Computer Science 2426F — Fall, 2018
St. George Campus — University of Toronto

Notes #8 (for Lectures 11, 12)

We have already seen how two parties A and B can have a secure session over a totally insecure channel, given that they initially share a random key. One way they might share that key is by physically getting together, or through a trusted courier. But a more interesting and more efficient way is through some pre-existing infrastructure. One possibility is that they, or a larger group of friends, share a master key; whenever two people – or processes spawned by those people – want to have a session, they use that master key in a *session key exchange protocol* to share a session key.

An alternative infrastructure is a public key infrastructure (PKI), where each party has a public key that everyone knows, and a private key that only he knows; these master keys are then used to do a session key exchange protocol. Usually (parts of) these master keys will be used in the protocol to do signatures; often they will also be used to invoke a *public key encryption primitive.* (Key exchange is the main use of public key encryption primitives.) All of this (and more) is included under the subject of public key cryptography.

We will now discuss how to use number-theoretic conjectures to do public key cryptography. (It is interesting to note that one-way functions – that is, in effect, pseudo-random generators – do not appear to be sufficient for these tasks.) The two main types of conjectures typically used are the difficulty of integer factorization and the difficulty of (some version of) discrete-log. More recently, lattice-based algorithms have been studied, and these are becoming more practical. Integer factorization conjectures are used mainly by RSA related cryptosystems, and will not be discussed further here. There are many settings in which discrete-log has been studied, and most recently a lot of attention has been paid to the "elliptic curve" setting, but we will only discuss here the setting of discrete-log with respect to multiplication mod a prime.

Before doing this, we have to discuss how to do efficient modular exponentiation.
Say that x, y, z are (at most) n bit nonnegative integers, z is positive, and we want to compute $x^y \bmod z$ in time polynomial in n. The standard algorithm uses "repeated squaring". We begin by computing $x_0, x_1, \cdots, x_{n-1}$ where $x_0 \leftarrow x \bmod z$, and $x_{i+1} \leftarrow (x_i)^2 \bmod z$. Note that for each i, $x_i = x^{2^i} \bmod z$. Now say that $y = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_k}$ where $0 \leq e_1 < e_2 < \cdots < e_k < n$. So it merely remains to compute $x^y \bmod z = (x_{e_1} \cdot x_{e_2} \cdot \ldots \cdot x_{e_k}) \bmod z$.

We will now begin to discuss the discrete log problem (mod a prime). Let p be a prime number, and let n be the number of bits in p, that is, $2^{n-1} < p < 2^n$. We define the set $\mathbb{Z}_p^* = \{1, 2, \cdots, p-1\}$. This set forms a cyclic group under the operation of multiplication mod p. If you are not familiar with this terminology, all that's important is the following:

- $\mathbb{Z}_p^*$ is closed under the operation of multiplication mod p.
- There are numbers $g \in \mathbb{Z}_p^*$ called *generators*, with the property that $\{g^0 \bmod p, g^1 \bmod p, \cdots, g^{p-2} \bmod p\} = \mathbb{Z}_p^*$.

If g is a generator of $\mathbb{Z}_p^*$, the value of $g^x \bmod p$ only depends on the value of $x \bmod p-1$. The discrete log problem with respect to a prime p and generator g is to determine the value of x, when one is given $g^x \bmod p$ for a random $x \in \{0, 1, \cdots, p-2\}$. This problem can be done (even in the worst case) in polynomial time, for the special case where the values of all the prime factors of $p-1$ are small, that is, have values (not length) polynomial in n. More generally, if q^e is a prime-power factor of $p-1$, then it is easy to find the value of $x \bmod q^e$ (given $g^x \bmod p$) in time $\sqrt{q}$ times a polynomial in n; by "Chinese remaindering", this allows us to find the value of x in time $\sqrt{q}$ times a polynomial in n where q is the largest prime factor of $p-1$. For this reason, in order for the discrete-log problem mod p to be hard, it better be the case that $p-1$ has at least one large prime factor. Often one chooses p such that $p-1 = 2q$ where q is a prime.

