CSCI-GA.3210-001 MATH-GA.2170-001	**Introduction to Cryptography** February 1, 2012

Lecture 2

Lecturer: Yevgeniy Dodis *Spring 2012*

This lecture begins with a discussion on secret-key and public-key cryptography, and then discusses One-way Functions (OWF), and their importance in cryptography. Essentially, an OWF is easy to compute, but difficult to invert. A One-way Permutation (OWP) is an OWF that permutes elements from a set. A Trapdoor Permutation (TDP) is essentially an OWP with certain secret information, that if disclosed, allows the function to be easily inverted.

No OWF is known to exist unconditionally, since showing the existence of OWFs is at least as hard as showing that complexity class P is different from NP — a long standing open problem. However, there exists several good candidates for OWF, OWP, and TDP. We provide such example later, but start with sample applications of general OWFs, OWPs and TDPs. First, we will see how the assumption of the existence of OWF leads to a secure password-authentication system. Next, we show that a widely used S/Key Password Authentication System is secure using any OWP (but not general OWFs). Finally, we deascribe a (very weak) public-key encryption scheme based on TDPs.

We also briefly provide (conjectured) number-theoretic candidates of a OWF, a OWP and a TDP: Prime Product as an example of a OWF candidate, Modular Exponentiation as an example of a OWP candidate, and RSA as an example of a TDP candidate. More detailed introduction to number theory will be given next lecture.

Finally, we describe some criticisms regarding OWFs, OWPs, and TDPs in practical applications, and give suggestions of how to overcome these criticisms.

1 Symmetric and Public-Key Encryption

We briefly recap these notions below.

1. **Secret-Key (Symmetric-Key) Encryption**

   - **Before the Encryption**
     Bob and Alice have some sort of secret key S they arranged to use in advance that Eve does not know.
   - **Encryption**
     When Bob wishes to send Alice a plaintext message M via the Internet, Bob encrypts M using secret key S to form a ciphertext C. (Formally, we summarize encryption with S as E_S, and say that $C = E_S(M)$.) Bob then sends C over the Internet to Alice.
   - **Decryption**

Upon receiving C, Alice uses the same secret key S to decrypt C, giving her M, the original plaintext message. (Formally, we summarize decryption with S as D_S and say that $D_S(C) = D_S(E_S(M)) = M$.)

- **Eve's Standpoint**
  Eve only sees C being sent over the Internet. She has no knowledge of S. Very informally (stay tuned!), the scheme is "Secure" if it is hard for Eve to learn about S or plaintexts based on the ciphertexts.

A more formal definition of symmetric-key encryption is given below (only syntax, not security yet!):

DEFINITION 1 [Symmetric-key encryption (SKE)] A SKE is a triple of PPT algorithms $\mathcal{E} = (G, E, D)$ where:

(a) G is the *key-generation algorithm*. $G(1^k)$ outputs $(S, \mathcal{M})$, where S is the shared secret key, and $\mathcal{M} = \mathcal{M}(S)$ is the (compact description of the) message space associated with the scheme. Here k is an integer usually called the *security parameter*, which determines the security level we are seeking for (i.e., everybody is polynomial in k and adversary's "advantage" should be negligible in k). Very often, S is simply a random string of length k, and $\mathcal{M}$ is $\{0,1\}^*$.

(b) E is the *encryption algorithm*. For any $m \in \mathcal{M}$, E outputs $c \xleftarrow{r} E_S(m, S)$ — the encryption of m. c is called the *ciphertext*. We usually write $E(m, S)$ as $E_S(m)$ (or $E_S(m;\ r)$, when E also takes randomness r and we wish to emphasize this fact).

(c) D is the (deterministic) *decryption algorithm*. $D(c, S)$ outputs a value $\tilde{m} \in \{\mathsf{invalid}\} \cup \mathcal{M}$, called the decrypted message. We also usually denote $D(c, S)$ as $D_S(c)$.

(d) We require the **correctness** property:

$$\forall m \in \mathcal{M}, \quad \tilde{m} = m, \quad \text{that is} \quad D_S(E_S(m)) = m$$

$\diamondsuit$

2. **Public-Key Encryption**

- **Before the Encryption**
  Alice publishes to the world her public key PK. Therefore, both Bob and Eve know what PK is. This public key is only used to encrypt messages, and a separate key SK is used to decrypt messages. (This is unlike the Secret-Key scheme where one key S is used to both encrypt and decrypt.) Only Alice knows what SK is, and nobody else, not even Bob.
- **Encryption**
  When Bob wishes to send Alice a plaintext message M via the Internet, Bob encrypts M using Alice's public key PK to form a ciphertext C. (Formally, we summarize encryption with PK as E_{PK} and say that $C = E_{PK}(M)$.) Bob then sends C over the Internet to Alice.

- **Decryption**
  Upon receiving C, Alice uses her secret private key SK to decrypt C, giving her M, the original plaintext message. (Formally, we summarize decryption with SK as D_{SK} and say that $D_{SK}(C) = D_{SK}(E_{PK}(M)) = M$.)
- **Eve's Standpoint**
  Unlike the Secret-Key scheme, Eve knows everything Bob knows and can send the same messages Bob can. And, only Alice can decrypt. And, when Bob sends his message, Eve only sees C, and knows PK in advance. But, she has no knowledge of SK. Very informally (stay tuned!), the system is "secure" if it is hard for Eve to learn about SK or plaintexts based on ciphertexts and PK.

