**6.857 Computer and Network Security** September 24, 2002

# Lecture Notes 6 : Number Theory

*Lecturer: Ron Rivest* *Scribe: Burkard/Gu/Yu*

[These notes come from Fall 2001. Check with students' notes for new topics brought up in 2002.]

## Outline

- $WCR \not\Rightarrow CR$ (example)
- Digital Signatures (concepts)
- Number Theory & Primality-Testing

## 1 $WCR \not\Rightarrow CR$

It should be noted that a hash function can be weak collision resistant and not collision resistant.

**Theorem 1** *$\exists$ hash function $h$ s.t. $h \in WCR$ and $\notin CR$*

**Proof:**

Given some hash function $g \in OW \cap CR$, we construct a hash function $h$, s.t. $h \notin CR$ but $h \in WCR$.

$h$ takes pairs $(x, y)$ as input:

**1** Finds the least $i > 0$ s.t. $g^{(i)}(x)$ ends in 4 zeros

$$\overset{x = x_0}{\bullet} \xrightarrow{g} \overset{x_1}{\bullet} \xrightarrow{g} \overset{x_2}{\bullet} \xrightarrow{g} \overset{x_3}{\bullet} \xrightarrow{g} \ldots \xrightarrow{g} \overset{g^{(i)}(x)}{\bullet}$$

**2** Finds the least $j > 0$ s.t. $g^{(j)}(y)$ ends in 4 zeros

$$\overset{y = y_0}{\bullet} \xrightarrow{g} \overset{y_1}{\bullet} \xrightarrow{g} \overset{y_2}{\bullet} \xrightarrow{g} \overset{y_3}{\bullet} \xrightarrow{g} \ldots \xrightarrow{g} \overset{g^{(j)}(y)}{\bullet}$$

**3** Outputs $(g^{(i)}(x), g^{(j)}(y), i + j)$

$h \notin CR$: Choose any $x$, $y$ and create pairs $(x, g(y))$ and $(g(x), y)$ s.t. $h(x, g(y)) = h(g(x), y)$.

$h \in WCR$: Given $(x, y)$, it is infeasible to find $(x', y') \neq (x, y)$ s.t. $h((x, y)) = h((x', y'))$.

**Note:** $g \in OW$, so the $h(x, g(y)) = h(g(x), y)$ scenario used to show $\notin CR$ cannot be used to show $\notin WCR$. ■

# 2 Digital Signatures

## 2.1 Public Key Cryptography

So far our cryptographic techniques have utilized a shared secret; because of this they operate in a symmetric and point-to-point manner. Because public key cryptography breaks the symmetry and allows one-to-many and many-to-one operation, it is also called *asymmetric cryptography.*

The main idea is to separate the capability of encryption from decryption. Thus, the system has the following requirements:

- an encryption key $E_A$, such that $E_A(M) = C$
- a decryption key $D_A$, such that $D_A(E_A(M)) = M$
- knowing $E_A$ does not imply knowing $D_A$ (i.e. knowing how to encrypt does not imply knowing how to decrypt)

With such a system, $E_A$ can be published in a directory as Alice's *public key*, and people can use the key to create messages that only Alice can decrypt.

## 2.2 Digital Signatures

We can use a corresponding idea to public-key encryption for providing authentication: we separate the process of signing from verifying. This allows anyone to verify that an individual signed a message. Since this is like signing a paper document, the technique is called a *digital signature.*

Because the encryption function produces a message with at least as much information as the original message, it suffices to sign only a hash of the message. Hence it's important that a collision resistant hash function is chosen.

$$\begin{array}{ccl} \text{Alice} & & \text{Bob} \\ Sk_A & \overset{M,\ \text{sign}(Sk_A, h(M))}{\longrightarrow} & \text{verify}(M,S,Pk_a) = \begin{cases} Y \\ N \end{cases} \end{array}$$

In this example, because Alice sends a signed document to Bob, she first creates a digital signature sign$(Sk_A, h(M))$, a function that takes in Alice's secret key and a hash of message $M$. Alice then sends the message $M$, together with the digital signature sign$(Sk_A, h(M))$. Bob can verify the authenticity of the message using verify$(M,S,Pk_a)$, which hashes $M$, and compares it to the decrypted signature. If the hashes match, it outputs a $Y$. Otherwise it outputs a $N$.

Required components for digital signatures:

- keygen$(k) \longrightarrow (Sk,\ Pk)$ (randomized)

Given a $k$, keygen($k$) should be able to generate a pair of keys. The bigger the $k$, the larger the keys are, and the higher the security. The function should be random, and not deterministically generated. The generator can query the user for some random keyboard input or mouse movement, and use the user's input to seed the random generator.

- sign($Sk$, $M$) $\longrightarrow$ signature (randomized or deterministic)

  The signing function could be a deterministic, one-to-one function, but this would allow a *guess and check* attack. Therefore, sign($Sk$, $M$) should be (and is) randomized in practice.

  Note that sign($Sk$, $M$) should also be independent of hash function h($x$).