For general p (that is, where $p-1$ has a large prime factor), the best discrete-log algorithm known has the following characteristics: is it probabilistic, and it appears (although we can't prove it) to work in time $2^{c\sqrt[3]{n \log^2 n}}$.

(In the elliptic curve setting, the best discrete-log algorithm known runs in time $\sqrt{q}$ times a polynomial in n, where n is the size (in bits) of the problem and q is the largest prime factor of the number of elements in the relevant group. This is the reason why many people like to use the elliptic curve setting, in practice. It allows for more efficient algorithms, and for shorter keys when doing public key cryptography and shorter signatures when doing signature schemes.)

It is important to realize that for the discrete log problem, there is not much difference between doing well for many $x \in \{0, 1, \cdots, p-2\}$, and doing well for *all* x. The reason is as follows. Say that we had an algorithm or circuit ALG that solved the discrete log problem (with respect to p and g) for a fraction $\geq \frac{1}{n^c}$ of the $x \in \{0, 1, \cdots, p-2\}$. We can then solve for *any* x as follows: Given $g^x \bmod p$, for a large but polynomial number of random $r \in \{0, 1, \cdots, p-2\}$, run ALG on $(g^x \cdot g^r) \bmod p$ (note that if r is random, this is a random input to ALG); with high probability at least one of these runs will succeed in finding the discrete log of its input (and we can always know when we are successful by computing modular exponentiation); when for a particular r we succeed at finding the discrete log y, we need only compute $x = (y - r) \bmod (p-1)$.

In practice, we can find n bit primes p and generators g such that it is reasonable to conjecture that the discrete log problem is hard. One way of doing this is as follows. We will use the fact that we have an efficient algorithm for testing primality. (Until recently, only probabilistic algorithms were known, but in fact we now have deterministic, polynomial time algorithms.) We begin by choosing random $n-1$ bit numbers q until we find a prime; we then test if $2q+1$ is prime; we continue in this way until we find a prime q such that $2q+1$ is also prime, and we let $p = 2q+1$. Note that this guarantees that $p-1$ has a large prime factor. Although there is no theorem of number theory that guarantees that this process is likely to halt quickly, it is conjectured that this is the case. (We DO know that there are many n-bit primes q, but we are unable to prove that for many of these, $2q+1$ is prime.)

Also, since we know the prime factorization of $p-1$, we can find a generator of $\mathbb{Z}_p^*$ as follows. We first observe that we can test if g is a generator, by checking that $g^2 \bmod p \neq 1$ and $g^q \bmod p \neq 1$. We can then find a generator by choosing random $g \in \mathbb{Z}_p^*$ until we find one that is a generator; there is a simple proof that this process works, and is likely to halt quickly. Note that it doesn't matter what generator g we choose: if we can find a discrete log with respect to a particular generator g, then we can find the discrete log with respect to any other generator g' by multiplying (mod p-1) by the discrete log of g with respect to g' (which we can assume, nonuniformly, exists).

If you do not believe that AES type constructions or subset-sum type constructions yield pseudo-random generators and one-way functions, but you do believe the assumption that discrete log is hard for the primes generated above, then this assumption can be used to construct pseudo-random generators and one-way functions. (The constructions are much less efficient than those of AES or subset-sum.) In fact, we can define a function f that is one-way if this assumption is true. The input to f will be strings r and s; r is treated as random bits used to generate an n bit prime p and generator g, as described above; s will be used to generate a (nearly) random $x \in \{0, 1, \ldots, p-2\}$; we define $f(rs) = [r\ ,\ g^x \bmod p]$.

It is extremely awkward to view in this way the input to f as containing random bits used to find a prime, and in practice we would just view the primes and their generators as pre-existing, to be used by both good guys and adversaries. For this reason, we introduce the following setting.