A more formal definition of public-key encryption is given below (only syntax, not security yet!):

DEFINITION 2 [Public-key encryption (PKE)] A PKE is a triple of PPT algorithms $\mathcal{E} = (G, E, D)$ where:

(a) G is the *key-generation algorithm*. $G(1^k)$ outputs $(PK, SK, \mathcal{M})$, where SK is the secret key, PK is the public-key, and $\mathcal{M}$ is the (compact description of the) message space associated with the PK/SK-pair. As before, k is an integer the *security parameter*.

(b) E is the *encryption algorithm*. For any $m \in \mathcal{M}$, E outputs $c \xleftarrow{r} E(m,\ PK)$ — the encryption of m. c is called the *ciphertext*. We sometimes also write $E(m, PK)$ as $E_{PK}(m)$ (or $E_{PK}(m;\ r)$, when we want to emphasize the randomness r used by E).

(c) D is the (determinstic) *decryption algorithm*. $D(c,\ SK)$ outputs $\tilde{m} \in \{\mathsf{invalid}\} \cup \mathcal{M}$, called the decrypted message. We also sometimes denote $D(c,\ SK)$ as $D_{SK}(c)$.

(d) We require the **correctness** property:

$$\forall m \in M, \quad \tilde{m} = m, \quad \text{that is} \quad D_{SK}(E_{PK}(m)) = m$$

♢

2 Primitives

We mentioned the following three primitives commonly used in Cryptography:

1. OWF: One-Way Functions
2. OWP: One-Way Permutations
3. TDP: Trap-Door Permutations

The next sections will define these primitives and give conjectured candidates of each one (we say "candidate", and not "example", because the formal existence of OWF, and consequently OWP and TDP, has yet to be proven).

3 OWF

A function is One-way if it is easy to compute, but difficult to invert. More formally,

DEFINITION 3 [OWF] A function $f : \{0,1\}^* \to \{0,1\}^*$ (or, a function f that maps finite strings to finte strings) is *one-way* it is satisfies two properties:

1. $\exists$ poly-time algorithm which computes $f(x)$ correctly $\forall x$. (Thus, easy to compute.)
2. $\forall$ PPT Algorithm A,

$$\Pr(f(z) = y \mid x \leftarrow^R \{0,1\}^k;\ y = f(x);\ z \leftarrow^R A(y, 1^k)) \leq \mathsf{negl}(k)$$

where $\leftarrow^R$ means randomly chosen. (So x is randomly chosen from the set of k-bit numbers, and z is randomly outputted from algorithm A when it has y as input.) Thus, f is hard to invert. So, in polynomial time (in k) Eve has probability $\mathsf{negl}(k)$ or less of figuring out *any* preimage of $f(x)$.

$\diamondsuit$

And, keep in mind that no proof derived yet shows that OWF's exist. However, there's good evidence OWF's do exist.[1] And later on, candidate OWF's will be shown.

COLLECTION OF OWFS. The above definition of a OWF is very convenient to us when building other objects from OWFs. (In particular, we will usually use it in the homework and exams.) However, it is somewhat inconvenient to use if we want to *build* OWFs in practice. For one thing, the domain of a OWF is $\{0,1\}^*$, which is not the case for practical constructions. Also, the above definition does not allow one to randomly choose and publish some public parameters which woiuld actually define a given OWF f. Finally, we see that this definition will be hard to extend to *trapdoor permutations*, which we will do soon. Therefore, in order to include practical constructions, we define a collection of OWFs (which still suffice for all the applications of OWFs) as follows.

DEFINITION 4 [colelction of OWF] A *collection of one-way functions* is given be three PPT algorithms (Gen, Eval, Sample).

1. Gen is the *key-generation algorithm*. $G(1^k)$ outputs a public key PK, which implicitly defines a domain $D = D(PK)$ and the range $R = R(PK)$ of a given one-way function f_{PK} (which we simply denote by f when PK is clear from the context). As before, k is an integer the security parameter.
2. Eval is the (deterministic) *function evaluation algorithm*. $\mathsf{Eval}(x, PK)$ computes the value $f_{PK}(x) \in R$, for any $x \in D$. We usually write $f_{PK}(x)$, or just $f(x)$, to denote the output of $\mathsf{Eval}(x, PK)$.
3. Sample is a probabilistic algortihm, which, on input PK, samples a random element from the domain D of f_{PK}. We write $x \leftarrow D(PK)$, or just $x \leftarrow D$, as a shorthand for running $\mathsf{Sample}(PK)$.

[1] Those familiar with the Complexity Theory can observe that the existence of OWFs trivially implies that $P \neq NP$, which is a long-standing open problem. In fact, we do not know how to prove OWFs exist even if we assume that $P \neq NP$!

With this notation, the security of a collection of OWFs is very similar to security of OWFs defined earlier: $\forall$ PPT Algorithm A,

$$\Pr(f_{PK}(z) = y \mid PK \leftarrow \mathsf{Gen}(1^k);\ x \leftarrow D(PK);\ y = f_{PK}(x);\ z \leftarrow A(y, PK, 1^k)) \leq \mathsf{negl}(k)$$

$\diamondsuit$

As we can see, collection of OWFs allow us a bit more freedom in specifying the domain and range of f, as well as publishing some parameter PK describing f. We will see examples later on.

