CS 161 Fall 2005 — Computer Security — Joseph/Tygar/Vazirani/Wagner — Notes 10

1 One-way function

A one-way function is a fundamental notion in cryptography. It is a function on n bits such that given x it is easy to compute $f(x)$ but on input $f(x)$ it is hard to recover x (or any other preimage of $f(x)$). One of the fundamental sources of one-way functions is the remarkable contrast between multiplication, which is fast, and factoring, for which we know only exponential time algorithms. The simplest procedures for factoring a number require an enormous effort if that number is large. Given a number N, one can try dividing it by $1,2,\ldots,N-1$ in turn, and returning all the factors that emerge. This algorithm requires $N-1$ steps. If N is in binary representation, as is customary, then its length is $n = \lceil \log_2 N \rceil$ bits, which means that the running time is proportional to 2^n, exponential in the size of the input. One clever simplification is to restrict the possible candidates to just $2,3,\ldots,\sqrt{N}$, and for each factor f found in this shortened list, to also note the corresponding factor N/f. As justification, witness that if $N = ab$ for some numbers a and b, then at most one of these numbers can be more than $\sqrt{N}$. The modified procedure requires only $\sqrt{N}$ steps, which is proportional to $2^{n/2}$ but is still exponential. Factoring is one of the most intensely studied problems by algorithmists and number theorists. The best algorithms for this problem take $2^{cn^{1/3}log^{2/3}n}$ steps. The current record is the factoring of RSA576, a 576 bit challenge by RSA Inc. The factoring of 1024 bit numbers is well beyond the capability of current algorithms.

The security of the RSA public key cryptosystem is based on this stark contrast between the hardness of factoring and multiplication.

2 Outline of RSA

In the RSA cryptosystem, each user selects a public key (N,e), where N is a product of two large primes P and Q, and e is the encryption exponent (usually $e = 3$). P and Q are unknown to the rest of the World, and are used by the owner of the key (say Alice), to compute the private key (N,d). Even though d is uniquely defined by the public key (N,e), actually recovering d from (N,e) is as hard as factoring N. i.e. given d there is an efficient algorithm to recover P and Q. The encryption function is a permutation on $\{0,1,\ldots,N-1\}$. It is given by $E(m) = m^e \ mod\ N$. The decryption function is $D(c) = c^d \ mod\ N$, with the property that $D(E(m)) = m$. i.e. for every m, $m^ed = m \ mod\ N$. To establish these properties and understand how to choose d,e we must review modular arithmetic.

Before we do that let us make some observations about RSA. First, what makes public key cryptography counter-intuitive is the seeming symmetry between the recepient of the message, Alice, and the eavesdropper, Eve. After all, the ciphertext $m^e \ mod\ N$ together with the public key (N,e) uniquely specifies the plaintext m. In principle one could try computing $x^e \ mod\ N$ for all $0 \le x \le N-1$ until one hits upon the ciphertext. However this is prohibitively expensive. RSA breaks the symmetry between Alice and Eve because RSA encryption is actually a trapdoor function: it is easy to compute, and hard to invert, unless you have knowledge of d (the hidden trapdoor). Then it is easy to invert.

Secondly, public key encryption schemes including RSA are substantially slower than symmetric-key encryption algorithms such as DES and AES. For this reason, public key encryption is typically used to establish private session keys between two parties who then communicate using a symmetric encryption scheme. Thus public key encryption is used to solve the key distribution problem in symmetric encryption schemes, where if n people wish to communicate it is necessary to establish $\binom{n}{2}$ keys. For a public key scheme they only need n keys.

3 Algorithms for modular arithmetic

We start by considering two number-theoretic problems – modular exponentiation and greatest common divisor – for which the most obvious algorithms take exponentially long, but which can be solved in polynomial time with some ingenuity. The choice of algorithm makes all the difference.