Setting for Discrete Log Conjectures: Assume we have, for every n, a pair (p_n, g_n) where p_n is an n bit prime and g_n is a generator of $\mathbb{Z}^*_{p_n}$; we also assume $2^{n-1} < p_n < 2^n$ and $p_n - 1 = 2q$ where q is prime.

We will eventually state a sequence of three, progressively stronger, discrete-log assumptions. The simplest and weakest is the following. For convenience, we will use the nonuniform adversary setting.

(basic) Discrete Log Assumption (with respect to the Setting):
Let $\{D_n\}$ be a polynomial size family of circuits where D_n has n input bits and n output bits.
Let $prob_D(n)$ be the probability that if x is randomly chosen from $\{0, 1, \ldots, p-2\}$ and
D_n is given $g_n^x \bmod p_n$, then D_n outputs x.
Then $prob_D(n) \leq \frac{1}{n^c}$ for each c and sufficiently large n.

The (basic) discrete log assumption appears not to be strong enough to allow us to do secure public key cryptography, but it is strong enough to give us one-way functions and pseudo-random generators. In fact, it directly gives us one-way permutations.

To see this, define the function $f_n : \{1, 2, \ldots, p_n - 1\} \to \{1, 2, \ldots, p_n - 1\}$ by $f_n(x) = g_n^x \bmod p_n$. Note that instead of viewing the exponent as ranging over $\{0, 1, \ldots, p-2\}$, we are (equivalently) viewing it as ranging over $\{1, 2, \ldots, p-1\}$. f_n is clearly one-one, onto. The discrete log assumption tells us that f_n is hard, on the average to invert. We can use ideas as in Notes #6 to create hard-core predicates and then pseudo-random number generators. However, it turns out that the assumption allows us to prove a particular bit is hard-core.

Define $B_n : \{1, 2, \ldots, p_n - 1\} \to \{0, 1\}$ by $B_n(x) = 1$ if $x > p_n/2$, and $B_n(x) = 0$ if $x < p_n/2$. That is, $B_n(x)$ is the "high order bit" of x, where we understand "high order bit" in the natural, balanced sense. From the assumption, it can be proven that B_n is hard-core for f_n. This gives us a way of constructing a pseudo-random number generator.

For $s \in \{1, 2, \ldots, p-1\}$ consider the function that maps s to $[f_n(s), B(s)]$.
If s is chosen randomly, then $f_n(s)$ is a random member of $\{1, 2, \ldots, p-1\}$; and if a polynomial time adversary is given $f_n(s)$, $B_n(s)$ will be indistinguishable from a random bit, precisely because B_n is hard-core for f_n. So in a certain technical sense, this function satisfies "unpredictability"; the only problem is that the first n bits of output look like a random member of $\{1, 2, \ldots, p-1\}$ rather than as a random string.

However, let us do the following.
For $s^0 \in \{1, 2, \ldots, p_n - 1\}$, define $s^1, s^2, \ldots$ by $s^{i+1} = f_n(s^i)$. Then define
$G_n(s^0) = B_n(s^0), B_n(s^1), B_n(s^2), \ldots$.
The ideas we discussed earlier imply that this is a pseudo-random number generator, assuming the key is chosen randomly from $\{1, 2, \ldots, p-1\}$. This is useful for most applications of pseudo-random number generators, but because the seed is not a random string, it doesn't quite allow us

to perform the tree-construction for pseudo-random function generators. However, there are simple modifications (exercise!) that will allow us to do this tree-construction.

It is interesting to note that we can also prove that the assumption implies the following: Given $f_n(x)$ for random $x \in \{1, 2, \ldots, p-1\}$, the string consisting of the $\log n$ "high-order bits" of x is indistinguishable from a random string (when we define "high-order bits" in a natural, balanced sense). This fact can be used to construct more efficient pseudo-random generators than that described above.

As we stated above, the (basic) discrete log assumption doesn't appear to be strong enough to allow us to do public key cryptography. We therefore introduce two stronger assumptions.