4 OWP

A function f is a OWP if it is OWF, and a permutation. More formally,

DEFINITION 5 [OWP] A function f is OWP if:

1. It satisfies all requirements for being OWF.
2. It is a permutation (that is every y has a unique preimage x).

$\diamondsuit$

Similarly, one can define *collection of* OWP*s* in the same manner as we defined collection of OWFs. The only addition is that $D(PK) = R(PK)$, and f_{PK} must be a permutation (be one-to-one and onto) over $D(PK)$.

5 TDP

A function f is a TDP if it is OWP, and given certain information, f can be inverted in PPT. More formally, one can attempt to define TDPs similar to Definition 3 and Definition 5 as follows. A function f is TDP if (a) it satisfies the requirements for OWP from Definition 5; and (b) There exists a poly-time algorithm I, some constant c, and a string t_k (for each k) such that, for all large enough k, the size of t_k is at most $O(k^c)$, and for any $x \in \{0,1\}^k$, $I(f(x), t_k) = z$ where $f(z) = f(x)$.

A moment relection, however, shows that this definition seems to be quite useless in applications. Indeed, one may wonder what is the usage of TDPs, if nobody can find the trapdoor information t_k in polynomial time (if one could, then one would be able to invert it, and contradict the one-wayness of f). Indeed, this observation is correct, and this is why TDPs are only defined as a *collection* of functions, similarly to a collection of OWFs and OWPs we defined earlier. Indeed, such collections have a key generation algortihm Gen which outputs the public key SK. Now, we simply let Gen also output the trapdoor information (which we denote SK) which would allow one to always invert a given TDP. More formally,

DEFINITION 6 [collection of TDP] A *collection of trapdoor permutations* is given be four PPT algorithms (Gen, Eval, Sample, Invert).

1. Gen is the *key-generation algorithm.* $G(1^k)$ outputs a public key PK, which implicitly defines a domain $D = D(PK)$ of a given function f_{PK} (which we simply denote by f when PK is clear from the context), and a corresponding secret (or *trapdoor*) key SK.

2. Eval is the (deterministic) *function evaluation algorithm.* $\mathsf{Eval}(x, PK)$ computes the value $f_{PK}(x) \in R$, for any $x \in D$. We usually write $f_{PK}(x)$, or just $f(x)$, to denote the output of $\mathsf{Eval}(x, PK)$.

3. Sample is a probabilistic algortihm which, on input PK, samples a random element from the domain D of f_{PK}. We write $x \leftarrow D(PK)$, or just $x \leftarrow D$, as a shorthand for running $\mathsf{Sample}(PK)$.

4. Invert is a (deterministic) inversion algortihm which, on input $y \in D$ and a secret key SK computes a value $z \in D(PK)$ such that $f_{PK}(z) = y$. We usually write $z = f_{PK}^{-1}(y)$, or just $z = f^{-1}(y)$ to denote the output of $\mathsf{Invert}(y, SK)$, with an implicit understanding that f^{-1} is only easy to compute with the knowledge of the trapdoor key SK.

The security of TDPs is idential to that of OWPs, where the attacker cannot invert f without the trapdoor key SK: $\forall$ PPT Algorithm A,

$$\Pr(z = x \mid (PK, SK) \leftarrow \mathsf{Gen}(1^k);\ x \leftarrow D(PK);\ y = f_{PK}(x);\ z \leftarrow A(y, PK, 1^k)) \leq \mathsf{negl}(k)$$

$\diamondsuit$

6 Number-Theoretic Examples of OWF, OWP, TDP

BEFORE YOU BEGIN: **If you are not very familiar with number theory, I strongly recommend you skip this section on first reading.** *Indeed, the main purpose of this somewhat dense section is to make the treatment of* OWF*,* OWP*s and* TDP*s a bit less "abstract", by giving examples we will study in more detail later. In particular, if "abstract" is fine with you, you can move directly to the next "application" section. In other words, the specific examples are not needed to understand the remainder of this lecture. We will study number theory in more detail the next lecture.*

As we said, we do not know how to unconditionally prove the existence of OWFs, OWPs and TDPs. However, we have several plausible candidates. In this section we give a sample candidate for each primitive based on *number theory.*[2]

6.1 OWF Candidate: Integer Multiplication

Let's define a function f as $f(p, q) = p * q$, where p and q are k-bit primes and $*$ is the regular integer multiplication. (So, our domain is the set of pairs of k-bit primes, and

[2]For OWPs and TDPs, all examples that we know use number theory. For OWFs, there are many other proposed candidates. However, we will study them a bit later in the course, concentrating on number-theoretic examples for now.

$f : \mathbb{P}_k * \mathbb{P}_k \to X$ where $\mathbb{P}_k$ is the set of k-bit primes, and X is the set of $2 * k$-bit numbers whose factorization is made up of two k-bit primes.) Clearly, f is not a permutation, but this is fine.

Let us denote $n = p * q$, so that $f(p, q) = p * q = n$. Upon seeing n, there's no known polynomial time algorithm A such that $A(n)$ outputs values p' and q' so that $p' * q' = n$ (since we assume n is a product of two primes, then either $p' = p, q' = q$, or $p' = q, q' = p$).

You might think, "Hey! That's not true! Just test all the numbers from 2 to $\sqrt{n}$." And propose a program that works like the following:

for $i = 2$ *to* $\sqrt{n}$ *do*
    if (i divides n) then output(i,$\frac{n}{i}$*);*

And you would then claim that your program runs in time $O(\sqrt{n})$, which is polynomial in terms of n. However, keep in mind that the number n inputted into this algorithm is of magnitude roughly 2^{2k} and of length $2k$. Thus, since $\sqrt{n} \approx \sqrt{2^{2k}} \approx 2^k$, this algorithm runs in time $O(2^k)$, which is exponential in terms of the input size. Thus, this algorithm runs in exponential time. And, no algorithm that's polynomial (or even probabilistically polynomial) in terms of k is known that can factor n.