3.1 Simple modular arithmetic

Two n-bit integers can be added, multiplied, or divided by mimicking the usual manual techniques which are taught in elementary school. For addition, the resulting algorithm takes a constant amount of time to produce each bit of the answer, since each such step only requires dealing with three bits – two input bits and a carry – and anything involving a constant number of bits takes $O(1)$ time. The overall time is therefore $O(n)$, or linear. Similarly, multiplication and division take $O(n^2)$, or quadratic, time.

Modular arithmetic can be implemented naturally using these primitives. To compute $a \bmod s$, simply return the remainder upon dividing a by s. By reducing all inputs and answers modulo s, modular addition, subtraction, and multiplication are easily performed, and all take time $O(\log^2 s)$ since the numbers involved never grow beyond s and therefore have size at most $\lceil \log_2 s \rceil$.

3.2 Modular exponentiation

Modular exponentiation consists of computing $a^b \bmod s$. One way to do this is to repeatedly multiply by a modulo s, generating the sequence of intermediate products $a^i \bmod s$, $i = 1, \ldots, b$. They each take $O(\log^2 s)$ time to compute, and so the overall running time to compute the $b-1$ products is $O(b \log 2s)$, exponential in the size of b.

A repeated squaring procedure for modular exponentiation.

```
function ModExp1($a,b,s$)
Input:  A modulus $s$, a positive integer $a < s$ and a positive exponent $b$
        Let $b_{n-1} \cdots b_1 b_0$ be the binary form of $b$, where $n = \lceil \log_2 b \rceil$
Output:  $a^b \bmod s$

// Compute the powers $p_i = a^{2^i} \bmod s$.
$p_0 = a \bmod s$
for $i = 1$ to $n-1$
    $p_i = p_{i-1}^2 \bmod s$

// Multiply together a subset of these powers.
$r = 1$
for $i = 0$ to $n-1$
```

```
  if $b_i = 1$ then $r = rp_i \bmod s$
return $r$
```

The key to an efficient algorithm is to notice that the exponent of a number a^j can be doubled quickly, by multiplying the number by itself. Starting with a and squaring repeatedly, we get the powers $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log_2 b \rfloor}}$, all modulo s. Each takes just $O(\log^2 s)$ time to compute, and they are all we need to determine $a^b \bmod s$: we just multiply together an appropriate subset of them, those corresponding to ones in the binary representation of b. For instance,

$$a^{25} \;=\; a^{11001_2} \;=\; a^{10000_2} \cdot a^{1000_2} \cdot a^{1_2} \;=\; a^{16} \cdot a^8 \cdot a^1.$$

This repeated squaring algorithm is shown above. The overall running time is $O(\log^2 s \log b)$, *cubic* in the input size when the exponent b is large. This explains the use of encryption exponent $e = 3$ in the RSA cryptosystem — this way encryption can be carried out in quadratic time. For obvious reasons the decryption exponenent d cannot be selected to be small. Thus decryption takes cubic time.

Another procedure for modular exponentiation.

```
function ModExp2($a,b,s$)
Input:  A modulus $s$, a positive integer $a < s$ and a positive exponent $b$
        Let $b_{n-1} \cdots b_1 b_0$ be the binary form of $b$, where $n = \lceil \log_2 b \rceil$
Output:  $a^b \bmod s$

$r = 1$
for $i = n-1$ downto $0$
  $r = r2 \bmod s$
  if $b_i = 1$ then $r = ra \bmod s$
return $r$
```

Squaring a number a^j is equivalent to left-shifting the binary representation of its exponent. This suggests determining $a^b \bmod s$ for $b = b_{n-1} \cdots b_1 b_0$ by iteratively calculating

$$a^{b_{n-1}} \;\longrightarrow\; a^{b_{n-1}b_{n-2}} \;\longrightarrow\; a^{b_{n-1}b_{n-2}b_{n-3}} \;\longrightarrow\; \cdots \longrightarrow a^{b_{n-1}b_{n-2}\cdots b_0}.$$

Each number in this sequence is computed from the previous one by left-shifting the exponent and possibly adding one to it (that is, multiplying by a). This alternative algorithm, shown above, improves the previous algorithm only in the constant factors. It has the same asymptotic running time.

