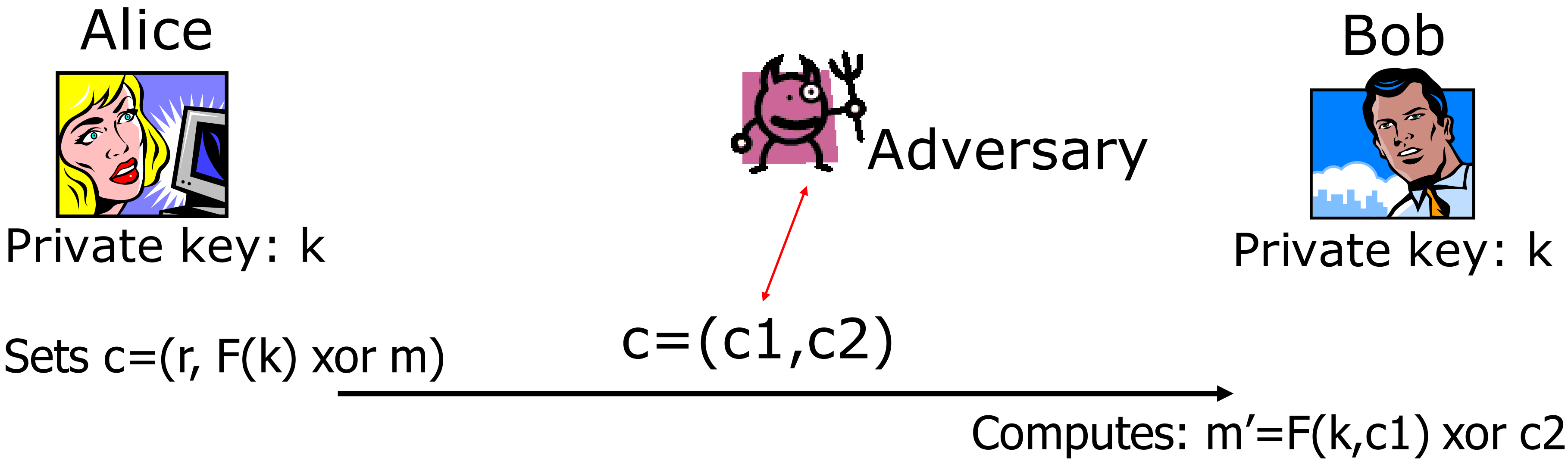
Pseudo-random Permutations

- Defined like pseudo-random functions, with the difference that all functions involved are permutations
  - A crucial difference in applications like encryption
- Theorem: A pseudo-random permutation exists if and only if a pseudo-random function exists
  - Note: similar theorem is not known for one-way functions
- Corollary: A pseudo-random permutation exists if and only if a one-way function exists
- Corollary: A pseudo-random permutation exists if
  - factoring is hard, or
  - the RSA problem is hard

PRPs from PRFs: construction

- Construction: 3-round application of Feistel transform
- Feistel transform: given a pseudo-random function f, on input (L,R) the transform returns (L’,R’), where
  - L’=R and R’=L xor f(k, R)
- 3-round Feistel transform: cascading Feistel transform 3 times; that is, on input (L(0),R(0)) returns (L(3),R(3)), where
  - L(1)=R(0), R(1)=L(0) xor f(k, R(0))
  - L(2)=R(1), R(2)=L(1) xor f(k, R(1))
  - L(3)=R(2), R(3)=L(2) xor f(k, R(2))
- Results:
  - Feistel transform is a permutation: R=L’ and L= R’ xor f(k, R)
  - 3-round transform is pseudo-random (proof by hybrid argument)
  - 1-round and 2-round versions are not pseudo-random
  - 4-round version is (super-)pseudo-random, that is, pseudo-random even if adversary A is allowed to query both O and O’s inverse

Applications of PRFs and PRPs to Private-Key Encryption



- One-time pad definition: Enc(k,m) = k xor m, Dec(k,c) = k xor c
- Properties: perfect secrecy (+), key length = message length (-)
- Using pseudo-random functions F:
  - Enc(k,m) = (r, F(k,r) xor m), where r is a random string
  - Dec(k,(c1,c2)) = F(k,c1) xor c2
- Properties: key length << message length (+), computational secrecy (-), higher security than pseudo-random generator construction (+)
  - Ex.: can securely encrypt m(1), m(2) using the same k

Applications and Practical Considerations

- Certain applications (e.g., in finance) do not need cryptographically secure PRGs
  - Linear congruential generators (and various improved ones) are still used
- In cryptography PRGs are useful in any protocol where parties use random bits
  - Essentially all protocols
- In addition to PRGs based on number-theoretic assumptions, there are other classes of constructions, such as block cipher based and heuristic constructions
  - The latter ones have less or no provable guarantees but are typically faster
- Many heuristic PRGs have been standardized
  - See Wikipedia (Cryptographically secure pseudo-random generators → Designs, Standards)
- PRFs and PRPs also have more efficient but heuristically secure constructions (typically based on block ciphers or hash functions, which we will study later)
  - See, for instance, http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf
  - See also the Mersenne twister at https://en.wikipedia.org/wiki/Mersenne_Twister

Question set 11

- Let $F:\{0,1\}^k x\{0,1\}^n \rightarrow \{0,1\}^n$ be a pseudo-random function. Which distribution is $(F(k,1),F(k,2),F(k,3),\ldots)$ computationally indistinguishable from, for a random k?
- Let $P:\{0,1\}^k x\{0,1\}^n \rightarrow \{0,1\}^n$ be a pseudo-random permutation. Which distribution is $(P(k,1),P(k,2),\ldots)$ computationally indistinguishable from, for a random k?

- Let F be a pseudo-random function. Is F(k,.) a one-way function?
- Let P be a pseudo-random permutation. Is P(k,.) a one-way permutation?
- Let F be a one-way function. Is F a pseudo-random function?
- Let P be a one-way permutation. Is P a pseudo-random permutation?

- Fill a 4x4 table whose entries indicate which of the statements

  "if there exists A then there exists B"

  is true or unknown or false, where A and B are taken from set {one-way functions, one-way permutations, pseudo-random functions, pseudo-random permutations}

Class CS 6903, End of Lecture n. 4

<table>
<tr><th>Reference →
Topic ↓</th><th>[KL]</th><th>[MOV]</th><th>[FSK]</th></tr>
<tr><td rowspan="2">Randomness, Pseudo-randomness before cryptography, Turing test</td><td colspan="3">http://en.wikipedia.org/wiki/Random_number_generation
http://en.wikipedia.org/wiki/Monte_Carlo_method
https://www.youtube.com/watch?v=sXx-PpEBR7k</td></tr>
<tr><td></td><td>5.1-5.4</td><td rowspan="4">9</td></tr>
<tr><td>Pseudo-random generators</td><td>3.3, 7.4, 7.8
(1st edition: 3.3, 6.4, 6.8)</td><td>5.5</td></tr>
<tr><td>Pseudo-random functions</td><td>6.5 (1st edition: 6.5)</td><td>2.3</td></tr>
<tr><td>Pseudo-random permutations</td><td>6.6 (1st edition: 6.6)</td><td></td></tr>
<tr><td>Applications, practical considerations</td><td colspan="3">Lecture Notes,
http://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf</td></tr>
</table>