15-503/859P: Cryptography **Lecturer:** Manuel Blum
Topic: Hard Problems and Apps **Date:** 15 Mar 2006
Scribe: Yinmeng Zhang

15.1 Administrivia

MIDTERM EXAM Wednesday after Spring Break (Mar 22)

15.2 Generators

When does $\mathbb{Z}_n^*$ have a generator? We have claimed in class that $\mathbb{Z}_p^*$ has a generator for any prime p. As it turns out, $\mathbb{Z}_{p^e}^*$ also has a generator if p is an odd prime. For example,

$$Z_{3^2}^* = \{1, 2, 4, 5, 7, 8\}$$

is generated by 2. $\mathbb{Z}_{2p^e}$ has a generator for odd p, but Z_{2^e} has generator only for $e = 1, 2$. For example,

$$\mathbb{Z}_8^* = \{1, 3, 5, 7\}$$

has no generator! (check by cases). We will come back to this.

15.3 Hard Problems

15.3.1 Discrete Log Problem

INPUT: prime p, generator g of $\mathbb{Z}_p^*$, $a \in \mathbb{Z}_p^*$

OUTPUT: x s.t. $g^x \equiv a \pmod{p}$

15.3.2 Principal Square Root Problem

Recall $\mathbb{Z}_p^* = g, g^2, \ldots, g^{(p-1)/2}, g^{(p-1)/2+1} \ldots$ up to (p-1)/2+1 (the lower half) are the principal square roots; the remainder are nonprinciple.

INPUT: prime p, generator g of $\mathbb{Z}_p^*$, a square $a^2 \pmod{p}$

OUTPUT: YES iff a is the *principal square root* of a^2

We could even include a and $-a$ in the input and the problem would still be hard (we can't determine which one is the smaller power of g).

Notce the principal square root problem depends on the generator given in the input; you can check that $\mathbb{Z}_{11}^*$ is generated by both 2 and 7. For generator 2, all the principal square roots are $2, 4, 8, 5, 10$, while for 7, the principal square roots are $7, 5, 2, 3, 10$.

15.3.3 Equivalence

Recall that the two problems are polynomial time equivalent. Intuitively, this means that if you can solve one in polynomial time, then you can solve the other in polynomial time. This is not quite correct, since if you *can't* solve the one, it implies nothing about the other. The correct statement is

> Given a blackbox solver for the first problem, you can construct a polynomial time solver for the second problem.

Notice this does not reference whether or not the blackbox solver exists.

Last time we showed how to solve the discrete log problem given an oracle for the principal square root problem; the other direction also holds.

15.4 Coin Tossing Into a Well

Suppose Alice and Bob have just divorced, and they want to decide who gets the house. They have agreed to "flip a coin" for it, but since they are not willing to be in each others' physical presence, they would like to run some protocol which names Alice the winner half the time and to Bob half the time. Also, since Alice and Bob are not in very trusting moods right now, they would like the protocol to allow them to check that the other party is not cheating.

To help Alice and Bob out, we will construct a *tossing a coin into the well* protocol based on the principal square root problem. Where do wells come in? Notice that tossing a coin into the well is just like commiting to a random bit - since the coin is down the well, the value can no longer be changed, but it can still be viewed.

Basically, Alice constructs an instance of the principal square root problem, and Bob guesses the answer - Bob wins iff he guesses correctly.

1. Alice chooses a prime p and picks e at random from $[(p-1)/2]$.
2. Alice sends over p, the factors of $p-1$ (call them q_1 through q_n), a generator g of $\mathbb{Z}_p^*$, the square $a^2 = g^{2e}$, and the roots $\{a = g^e, -a = -g^e\}$ in sorted order.
3. Bob checks that Alice really sent over a valid instance. That is, he checks
   (a) p and the q_i are really primes using the primality testing algorithm of his choice
   (b) $\Pi q_i = p - 1$ and for each factor q, $g^{(p-1)/q} \neq 1$, which proves g really is a generator for $\mathbb{Z}_p^*$.
   (c) the supposed roots Alice sent over really give g^{2e} when squared.
4. Bob selects a, $-a$ at random and sends it to Alice
5. If Bob sent the principal square root, then he wins, else he loses. Notice at this point only Alice knows the outcome of the coin flip.

6. Alice sends Bob e so that he can check if he won or not.

At the end of this protocol, Alice and Bob will have agreed on a bit - is this bit random?

Alice cannot cheat since we have included checks on everything she sends over in the protocol.

Bob cannot cheat because of the hardness of the principal square root problem. Suppose for contradiction Bob had an *advantage*, i.e. for given p, g, the probability over e of guessing correctly is at least

$$\frac{1}{2} + \frac{1}{\mathsf{poly}(n)}$$

(where n is the length of the prime p)

We wish to show that he can *amplify* this probability to solve the principal square root problem, which we assumed is hard.

To guess which of $\pm a$ is the principal square root of $a^2 \pmod p$, Bob repeatedly picks random $\hat{r}$, and solves the new problem of deciding which of $\pm a * a^{\hat{r}}$ is the principal square root of $a^2 a^{2\hat{r}}$.

15.5 Euler's Generalization of Fermat's Little Theorem

The Euler-phi function (also known as the totient) is defined as $\phi(n) := |\mathbb{Z}_n^*|$.