3.3 Modular division and Euclid's algorithm for greatest common divisor

Let us now turn to the question of how the decryption exponent d is selected. To understand this we must first understand how to divide modulo N. In real arithmetic we can divide by any number as long as it is not 0. Division in modular arithmetic is slightly more complicated, but the final rule says that it is possible to divide by $a \bmod s$ whenever $\gcd(a,s) = 1$. In particular, the reciprocal of $a \bmod s$ exists (i.e. b such that $ab = 1 \bmod s$) iff $\gcd(a,b) = 1$. To see why this is so, and to get an efficient algorithm for division we must examine one of the oldest algorithms: it computes the greatest common divisor of two integers a and b: the largest integer which divides both of them. The naive scheme of checking all numbers less than $\min(a,b)$ is exponential time and therefore hopelessly slow. Instead we will rely upon a simple rule discovered in ancient Greece by the eminent mathematician Euclid, which will serve as the basis of an efficient recursive algorithm.

Lemma If $a > b$ then $\gcd(a,b) = \gcd(a \bmod b, b)$.

Proof: Actually Euclid noticed the slightly simpler rule $\gcd(a,b) = \gcd(a-b,b)$ from which the one above can be derived by the repeated subtraction of b from a.

Why is $\gcd(a,b) = \gcd(a-b,b)$? Well, any integer which divides both a and b must also divide both $a-b$ and b, so $\gcd(a,b) \leq \gcd(a-b,b)$. And similarly, any integer which divides both $a-b$ and b must also divide both a and b, so $\gcd(a,b) \geq \gcd(a-b,b)$. $\square$

How long does the `Euclid` algorithm (below) take? We will see that on each successive recursive call one of its arguments gets reduced to at most half its value while the other remains unchanged. Therefore there can be at most $\lfloor \log_2 a \rfloor + \lfloor \log_2 b \rfloor + 1$ recursive calls before one of the arguments gets reduced to zero. The following lemma summarizes this key observation.

Lemma If $a \geq b$ then $a \bmod b \leq a/2$.

Proof: Consider two possible ranges for the value of b. Either $b \leq a/2$, in which case $a \bmod b < b \leq a/2$, or $b > a/2$, in which case $a \bmod b = a - b \leq a/2$. $\square$

For an input size of $n = \lceil \log_2 a \rceil + \lceil \log_2 b \rceil$, there are at most $n+1$ recursive calls, and so the total running time is $O(n^3)$.

Euclid's algorithm for finding the greatest common divisor of two numbers.

```
function Euclid($a,b$)
Input:  Two positive integers $a,b$ with $a \geq b$
Output:  $\gcd(a,b)$

if $b = 0$ then return $a$
return Euclid($b, a \bmod b$)
```

3.4 An extension of Euclid's algorithm

It turns out that $\gcd(a,b)$ can always be expressed as an integer linear combination of a and b, that is, in the form $ax + by$ where x,y are integers. It is not immediately obvious how one would calculate such x,y, even given exponential time, but in fact they can be found quickly by incorporating the following observation into the recursion in Euclid's algorithm.

Lemma: If $\gcd(a \bmod b, b)$ is an integer linear combination of $a \bmod b$ and b, then $\gcd(a,b)$ is an integer linear combination of a and b.

Proof: Write a in the form $bq + r$, where $r = a \bmod b$. By hypothesis, there are some integers x', y' for which $\gcd(a \bmod b, b) = bx' + ry'$. Let $x = y'$ and $y = x' - qy'$; these are also integers and

$$ax + by = ay' + b(x' - qy') = bx' + (a - bq)y' = bx' + ry' = \gcd(a \bmod b, b) = \gcd(a,b),$$

where the final equality is simply Euclid's rule. $\square$

The `Extended-Euclid` algorithm given below directly implements this inductive reasoning. One way to prove its correctness in detail (you should try this) is to use induction on $\max(a,b)$.

Theorem: For any positive integers a,b, the `Extended-Euclid` algorithm returns integers x,y such that $ax + by = \gcd(a,b)$.

