CS 161 Computer Security
Fall 2008 Dawn Song

Notes 3

1 Modes of Operations:

A symmetric encryption scheme allows Alice and Bob to privately exchange a sequence of messages in the presence of an eavesdropper Eve. We will assume that Alice and Bob share a random secret key K. How Alice and Bob managed to share a key without the adversary's knowledge is not going to be our concern here. The encryption scheme consists of an encryption algorithm $\mathscr{E}$ that takes as input the key K and the plaintext message $M \in \{0,1\}^*$, and outputs the ciphertext. The decryption algorithm $\mathscr{D}$ takes as input the key and the ciphertext and reconstructs the plaintext message M. In general the encryption algorithm builds upon a block cipher to accomplish two goals: one is to show how to encrypt arbitrarily long messages using a fixed length block cipher. The other is to make sure that if the same message is sent twice, the ciphertext in the two transmissions is not the same. The encryption algorithm to achieve these goals can either be randomized or stateful - it either flips coins during its execution, or its operation depends upon some state information. The decryption algorithm is neither randomized nor stateful.

ECB Mode (Electronic Code Book): In this mode the plaintext M is simply broken into n bit blocks $M_1, \ldots M_l$, and each block is encoded using the block cipher: $C_i = E_K(M_i)$. The ciphertext is just a concatenation of these individual blocks: $C = C_1 \cdot C_2 \cdots C_l$. This scheme is adequate for simple tasks such as encrypting PINs for cash machine systems. However any redundancy in the blocks will show through and allow the eavesdropper to deduce information about the plaintext. We will discuss this in more detail after formalizing the notion of the security of a symmetric encryption scheme below.

CBC Mode (Cipher Block Chaining): This is a popular mode for commercial applications. A random n bit string, the initial vector or IV is selected. Define $C(0) = E_K(IV)$. The i^{th} encrypted block $C_i = E_K(C_{i-1} \oplus M_i)$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. The CBC mode does provide strong security guarantees on the privacy of the plaintext message. This is formalized later in this lecture and explored further in the homework.

OFB Mode (Output Feedback Mode): In this mode, the initial vector IV is repeatedly encrypted to obtain a set of keys K_i as follows: $K_0 = IV$ and $K_i = E_K(K_{i-1})$. These keys K_i are now used as keys for a one-time pad, so that $C_i = K_i \oplus M_i$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. This scheme suffers from an important weakness — the message is malleable. i.e. suppose that the adversary happens to know that the j^{th} block of the message specifies the amount of money being transferred to his account from the bank and is 100. Since he knows both M_j and C_j, he can determine K_j. He can then substitute any n bit block in place of M_j and get a new ciphertext where the 100 is replaced by any amount of his choice.

Counter Encryption: One drawback of the CBC and OFB mode is that successive blocks must be encrypted sequentially. For high speed applications it is useful to parallelize these computations. This is easily achieved by encrypting a counter initialized to IV to obrain a set of keys $K_i = E_K(IV + i)$. As before the block M_i is then encrypted simply as $K_i \oplus M_i$.

2 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.

3 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), where $ed = 1\ mod (P-1)(Q-1)$. 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, $m^e d = m\ mod\ N$.

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.

4 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.

4.1 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 $\lceil \log_2 s \rceil$.

4.2 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} ··· b_1 b_0 be the binary form of b, where n = ⌈log_2 b⌉
Output:  a^b mod s

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

// Multiply together a subset of these powers.
r = 1
for i = 0 to n-1
  if b_i = 1 then r = r p_i mod 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.

4.3 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 mod s whenever $\gcd(a,s) = 1$. In particular, the reciprocal of a mod 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 $\gcd(a,b) \geq \gcd(a-b,b)$. □

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 $a \bmod b = a - b \leq a/2$. □

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 ≥ b
Output:  gcd(a,b)

if b = 0 then return a
return Euclid(b, a mod b)
```

4.4 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. □

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 ≥ 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 mod$s$ 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 mod $(P-1)(Q-1)$. 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.