The Diffie-Hellman assumption says that from the values of g_n^x and g_n^y mod p_n, it is hard to compute g_n^{xy} mod p_n. Clearly this implies the basic assumption, since if we could find x and y, we could compute xy. We can use the Diffie-Hellman assumption to do secure key exchange, but we prefer to use the following stronger assumption.

The *Decisional* Diffie-Hellman assumption says that from the values of g_n^x and g_n^y mod p_n, not only is it hard to compute g_n^{xy} mod p_n, it is even hard to distinguish between g_n^{xy} mod p_n and a random member u of $\{1, 2, \ldots, p-1\}$. Actually, this is not exactly true. Recall that since 2 divides $p_n - 1$, it is easy to compute the values of x and y mod 2, and hence the value of xy mod 2. 2 is the only small divisor of $p_n - 1$, and the assumption states that this is the only way that g_n^{xy} mod p_n can be distinguished from u.

The Decisional Diffie-Hellman assumption can be used to do secure key-exchange in a more efficient way. Also, it can be used to construct a reasonably efficient, strongly secure public key encryption primitive (a fascinating theorem of Cramer and Shoup). We will define these notions of security later.

Diffie-Hellman Discrete Log Assumption (with respect to the Setting):
Let $\{D_n\}$ be a polynomial size family of circuits where D_n has $2n$ input bits and n output bits.
Let $prob_D(n)$ be the probability that if x and y are randomly chosen from $\{0, 1, \ldots, p-2\}$ and D_n is given g_n^x mod p_n, and g_n^y mod p_n, then D_n outputs g_n^{xy} mod p_n.
Then $prob_D(n) \leq \frac{1}{n^c}$ for each c and sufficiently large n.

Decisional Diffie-Hellman Discrete Log Assumption (with respect to the Setting):
(Recall that 2 is the only small factor of $p - 1$.)
Let $\{D_n\}$ be a polynomial size family of circuits where D_n has $3n$ input bits and 1 output bit.
Let $prob_D(n)$ be the probability that if x and y are randomly chosen from $\{0, 1, \ldots, p-2\}$, and z is randomly from $\{0, 1, \ldots, p-2\}$ such that z mod $2 = (xy)$ mod 2, and
u_0 is assigned g_n^{xy} mod p_n, and u_1 is assigned g_n^z mod p_n, and b is randomly chosen from $\{0, 1\}$ and D_n is given g_n^x mod p_n, and g_n^y mod p_n, and u_b, then D_n outputs b.
Then $prob_D(n) \leq \frac{1}{2} + \frac{1}{n^c}$ for each c and sufficiently large n.

Naive Key Exchange:
Here's one naive way we can use the Decisional Diffie-Hellman assumption to do key exchange. Say that A and B want to exchange a session key, where both know security parameter 1^n. Let $p = p_n$ and $g = g_n$.

A chooses a random $x \in \{0, 1, 2, \ldots, p-2\}$ and sends g^x mod p to B.
B chooses a random $y \in \{0, 1, 2, \ldots, p-2\}$ and sends g^y mod p to A.
At this point, both A and B can compute g^{xy} mod p:
A by computing $(g^y \bmod p)^x$ mod p and B by computing $(g^x \bmod p)^y$ mod p.

Now A and B can use $g^{xy} \bmod p$ as their exchanged key, or they can use some string derived from $g^{xy} \bmod p$. For example, they can use $h(g^{xy} \bmod p)$ where h is some appropriate, publicly known function mapping n bits to, say, $\lfloor n/2 \rfloor$ bits. If h is chosen randomly from a family satisfying "pair-wise independence", then the Decisional Diffie-Hellman assumption will imply that $h(g^{xy} \bmod p)$ looks – to someone who only knows $g^x \bmod p$, $g^y \bmod p$ and h – like a random $\lfloor n/2 \rfloor$ bit string.