Therefore, this function f is easy to compute and *seems* difficult to invert, making it a good candidate OWF.

Conjecture 1 *Integer multiplication is a* OWF.

This conjecture has stood up to attempts to disprove it for a long time, and forms one of the most common assumptions in cryptography. Usually, when we will use the above conjecture as an *assumption* to prove some result X, we will say

"If *factoring is hard*, then X is *provably* true."

Of course, this does not mean we have proven X unconditionally, but it means that the *only* way to break X is to break factoring, and this seems quite unlikely given the long history of this problem!

6.2 OWP Candidate: Modular Exponentiation

We will define the function $f(x, p, g) = (g^x \bmod p, p, g)$, where the explanation of the above letters will follows shortly. Notice, however, that since p and g are "copied" through, for convenience instead of this f we will write $f_{p,g}(x) = g^x \mod p$ (implicitly implying that p and g are part of the randomly generated input which are also part of the output). As we can see, in this example it is actually more convenient to use the notion of *collection* of OWPs. In this case, we can say that "copied" p and g are simply part of the randomly generated public key PK.

Let us now make some important math definitions, theorems, and observations to make the notation above clear. As stated, these can be skipped upon first reading (also see Lecture 3 for more extensive treatment), especially since several facts are stated without proof here.

MOMENTARY DETOUR. For any n, $\mathbb{Z}_n$ is the set of integers from 0 to $n - 1$. The multiplicative group of $\mathbb{Z}_n$, $\mathbb{Z}_n^*$, is defined as:

DEFINITION 7 $\mathbb{Z}_n^* = \{x \mid x \in \mathbb{Z}_n \wedge \gcd(x, n) = 1\}$ $\diamondsuit$

So, $\mathbb{Z}_n^*$ is the set of elements from $\mathbb{Z}_n$ that are relatively prime to n. Also note that for any n, $0 \notin \mathbb{Z}_n^*$. Also, note that for prime number p, $\mathbb{Z}_p^* = \mathbb{Z}_p - \{0\}$. This is because every number in $\mathbb{Z}_p$ except 0 is relatively prime to p.

Additionally, for any positive integer n, the set $\mathbb{Z}_n^*$ and the multiplication operator (we will often write ab to denote $a * b \bmod n$) form a group. This is because:

1. $(\forall a, b \in \mathbb{Z}_n^*)[a * b \in \mathbb{Z}_n^*]$
2. $\mathbb{Z}_n^*$ has an identity element, which is 1. This is because $(\forall a \in \mathbb{Z}_n^*)[a * 1 = 1 * a = a]$
3. $(\forall a \in \mathbb{Z}_n^*)(\exists a' \in \mathbb{Z}_n^*)$ such that $a * a' = a' * a = 1$. (I.e., every element in $\mathbb{Z}_n^*$ has an inverse.) This is because for every $a \in \mathbb{Z}_n^*$, $\gcd(a, n) = 1$. Therefore, there exist integers a', n' such that:

$$aa' + nn' = 1 \quad \Rightarrow \quad aa' = 1 + n(-n') \quad \Rightarrow \quad aa' \equiv 1 \mod n$$

(And, do keep in mind that this fact isn't always true if we were to deal with elements in $\mathbb{Z}_n$.)

Fermat's little theorem states that:

Theorem 2 (Fermat's Little Theorem) *For any prime p and $x \in \mathbb{Z}_p^*$, $x^{p-1} = 1 \mod p$*

Also, for some arbitrary $a \in \mathbb{Z}_p^*$, the smallest x where $a^x = 1 \mod p$ is referred to as the order of a in $\mathbb{Z}_p^*$. (And, there may be elements in $\mathbb{Z}_p^*$ with order less than $p - 1$. For example, if p is a prime larger than 3, then $(p-1)^2 = (-1)^2 = 1 \mod p$, so the order of $p - 1$ in $\mathbb{Z}_p^*$ is 2, and $2 < p - 1$ when $3 < p$)

And, also note the following number theory theorem.

Theorem 3 (Number Theory Fact) *When p is prime, $\mathbb{Z}_p^*$ has at least one element g with order $p - 1$.*

And, elements in $\mathbb{Z}_p^*$ with order $(p - 1)$ are commonly referred to as primitive elements or *generators*.[3] Notice that $\{g^1, g^2, ..., g^{p-1}\} = \mathbb{Z}_p^*$. So, raising g to powers ranging from 1 to $p - 1$ (or 0 to $p - 2$, since $g^{p-1} = 1 = g^0 \mod p$) "generates" all the elements in $\mathbb{Z}_p^*$.

COMING BACK TO OWPS. As a result of all these condensed definitions, we can revisit our function $f_{p,g}$ where $f_{p,g}(x) = g^x \mod p$. Here p is a k-bit prime, g is a generator of the set $\mathbb{Z}_p^*$, and $x \in \mathbb{Z}_p - \{0\}$ is the actual input.