A simple extension of Euclid's algorithm.

```
function Extended-Euclid($a,b$)
Input:  Two positive integers $a,b$ with $a \geq b$
Output:  Integers $x,y,d$ such that $d = \gcd(a,b)$ and $ax + by = d$

if $b = 0$ then return $(1,0,a)$
by division find $q,r$ such that $a = bq + r$
$(x',y',d) =$ Extended-Euclid($b,r$)
return $(x = y', y = x' - qy', d)$
```

This extension of Euclid's algorithm is the key to dividing in the modular world. In real arithmetic every $a \neq 0$ has a multiplicative inverse $1/a$, and dividing is the same as multiplying by this inverse. In modular arithmetic, a has a multiplicative inverse $\bmod s$ iff $\gcd(a,s) = 1$. By the extended Euclid algorithm, if $\gcd(a,s) = 1$, then there are integers x,y such that $ax + sy = 1$. Reducing both sides of this sum modulo s, we get $ax \equiv 1 \bmod s$. In short: x is the multiplicative inverse of a modulo s, and we have a quick way of finding it.

Corollary: If a is relatively prime to $s > a$, then a has a multiplicative inverse modulo s, and this inverse can be found in time $O(\log^3 s)$.

Returning to the question of how the decryption exponent d is selected. It turns out that d is the multiplicative inverse of the encryption exponent $e \bmod (P-1)(Q-1)$. By the above discussion such a d exists and can be efficiently computed iff $\gcd(e,(P-1)(Q-1)) = 1$. Since for efficiency reasons we wish to choose $e = 3$, it follows that we must pick P,Q each congruent to 2 mod 3. d is then computed by using the Extended Euclid algorithm.

4 The Trapdoor

Let us now turn to the task of justifying the fact that RSA encryption is a permutation of the numbers modulo N, and the fact that for all $0 \leq x \leq N-1$, $x^{ed} = x^{ed} = x \bmod N$ whenever $ed = 1 \bmod (P-1)(Q-1)$.

We need the following theorem of Fermat's:

Theorem [Fermat's Little Theorem] If p is prime then for every $1 \leq a < p$,

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof: Let S be a set consisting of the numbers $1,2,\ldots,p-1$ modulo p. Now multiply them each by a modulo p and call that set T. We will show that S and T are identical: they have exactly the same elements. Therefore the products of their elements, $(p-1)! \bmod p$ and $a^{p-1} \cdot (p-1)! \bmod p$ respectively, are also identical. Dividing out by $(p-1)!$ completes the proof.

First we make sure a is invertible modulo p. From $1 \leq a < p$ we know a and p are relatively prime, and by Corollary 3.4 a has a multiplicative inverse modulo p. Call it a^{-1}.

Next we show that S and T are identical because the elements $a \cdot i$, $i = 1 \ldots p-1$, are all distinct and non-zero modulo p. If $a \cdot i \equiv a \cdot j \pmod{p}$, then multiplying both sides by a^{-1} gives $i = j$. By similar reasoning, none of the numbers in T are congruent to zero modulo p.

Therefore T is the same as S and

$$1 \cdot 2 \cdots (p-1) \equiv (a \cdot 1) \cdot (a \cdot 2) \cdots (a \cdot (p-1)) \pmod{p},$$

in other words $(p-1)! \equiv a^{p-1}(p-1)! \pmod{p}$. We can divide out by $(p-1)!$ because it is relatively prime to p and therefore invertible modulo p. □

We will use Fermat's theorem to establish the fact that for $0 \leq x \leq N-1$, $x^{k(P-1)(Q-1)+1} = x \bmod N$. Here k is an arbitrary integer and $N = P \cdot Q$ where P, Q are primes.

First, observe that for all x, $x^{k(P-1)(Q-1)+1} = x \bmod N$. If $x \neq 0 \bmod P$, this follows from Fermat's theorem, since $x^{k(P-1)(Q-1)+1} = x^{{P-1}^{k(Q-1)}} \cdot x = x \bmod P$ Also if $x = 0 \bmod P$ then clearly $0^{k(P-1)(Q-1)+1} = 0 \bmod N$ trivially.