However, this is not a very secure key exchange protocol. An important problem is that an adversary can pretend to B that he is A (or vice versa); he can then share a key with B, at which point B may use the key to tell the adversary something intended for A. (In fact, this problem is impossible to avoid if we do not have "setup" such as a public key infrastructure.) We will define secure key exchange later, but if a key exchange is secure, an adversary should not be able to learn any session key (or learn anything about any session key) inappropriately.

Here is one naive way we can use the Decisional Diffie-Hellman assumption to construct a public key encryption primitive. It is similar to the above naive key exchange.

B's private key is a randomly chosen $y \in \{0, 1, 2, \ldots, p-2\}$, and his public key is $g^y \bmod p$.

To encrypt an $\lfloor n/2 \rfloor$ bit message m to B, A chooses a random $x \in \{0, 1, 2, \ldots, p-2\}$, computes $g^x \bmod p$ and $g^{xy} \bmod p$, and sends to B the pair $(g^x \bmod p, m \oplus h(g^{xy} \bmod p))$.

B decrypts in the obvious way.

We will see later that this satisfies a weak kind of security (semantic security) but not the strong kind of security we usually require when public encryption keys become part of the public key infrastructure (strong chosen-cipher-text security).

We saw above that it is convenient to assume that our Setting also includes a "hash" function h such that for all n, $h : \{0,1\}^n \to \{0,1\}^{\lfloor n/2 \rfloor}$.

DDHH – Decisional Diffie-Hellman Discrete Log Assumption (with respect to the Setting with hash function h):
Let $\{D_n\}$ be a polynomial size family of circuits where D_n has $2n + \lfloor n/2 \rfloor$ input bits and 1 output bit.
Let $prob_D(n)$ be the probability that if x and y are randomly chosen from $\{0, 1, \ldots, p-2\}$, and u_0 is assigned $h(g_n^{xy} \bmod p_n)$, and u_1 is assigned a random string from $\{0,1\}^{\lfloor n/2 \rfloor}$, and b is randomly chosen from $\{0, 1\}$ and D_n is given $g_n^x \bmod p_n$, and $g_n^y \bmod p_n$, and u_b, then D_n outputs b.
Then $prob_D(n) \leq \frac{1}{2} + \frac{1}{n^c}$ for each c and sufficiently large n.

We now wish to give an improved – although still flawed – key exchange protocol. For this protocol we will assume a PKI where everyone has keys for a secure signature scheme. That is, every person has a private key for signing messages and a public key that anyone can use to verify a message signed by him.

In our model, a person, say A spawns a process for talking to another person, say B. This process will also have a "role" bit b associated with it; we call this an A-process (because it was spawned by A) and say that it is a process of type $\langle A, B, b \rangle$. The role bit is used to give an asymmetry to the two parties participating in a key exchange protocol. One reason this is necessary is because some exchange protocols require the two parties to behave differently. However, even if both parties behave the same way in the key exchange protocol, they will want to behave asymmetrically when they later use the exchanged key, so this role bit is necessary. For example: we can use the exchanged key K to pseudo-randomly generate two keys K_0 and K_1, where K_0 is to be used as the shared key for the role-0 person to talk to the role-1 person, and K_1 is to be used as the shared key for the role-1 person to talk to the role-0 person.

So the $\langle A, B, b\rangle$ process hopes it is exchanging a session key with a $\langle B, A, 1-b\rangle$ process. What caused these two processes to be set up in the first place? We assume this was done somehow using the magic of the internet as a result of an insecure conversation between (supposedly) A and B.

We will now describe **Protocol 1**, which will be described in more detail in the next notes. Process $\langle A, B, b\rangle$ works as follows on security parameter n. We let $p = p_n$, $g = g_n$.