We now justify why we believe that $f_{p,g}$ is a OWP. First, since g is a generator, our function could be viewed as a permutation from $\mathbb{Z}_p - \{0\} = \mathbb{Z}_p^*$ to $\mathbb{Z}_p^*$. Second, we claim that computing $y = g^x \bmod p$ could be done in polynomial time as follows. Assuming p has k bits, for any arbitrary $x \in \mathbb{Z}_p^*$, we can find $f_{p,g}(x)$ as follows:

[3] The origin of the term "primitive element" is puzzling. For example, it is certainly non-trivial to show that primitive elements exist in $\mathbb{Z}_p^*$, as stated by the above theorem.

1. Compute $g^{2^i} \mod p$ for every value of i from 0 to $\log_2(p)$, which involves repeated squaring, and takes $\Theta(k)$ multiplications.

2. Look at the binary expansion of x, which might look like: $10011\ldots$, and note that $x = 2^{k-1}b_{k-1} + \ldots + 2^1 b_1 + 2^0 b_0$ where each b_i represents a binary digit in x.

3. By plug-in, since

$$g^x = g^{2^{k-1}b_{k-1}+\ldots+2^1 b_1+2^0 b_0} = g^{2^{k-1}b_{k-1}} \cdots g^{2^1 b_1} g^{2^0 b_0} = (g^{2^{k-1}})^{b_{k-1}} \cdots (g^{2^1})^{b_1} (g^{2^0})^{b_0}$$

and each $g^{2^i} \mod p$ has already been found, and each b_i is either 0 or 1, then each $(g^{2^i})^{b_i}$ term modulo p has a known value (either g^{2^i} or 1), and computing the product of all the $(g^{2^i})^{b_i}$ terms modulo p to give g^x is doable in $O(k)$ multiplications.

Overall, we get $O(k^3)$ algorithm. Therefore, $f_{p,g}(x)$ is easy to compute. The inversion problem corresponds to finding x s.t. $g^x = y \bmod p$, when given g, p, y as inputs. This is known as the Discrete-Log Problem which is believed to be very hard. Therefore, $f_{p,g}$ is believed to be hard to invert. Because of $f_{p,g}$ is easy to compute, believed to be hard to invert, and definitely is a permutation, $f_{p,g}$ makes a good candidate OWP. Unfortunately, there's no known trapdoor information that can make inverting f easy (which would make it a TDP).

Conjecture 4 *Modular exponentiation is a* OWP.

Similar to factoring, this conjecture has stood up to attempts to disprove it for a long time, and forms one of the most common assumptions in cryptography. Usually, when we will use the above conjecture as an *assumption* to prove some result X, we will say

"If *discrete log is hard*, then X is *provably* true."

6.3 TDP Candidate: RSA

An RSA function is defined as $f(x, n, e) = x^e \bmod n$. As before, we write for convenience $f_{n,e}(x) = x^e \mod n$, where n is the product of two primes p and q, $x \in \mathbb{Z}_n^*$, $e \in \mathbb{Z}_{\varphi(n)}^*$. Now for a bit more number theory. (The fun never stops!)

DEFINITION 8 [Euler phi-function] For any positive integer m, $\varphi(m)$ is the number of positive integers less than m that are relatively prime to m. ♢

As you might have guessed, for any positive integer m, the number of elements in the set $\mathbb{Z}_m^*$ is $\varphi(m)$. For any prime p, $\varphi(p) = p - 1$, since there are $p - 1$ positive integers less than p, and they're all relatively prime to p. Additionally, if $n = pq$ where p and q are primes, then, $\varphi(n) = (p-1) * (q-1) = n - (p + q - 1)$.

Now for Euler's Theorem, which is more general than Fermat's Little Theorem (mentioned earlier).

Theorem 5 *For any positive integer m and any $a \in \mathbb{Z}_m^*$, $a^{\varphi(m)} = 1 \bmod m$.*

Now note that for our RSA function f, we have $e \in \mathbb{Z}^*_{\varphi(n)}$. This is to make sure that e is relatively prime to $\varphi(n)$, so that there exists a $d \in \mathbb{Z}^*_{\varphi(n)}$ such that $ed = 1 \bmod \varphi(n)$, so that for our $f_{n,e}(x) = x^e \mod n$, we can get x back by doing

$$(x^e)^d = x^{ed} = x^{ed \bmod \varphi(n)} = x^1 = x \bmod n$$

And, assuming $c = f_{n,e}(x) = x^e \mod n$ for some value x, and d is the inverse of e modulo $\varphi(n)$ for the rest of this section, note that then since $x \in \mathbb{Z}^*_n$, and $\mathbb{Z}^*_n$ with the multiplication operator is a group (for reasons mentioned earlier), then x raised to any power is also an element of $\mathbb{Z}^*_n$. Therefore, $c \in \mathbb{Z}^*_n$. And so, $f_{n,e} : \mathbb{Z}^*_n \to \mathbb{Z}^*_n$, so f is a permutation function.

And in our RSA function f and values x and c, we assume that n, e, and the method for obtaining $f_{n,e}(x')$ for some value x' is public; but the x value that yields c, the primes p and q that make up n, and the value d that would give x from c (since $c^d = (x^e)^d = x \mod n$) are all private information.

Notice how things differ here for f in RSA compared to the f used in the modular exponentiation section (in spite the fact that they both involve modulus exponentiation). With the modular exponentiation section, the exponent is secret and the base is public, whereas in RSA, the base is secret and the exponent is public.

And, assuming n is a k-bit number, (and therefore, so is x and e), from an arbitrary x, $f_{n,e}(x) = x^e \mod n$ can be computed in time polynomial in terms of k, for reasons mentioned in the Modular Exponentiation section. Therefore, $f_{n,e}$ is easy to compute.