- verify($M$,$S$,$Pk$) $= Y$ iff $s$ is possible output of sign($Sk$, $M$)

# 3 Number Theory

## 3.1 Modular Arithmetic

Modular arithmetic is used frequently in public key cryptography and in digital signatures. Numbers run over only a finite space of integers, and the exponentiation of the numbers in modular arithmetics creates some interesting properties.

For any given integer $> 1$, we define their modulo space as follows:

$$\begin{array}{rl} \pmod{p} & Z_p = \{0, 1, 2, \ldots, p-1\} \\ \pmod{n} & Z_n = \{0, 1, 2, \ldots, n-1\} \end{array}$$

**Note:** $p$ is generally assumed to be prime.

For example:

$3 \cdot 5 \equiv 1 \pmod{7}$

$-1 \equiv 6 \pmod{7}$

**Note:** For any given a, b, p, if $a \equiv b \pmod{p}$, then $a - b$ is a multiple of $p$.

Under modular arithmetic $+$, $-$, $\times$ is easy, but / is somewhat tricky. In order to compute divisions under some mod($n$), we have to rely on some concepts on modular exponentiation.

## 3.2 Modular Exponentiation

To find $a/b$, we calculate $a \cdot b^{-1}$ using exponentiation. How do we compute $a^b \pmod{p}$ efficiently?

$$a^b \pmod p = \begin{cases} 1 & \text{if } b = 0 \\ (a^{b/2})^2 \pmod p & \text{if } b \text{ is even} \\ a \cdot a^{b-1} \pmod p & \text{if } b \text{ is odd} \end{cases}$$

It only takes $\leq 2\log_2 b$ modular multiplications, which is a few milliseconds on a modern computer.

**Theorem 2** *Fermat's Little Theorem: If p prime,* $\exists$ *some* $a$*,* $1 \leq a < p$*, s.t* $a^{p-1} \equiv 1 \pmod p$.

For example:

$1^6 \equiv 1 \pmod 7$

$2^6 \equiv 64 \equiv 1 \pmod 7$

$3^6 \equiv 1 \pmod 7$

### 3.3 Division

Using modular exponentiation, we can compute divisions of numbers in modular arithmetic:

$1/a \equiv ? \pmod p$

$(1/a) \cdot a \equiv 1 \pmod p$

$a^{p-2} \cdot a \equiv a^{p-1} \equiv 1 \pmod p$ (Fermat's Little Theorem)

$2^5 \equiv 4 \equiv 2^{-1} \pmod 7$

### 3.4 Prime Numbers

The ideas behind public cryptography requires prime numbers, and preferably large prime numbers. How can we find such large prime numbers?

- Pick a large random (odd) number
- Test it for primality
- Repeat until you succeed (maybe a few hundred times)

**Theorem 3** *Prime Number Theorem*

$$\begin{aligned} \pi(x) &= \textit{number of primes} \leq x \\ &\cong \frac{x}{\ln x} \end{aligned}$$

**Note:** 1 out of every $k - ln(2)$ = k-bit #'s are prime

## 3.5 Primality Test

Test if $2^{p-1} \equiv 1 \pmod{p}$

*NO* - p not prime

*YES* - guess "p is prime"

There is a significant difference between testing a random number for primality and testing an arbitrary number for primality. Random numbers are great for primality tests because the chances of picking a composite that passed the primality test is very small. However, since there exist composites that pass the primality test, testing an arbitrary number (provided by an adversary) is not that useful.

The chances of picking a 1024-bit number $p$ that is not prime, yet for which $2^{p-1} \equiv 1 \pmod{p} \leq 10^{-41}$

As long as the numbers are random, failing the primality test is not worth worrying about.

You can improve your chances to weed out composites by running the primality test in different bases. For example, if $2^{p-1} \equiv 1 \pmod{p}$, try $3^{p-1} \equiv 1 \pmod{p}$, and so on.

There is a class of numbers however, called **Carmichael numbers**, that passes the primality test for all bases.

**Fact** $a^{560} \equiv 1 \pmod{561}$ *for all* $a$*,* $1 \leq a \leq 560$*.*

There is also another way to distinguish primes from composites:

**Fact** *Primality Test #2*

*In mod* $p$*, if* $p$ *is prime, the only square roots of* $1$ *mod* $p$ *are* $\pm 1$

*In mod* $n$*, if* $n$ *is not prime, there are at least 4 square roots of* $1$*.* $\pm 1$ *and* $\pm x$*.* $\pm x$ *are the non-trivial square roots of 1*

Append check to modular exponentiation: We can append this check to the squaring $((a^{b/2})^2 \pmod{p})$ of the modular exponentiation algorithm. If result is 1, check to see if the input $\neq \pm 1$. If yes, then the number is not a prime.

**Theorem 4** *If* $p$ *is not prime and* $a$ *is chosen randomly* $(1 \leq a < p)$ *then we have* $\geq \frac{1}{2}$ *chance of discovering that* $p$ *is not prime, either because* $a^{p-1} \neq 1 \pmod{p}$ *or because we discover a non-trivial square root of* $1$