- The process chooses a random $x \in \{0, 1, \ldots, p-2\}$, computes $\alpha = g^x \bmod p$ and sends α together with a signature for $[\alpha, b]$, to (hopefully) the B process.
- The process receives a string $\beta = g^y \bmod p$ together with a what is supposed to be a signature by B of $[\beta, 1-b]$.
- If the signature check fails, then the process outputs FAIL;
  otherwise, it outputs $h(\beta^x \bmod p)$.
  (Note that $\beta^x \bmod p = g^{xy} \bmod p$.)

This protocol is insecure for a more subtle reason than the previous one was. To understand this it helps to have some idea of what the adversary is trying to do. The world consists of "good guys" and "bad guys" where the bad guys are really just pseudonyms for the adversary. The bad guys can choose their public keys any way they like (although this feature will not play a role in the breaking of this protocol).

The adversary is trying to learn something about the key K output by a $\langle A, B, b\rangle$ process where both A and B are good guys. Therefore this key K should look random and unrelated to all the other keys, except for the (possibly) one key output by a $\langle B, A, 1-b\rangle$ process that is identical to it. So we say (informally) that the adversary should not be able to learn anything about K, even when he is allowed to see all the keys that are output except for one from a $\langle B, A, 1-b\rangle$ process that is identical to K.

Here is how the adversary will break an $\langle A, B, 0\rangle$ process, where A and B are good guys. He will use a pseudonym U for himself; he will choose the public/private key for U properly so that he can sign anything he wants on behalf of U. "U" will cause to be set up a B process $\langle B, U, 1\rangle$.

The adversary will see the string $\alpha = g^x \bmod p$ sent out by the $\langle A, B, 0\rangle$ process, ignoring the signature by A.

He will then send α, together with a signature by U of $[\alpha, 0]$, to the $\langle B, U, 1\rangle$ process.

He will then read from the $\langle B, U, 1\rangle$ process the string $\beta = g^y \bmod p$ together with the signature σ by B of $[\beta, 1-b]$.

He will then send β and σ to the $\langle A, B, 0\rangle$ process.

At this point both processes are happy with the signatures they have seen, and they will both output the same key K. The adversary is allowed to look at the key output by the $\langle B, U, 1\rangle$ process since it is not supposed to match the key output by the $\langle A, B, 0\rangle$ process. Once he does this, he knows something about (in fact everything about) the key output by the $\langle A, B, 0\rangle$ process.

A bit of philosophy here. Why do we allow the adversary to look at the key output by the $\langle B, U, 1\rangle$ process? Because in the real world he may know something about the conversation taking place using that key, and therefore learn something about the key. Why do we allow the adversary to look at the key output by a $\langle B, A, 0\rangle$ process or a $\langle A, B, b\rangle$ process? Because if one of these keys related to a $\langle A, B, 0\rangle$ key, it could cause miscommunication between A and B, and/or it could make the communication between them insecure. Why do we allow the adversary to look at the key output by a $\langle B, A, 1\rangle$ process if it is not the same as the $\langle A, B, 0\rangle$ key he is interested in? Because if the keys are related but different, it could cause miscommunication between A and B, and/or it could make the communication between them insecure. Of course, if the key output by a $\langle B, A, 1\rangle$ process is exactly the same as the $\langle A, B, 0\rangle$ key he is interested in, then the adversary is not allowed to look at it. We will make this definition precise in the next set of notes.

One more remark. We have said that the adversary wins if he can make certain keys too predictable given certain others. Should we not say that he also wins if he can make certain keys that should be identical, to be instead random and unrelated? No! This is a trivial task for an adversary: all he has to do is take a $\langle A, B, 0\rangle$ process that has shared a key with a $\langle B, A, 1\rangle$ process, and another $\langle A, B, 0\rangle$ process that has shared a key with another $\langle B, A, 1\rangle$ process, and "cross-wire" them!

One final remark. A and B are not really "people" (whatever that means). A and B are names (strings really) that appear in a public key infrastructure table. Although, as we will see below, a bad guy might choose his public key to be the same as some good guy's public key, we require the names in the table to be distinct.