We also notice that the best known way to invert RSA involves factoring n into its primes (which allows one to learn $\varphi(n)$, which allows one to figure out a d, and therefore get x from c, as mentioned earlier), and this is believed to be hard to do (no PPT algorithm known for it yet). In particular, other common methods for inverting $f_{n,e}$, like finding out $\varphi(n)$ or d directly have been shown to be just as hard as factoring n. Therefore, f is believed to be hard to invert.[4]

Additionally, since we just argued that inverting RSA is easy with either the factorization on n (or the value d above), RSA is a good TDP candidate.

Conjecture 6 *RSA is a* TDP *(family).*

7 Things to Consider

After seeing all this info on OWF, OWP, and TDP. Here are three important principles in Cryptography to consider.

1. **Cryptography based on general theory**

   There are plenty of candidates for OWF, OWP, and TDP with believed hardness to invert based on differing reasons. Therefore, even if we figure out how to easily compute the prime factorization of any number (which for example will break RSA as a TDP candidate), we can still use something else as a candidate. Thus, there

[4] Even though breaking RSA is believed to be slightly easier than breaking factoring, RSA resisted many attacks and believed to be very hard to invert too.

are a lot of merits in basing cryptographic constructions on such general primitives like OWF's, OWP's or TDP's. Not only this gives us protection against breaking some specific function believed to be OWF, etc., but also allows us to distill which properties of a given function are crucial to make the construction work.

2. **Cryptography based on specific function assumptions**

   Sometimes, we can get more effective schemes if we assume certain function properties, like if $f(a) * f(b) = f(a + b)$ (which is true with the function f used in the Modular Exponentiation section). Thus, for the purposes of efficiency and simplicity, schemes based on specific functions are also extremely useful in practice. Of course, these constructions are also less general than general constructions.

3. **Specific things give rise to actual implementations**

   Finally, even implementing general constructions using specific candidates gives rise to the actual real-life systems, showing ho our general theory can be applied in practice.

In the next section we return to OWFs, OWPs and TDPs, and give a sample *general* application for these concepts.

8 Application of OWF's: Password Authentication

Assume I wish to login to a server with my password x, but I don't want my server to store x, since the server's contents might be open to the world. (An in fact, in one of my previous jobs, the server not only stored passwords, but did so in a text file that was marked public to the world... oops!)

To solve this issue, I compute $f(x) = y$ where f is OWF, and I tell the server only about y and f. (So the server doesn't store x.) When I login, I give the server z, and the server checks if $f(z) = y$. Since f is OWF, a hacker can't figure out a z such that $f(z) = y$ easily (since f being OWF implies that any algorithm A where $A(y) \to z$ and $f(z) = y$ doesn't run in PPT), so this system is secure.

However, some problems do occur:

1. How can we be sure x is uniform?

   Too many times, we pick passwords based on our login name, real name, birthday, family, friends, the new fancy word used by George W. Bush, etc. Or we pick passwords based on dictionary words, or our passwords are simply too short in length. (A hacker can easily write a program to guess all four-character password combinations until it gets a correct one.) Thus for many, x is not truly randomly chosen and might be easily found.

   Turns out, there are tools (called commitment schemes, extractors and password-atuhenticated key exchange) to somewhat overcome this problem. Except for commitments, we will not have time to talk about them in this course though.

2. Although x is not stored on the server, how can we trust that a hacker never sees x?

For example, if I connect to this server via `telnet`, Eve can run a program to identify myself, snoop onto my session, and read the text of everything I type during this session. From this, she can figure x based on what I typed. One way to overcome this is to use `ssh` (secure shell), because `ssh` encrypts the text that I type before it is being sent over the Internet. Therefore, Eve's snooping will only get her an encryption of the text I typed, and she's forced to attempt and decrypt in order to get x. Thus `ssh` is much safer than telnet, and is the reason why various companies only allow people to connect remotely via `ssh` and not `telnet`. However, there might exist ways snoop the password by other ways, so even this also isn't a perfect solution.

It turns out there are advanced techniques (called identification schemes, zero-knowledge arguments, and proofs of knowledge) which will solve this problem as well. In the next section we will present a much simpler simple way to partially solve this problem.

However, assuming a hacker can only see the server's storage y, cannot see typed password x, and assuming that x is truly randomly chosen, we immediately see that the password system mentioned earlier is secure assuming OWF's exist.

9 Application of OWP's: S/Key System

Taking the notion of password authentication a step further, let's suppose that a server keeps track of T logins you make onto it, and changes the information it stores every time you login, and what you'd have to type to authenticate yourself will also keep changing accordingly. (This would prevent a hacker from getting being able to impersonate you after snooping one of your authentications.) Here is the way to do it. Take a *random* x, and compute for each i from 0 to T, $y_i = f^{T-i}(x)$ (so $y_i = f(f \dots f(x) \dots)$) applied $(T - i)$ times) where f is OWP. Notice, being a permutation ensures that we can apply f to itself many times. We give the server only the last result y_0 (and the public function f), and nothing else. Note $y_0 = f^T(x)$.

Then, the first time we login, the server asks for y_1, and will check if $y_0 = f(y_1)$. If we correctly give it y_1 where $y_1 = f^{T-1}(x)$, then the server replaces y_0 with y_1.

The next time we login, the server will ask for y_2, and will check if $y_1 = f(y_2)$. Correctly giving y_2, where $y_2 = f^{T-2}(x)$, will make the server replace y_1 with y_2.