Theorem: If $n \in \mathbb{Z}^+$, then $\forall a \in \mathbb{Z}_n^*$

$$a^{\phi(n)} \equiv 1 \pmod n$$

To calculate the Euler-phi function, we use the fact if $n = \prod p_i^{e_i}$, then $\phi(n) = \prod p_i^{e_i - 1} = n \prod (1 - 1/p_i)$. This shows that, given the factorization of n, we can find $\phi(n)$ efficiently. The converse also holds.

Theorem: There is an efficient algorithm that, given composite N, $\phi(n)$, *splits* n, i.e. returns nontrivial factors a, b, such that $ab = n$.

An interesting special case of the second theorem is when $n = p_1 p_2$, the product of two unknown primes. The theorem claims that we can efficiently find p_1 and p_2. By definition $\phi(n) = (p_1 - 1)(p_2 - 1)$, so factoring n boils down to solving a system of two equations in two unknowns.

One particularly cute way to approach the system is to notice $\phi(n) = (p_1 - 1)(p_2 - 1) = n - (p_1 + p_2) + 1$, so we can derive $(x - p_1)(x - p_2) = x^2 - (N - \phi(n) + 1) + N$. The roots of the equation are exactly p_1 and p_2, so we simply solve the equation to extract the factors.

15.6 Carmichael's lambda function

We know that $a^{\phi(n)} \equiv 1 \pmod n$ for a in $\mathbb{Z}_n^*$ - is $\phi(n)$ the smallest exponent for which this holds? For primes, yes, in general, no. The smallest exponent is called Carmichael's lambda function. Given n and its factorization $\prod p_i^{e_i}$, we can calculate $\lambda(n)$ using the formula

$$\lambda(n) = \mathsf{lcm}_i[p_i^{e_i - 1}(p_i - 1)]$$

In cryptographic applications, the interesting case is usually $n = p_1 p_2$, the product of two distinct primes.

We construct p_i by repeatedly picking q_i until we find prime $p_i = 2q_i + 1$. This gives $\lambda(n) = \mathsf{lcm}[2q_1, 2q_2] = 2q_1 q_2 = \phi(n)/2$.

The upshot is that we can factor n given n and $\lambda(n)$. In fact,

Theorem:(Miller) Given N and any integer multiple of $\lambda[n]$ or $\phi(n)$ we can efficiently factor n.

15.6.1 Generators Again

Q: What is generator of $\mathbb{Z}_n^*$ where $n = p_1 p_2$, p_i distinct odd primes?

A: It has none! This is because (1) every element a of $\mathbb{Z}_n^*$ has order which divides $\lambda(n)$, and (2) $\lambda(n) < \phi(n)$.

(1) comes from the Chinese Remainder Representation. Given generators g_1 for $\mathbb{Z}_{p_1}^*$ and g_2 for $\mathbb{Z}_{p_2}^*$, we can express every number as $\langle g_1^i, g_2^j \rangle$. The order of $\langle g_1^i, g_2^j \rangle$ is the smallest k such that $\langle g_1^{ik}, g_2^{jk} \rangle = \langle 1, 1 \rangle = 1$.

By Fermat's little, every $k(p_1 - 1)$th power of $\langle g_1^i, g_2^j \rangle$ will have the form $\langle 1, y \rangle$, and every $k(p_2 - 1$th power will have the form $\langle x, 1 \rangle$. So, we want to find a number which is divisible by $p_1 - 1$ and $p_2 - 1$; $\mathsf{lcm}(p_1 - 1, p_2 - 1)$ springs to mind, which happens to be $\lambda(n)$. Now, by Lagrange, the order of the element must divide $\lambda(n)$.

(2) follows from the definitions of $\lambda(n)$ and $\phi(n)$. By construction both $p_i - 1$ are even, so their lcm is less than their product by at least a factor of 2. It so happens that their product is $\phi(n)$ and their lcm is $\lambda(n)$.

15.7 Collision-Resistant Hash Functions

Given a class of integers $\mathcal{C}$ with associated hard computation problem, for each $N \in \mathcal{C}$ we define a *collision-resistant hash function* $h_N : \mathbb{Z}^+ \to \mathbb{Z}_n^+$ to be a function with the property that from any two distinct integers (messages) $m_1, m_2 \in \mathbb{Z}^+$ s.t. $h_N(m_1) = h_N(m_2)$, one can efficiently solve the hard problem associated with N.

Here, $\mathcal{C}$ consists of products of large odd primes, and the hard computational problem is factoring.

For $N = p_1 p_2$, let g_1 be a generator for $\mathbb{Z}_{p_1}^*$, g_2 a generator for $\mathbb{Z}_{p_2}^*$. We construct a "pseudo-generator" $g = \langle g_1, g_2 \rangle$ for $\mathbb{Z}_N^*$, which we argued above has order $\lambda[N]$. Now, we define

$$h_N(M) = g^m \pmod{n}$$

This hash function is collision-resistant. Suppose we can find distinct M_1 and M_2 s.t. $h(M_1) = h(M_2)$, then $g^{M_1 - M_2} \equiv 1 \pmod{n}$. In other words, the order of g divides $M_1 - M_2$, so we would have a multiple of $\lambda(N)$! Now that we have N and $c\lambda(N)$, we can factor N.