An equivalent way of writing this is that $P \mid x^{k(P-1)(Q-1)+1} - x$.

By a similar argument $Q \mid x^{k(P-1)(Q-1)+1} - x$.

Since P, Q are distinct primes, it follows that $P \cdot Q \mid x^{k(P-1)(Q-1)+1} - x$.

This property establishes that RSA decryption works as claimed, since $ed = 1 \bmod (P-1)(Q-1)$ implies that $ed = k(P-1)(Q-1)+1$ for some k. Now by the above property for all x, $x^{ed} = x \bmod N$.

5 Selecting Large Primes:

Let us now turn to the last issue in implementing the RSA cryptosystem, namely how does Alice select the primes P and Q to create $N = P \cdot Q$. We have already bemoaned the fact that searching for the factors of a large number seems intractable. However, the closely related problem of testing a number for primality is easy, since Fermat's theorem forms the basis of a kind of litmus test which helps decide whether a number is prime or not, without explicitly indentifying its factors. The idea is that to test if a number M is prime we select an $a \bmod M$ at random and compute $a^{M-1} \bmod M$. If the result is not 1 then by Fermat's theorem it follows that M is not a prime. On the other hand if the result is 1 then this can be shown to provide evidence that M is indeed prime. With a little more work this forms the basis of an efficient probabilistic algorithm for testing primality.

For the purpose of selecting a random large prime (of several hundred bits), it suffices to pick a random number of that length and test it for primality. The prime number theorem tell us that among the n bit numbers roughly $\frac{log\ e}{n} \approx \frac{1.44}{n}$ fraction are prime. So $O(n)$ iterations of this procedure will result in the selection of a prime.

Since we are only testing random numbers for primality, it is sufficient for most practical purposes to perform the Fermat test with base $a = 2, 3, 5$. Numbers which pass this test have been jokingly referred to as industrial grade primes. To see how discerning the Fermat test is, consider the following example:

Suppose we perform the Fermat test with base $a = 2$ for all numbers $M \leq 25 \times 10^9$. The number of primes in this range is about 10^9. The number of pseudoprimes (numbers which are actually composites, but pass the Fermat test with base $a = 2$) is about $20,000$. Thus the chance of erroneously outputting a composite is about $\frac{20 \times 10^3}{10^9} = 2 \times 10^{-5}$. This chance of error decreases rapidly as the length of the numbers involved is increased.

Bob chooses his public and private keys.

- He starts by picking two large (n-bit) random primes P and Q.
- His public key is (N,e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
- His secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the `Extended-Euclid` algorithm.

Alice wishes to send message x to Bob.

- She looks up his public key (N,e) and sends him $y = (x^e \bmod N)$, computed using an efficient modular exponentiation algorithm.
- He decodes the message by computing $y^d \bmod N$.

Figure 1: RSA.

6 RSA

Let us review the resulting RSA cryptosystem, which is named after its inventors Rivest-Shamir-Adleman. Anybody can send a message to anybody else using publicly-available information, rather like addresses or phone numbers. Each person has a public key known to the whole world, and a secret key known only to himself. When Alice wants to send message x to Bob, she encodes it using his public key e. He decrypts it using his secret key d, to retrieve x. Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them, under certain simple assumptions.

There are two procedures involved in RSA: the initial choice of keys, and the message-sending protocol. These are described in above. Both make heavy use of the efficient number-theoretic primitives we have developed.

The security of RSA hinges upon a simple assumption:

> Given N, e, and $y = x^e \bmod N$, it is computationally intractable to determine x.

How might Eve try to guess x? She could experiment with all possible values of x, each time checking whether $x^e \equiv y \bmod N$, but this would take exponential time. Or she could try to factor N to retrieve P and Q, and then figure out d by inverting e modulo $(P-1)(Q-1)$, but we believe factoring to be hard. Moreover, it can be shown that guessing d is as hard as factoring N: once d is known, it is easy to recover P and Q. This intractability is normally a source of dismay; the insight of RSA lies in using it to advantage.