And so on, until the T-th login, where the server asks for y_T, and checks if $y_{T-1} = f(y_T)$. Correctly giving y_T, where $y_T = f^{T-T}(x) = f^0(x) = x$, ends the chain of T logins, and we will have to start this process over (i.e., we give the server a new set of y_0 and f and redo this).

Notice that in the process described above, we start with $y_0 = f^T(x)$. And, for each i, $y_i = f(y_{i+1})$, therefore, $y_{i+1} = f^{-1}(y_i)$, so we're inverting f once at each step to get the next correct y_j value (which the server stores for next time, and the server and hackers can't figure out any inverse until you type it in). And, we start over once we end up giving x to the server.

Notice here that after each login, the server not only changes the value it stores, but stores the value you just entered, so the server (and the hacker) is always "a step behind". While the S/Key system intuitively seems secure, let us actually proof this *formally*! We

first need to learn learn about general T-time password authentication systems... Later we will show that S/Key is indeed such a system.

DEFINITION 9 [T-time Password Authentication System] A T-time Password Authentication System is a T-period protocol between the PPT server and the PPT user. First, given the "security parameter" k, the user and the server engage in the private setup protocol (which runs in time polynomial in k), after which the user has some secret information SK, and the server has some public information PK. From now on, all the server's storage (which is initially PK) is made public, and the user and the server now engage in T login protocols. After each such protocol, the server decides whether to accept or reject the user. We require the following:

1. If the server talks to the real user (who knows SK and is honest), the server must accept that user during any of the T logins.

2. For any number of logins t, even if a hacker sees information the user uses to login to the server from his previous $(t-1)$ logins (in addition to the server's storage), the Probability that the hacker can impersonate the user's t-th login is at best $\mathsf{negl}(k)$, where k is the security parameter (here roughly size of the data the hacker needs to guess).

   More formally, any $t < T$ and any PPT A, the probability that the server accepts A at t-th login is $\mathsf{negl}(k)$. The latter probability is taken over the randomness used to setup the system, the randomness used by the user to login the first $(t-1)$ times, the possible randomness of the server to verify all the logins, and the randomness of A.

♢

It sounds like the S/Key system might be a good example of a T-time Password Authentication System. In fact it is, but only if the f used is OWP. (Just having f being OWF isn't good enough. See the homework...) Now, let's prove that if f is OWP, S/Key is a T-time Password Authentication System.

Theorem 7 *$\forall$ OWP f, S/Key is a T-time Password Authentication System.*

Proof: Assume there's some OWP f such that S/Key is not secure. We'll show that this assumption leads to the fact that f is not OWP, which is a contradiction.

Since S/Key is not secure for this f, there is some period t and some PPT A that achieve the following. Let $x \leftarrow \{0,1\}^k$ be chosen at random in the setup phase, let $y_i = f^{T-i}(x))$, so that the server stores y_0. Seeing first $(t-1)$ logings of the real user together with the server's storage gives A exactly $y_0, \ldots, y_{t-1}$. A success for A at time t means that $A(y_{t-1}, \ldots, y_0) \rightarrow y'_t$ and $f(y'_t) = y_{t-1}$. Since f is a permutation, success means that $y'_t = y_t$. To summarize, the assumption that S/key is insecure at time period t implies:

$$\Pr[\ y'_t = y_t \mid x \leftarrow \{0,1\}^k,\ y_i = f^{T-i}(x),\ \forall\ 0 \le i \le T,\ y'_t \leftarrow A(y_{t-1}, \ldots, y_0)\ ] = \epsilon \qquad (1)$$

where ϵ is non-negligible.

With this in mind, we construct a new PPT adversary $\tilde{A}$ who will invert OWP f with non-negligible probability (actually, the same ϵ). $\tilde{A}$ will be given $\tilde{y} = f(\tilde{x})$, where $\tilde{x}$ was

chosen at random from $\{0,1\}^k$. The goal of $\tilde{A}(\tilde{y})$ is to come up with $\tilde{x}$. Naturally, $\tilde{A}$ will use the hypothetical A to achive this (impossible) goal. Specifically,

$\tilde{A}$: On input $\tilde{y}$ run $z \leftarrow A(\tilde{y}, f(\tilde{y}), \ldots, f^{t-1}(\tilde{y}))$ and output z.

In other words, $\tilde{A}$ "fools" A into thinking that $\tilde{y}$ was the password y_{t-1} at period $(t-1)$, $f(\tilde{y})$ was the password $y_{t-2} = f(y_{t-1})$ at period $(t-2)$, and so on. Notice, since computation of f is poly-time and A is PPT, $\tilde{A}$ is PPT as well. Also, since f is a *permutation* and both x and $\tilde{x}$ were chosen at random, the distribution

$$D_0 = \langle f^{T-t+1}(x), \ldots, f^T(x) \rangle = \langle y_{t-1}, \ldots, y_0 \rangle$$

really expected by A in (1), is the same as the distribution

$$D_1 = \langle f(\tilde{x}), \ldots, f^t(\tilde{x}) \rangle = \langle \tilde{y}, \ldots, f^{t-1}(\tilde{y}) \rangle$$

which A received from $\tilde{A}$. Thus, our $\tilde{A}$ will succeed in finding the preimage of $\tilde{y}$ (which is necessarily $\tilde{x}$) with the same probability ϵ that A find the preimage of y_{t-1}. But this violates the definition f being hard to invert. So f is not OWP, and we get a contradiction. □

10 Application of TDP's: Weak Public-Key Encryption

This is our original motivation to study TDP's. Namely, define the following public-key "encryption" scheme. The quotes are due to the facts that the encryption achieved will only satisfy a truly minimal (and insufficient) notion of security. Still, it is a good start.

The public key PK will be the description of the TDP f itself (i.e., the public key output by the TDP key generation algortihm Gen). The secret key SK will be the corresponding trapdoor information output by Gen that makes f easy to invert. To encrypt m, Bob sends Alice $c = f(m)$. Alice decrypts c using the trapdoor SK.[5] The security of TDP's says that f is hard to invert if m is *random* in $\{0,1\}^k$ (more generally, whatever the domain of f is). Thus, the only thing we can say about this encryption is that **Eve cannot completely decrypt encryptions of random messages**. This is a very weak notion, also called "one-wayness", but for encryption!

DEFINITION 10 An encryption scheme (G, E, D) is *one-way secure*, if $\forall$ PPT Algorithm A,

$$\Pr(m' = m \mid (PK, SK) \leftarrow G(1^k);\ m \leftarrow \mathcal{M};\ c \leftarrow E_{PK}(m);\ m' \leftarrow A(c, PK, 1^k)) \leq \mathsf{negl}(k)$$

♢

Now, it is trivial to see that

Lemma 1 *If f comes from a* TDP *family, then the above encryption scheme is one-way.*

Unfortunately, one-wayness is only a very weak security notion for encryption. For example, one-wayness does not ruled out the following:

[5] More formally, we can define encryption scheme (G, E, D) using TDP collection (Gen, Eval, Sample, Invert) by letting $G =$ Gen, $E_{PK}(m) =$ Eval(m, PK) (here m belongs to the domain of the TDP, which we assume is $\{0,1\}^k$ for simplicity), $D_{SK}(c) =$ Invert(c, SK).

- Maybe Eve can get most of the message m (but not all of it). Say, Eve might get half of the message. To see that this threat is actually possible, take any great TDP f'. Define $f(x_1, x_2) = (f'(x_1), x_2)$, where $|x_1| = |x_2| = k/2$. It is very easy to see that f is a TDP such that the "encryption" of $m = (x_1, x_2)$ reveals half of the bits of m (namely x_2). This is a contrived example, but even for "natural" f's (like RSA) it turns out we can get some information about m from $f(m)$. In fact, one such "information" is the value $f(m)$ itself!

- Nothing is said if m is not random. For example, if an army base uses encryption to communicate with a mobile unit, and the only two messages the base will tell the unit is "attack" or "retreat," then the enemy unit can compute the values for $f(m)$ when m is "attack" or "retreat," and based on these values, figure out m from the ciphertext c.

Thus, this encryption leaves much to be desired, but is a non-trivial start.

11 Criticisms against OWF, OWP, TDP

Motivated by the above and the previous example, we can put forward the following criticism to the notions of OWF, OWP, and TDP:

1. When input x is not random, how can we be sure the system is secure?

   See the "attack"/"retreat" example above for the demonstration. (I.e., if x can only be a few values, we can compute the f-map for all these few values, and use this to learn x). It turns out that this issue can be solved. Essentially, we will *design* our application so that x is always chosen at random! We will see how this is possible on later examples.

2. Viewed an as "encryption", the function f could reveal a lot of partial information about x. See the pathological example of $f(x_1, x_2) = (f'(x_1), x_2)$ above. More realistically, take the Modular Exponentiation candidate example earlier, where $f_{p,g}(x) = g^x \mod p = y$. It turns out that from y, the last bit of x can be efficiently extracted, despite the hardness to extract the entire x. (I.e., we can determine if x is even or odd). A proof of this will be shown on the next lecture.

12 Ways out of the Criticism against OWF, OWP, and TDP

But alas, there are few twists we can try in order to avoid the criticisms mentioned above...

1. One way is to design an encryption function $f(x)$ hide all info about x. Unfortunately, this is exactly our goal of designing secure encryption! Thus, we came back to where we started. A new idea is needed to break out! Notice, however, that it is clear that no *deterministic* f can achieve this goal (see the "attack"/"retreat" example again; more trivially, $f(x)$ is "information" about x). Thus, we know that such f must be *probabilistic*!

2. If x is only a few values, we can create a function $g(x) = f(x,t) = z$ where t is a counter increased by 1 each time a new message is sent. The German ENIGMA machine during World War II used a technique like this. Fortunately, this had bad consequences for the Germans.[6] Unfortunately, while useful in practice, this technique lacks formal justification, at least in this simple way. Indeed, if x is not random, (x,z) is not random as well, so we still cannot use our definitions. More importantly, $f(x,z)$ might still allow one to recover most (if not all) of the bits of x.

3. How about requiring $f(x)$ to *"completely hide"* information about some function $h(x)$. In other words, standard definition tells us that $f(x)$ does not allow the hacker to get x completely, but may allow to get a lot of partial information about x. So maybe we can pin-point some partial information $h(x)$ (i.e., whether x is even or odd) which still remains completely hidden from the adversary who knows $f(x)$. In other words, $f(x)$ does not allow the adversary to learn *anything* about $h(x)$. Put yet differently, **use $f(x)$ to "encrypt" $h(x)$!!!** We will see, this is exactly out golden way out...

[6] This shows that sometimes cryptographic ignorance could be of use to the humanity. Hopefully, this argument is outdated by now: ignorance should never be good!