

Cryptography: Public Key Cryptography; Mathematical Preliminaries

Greg Plaxton

Theory in Programming Practice, Spring 2004

Department of Computer Science

University of Texas at Austin

Secure Communication

- Earlier we discussed the problems associated with XORing the data with a random secret key
 - Need a secure method to exchange keys
 - Should use a new secret key for each communication (“one-time pad”)
- Other simple encryption schemes such as substitution cyphers are easily broken
 - Letter (and letter combination) frequencies give clues
- Public key cryptography yields a much more satisfactory solution

Public Key Cryptography (Diffie and Hellman)

- Each user Bob a public key (available to everyone) and a private key (known only to Bob)
 - Bob's public key is an encryption function f (specific to Bob) that is to be applied to any message sent to him
 - Bob's private key is f^{-1} , so Bob can use this function to decrypt messages that he receives
- Avoids the key exchange problem
- The function f needs to be “one-way”
 - Given any message x , it is easy to compute $f(x)$
 - Given any encrypted message $f(x)$, it is hard (i.e., requires a prohibitive amount of computational power) to compute x

Public Key Cryptography: RSA (Rivest, Shamir, and Adelman)

- The encryption function is chosen from a specific family of functions that are conjectured to be hard to invert
- If a fast algorithm for factoring were to be found, the “one-wayness” of this family of functions would be broken
 - We remark that it is conceivable that RSA could be broken without obtaining a fast factoring algorithm

Hardness of Factoring

- Every positive integer has a unique prime factorization
- How hard is it to determine this factorization?
- On the one hand, this may seem like an easy problem
 - Given any positive integer n , we can determine whether n has a nontrivial factor (i.e., a factor other than 1 or n) in $O(\sqrt{n})$ integer divisions
 - Why does this simple idea not yield a practical (and polynomial-time) algorithm?

Hardness of Factoring

- An algorithm is said to run in polynomial time if its running time is upper bounded by some polynomial in the input size (measured in bits)
- If the input to a factoring algorithm is an integer n , then the input size is approximately $\log_2 n$ bits
- Note that \sqrt{n} is exponential in the input size, since

$$\sqrt{n} = 2^{\frac{1}{2} \log_2 n}$$

- Factoring a 100-digit number might take something like 10^{50} operations
 - Assume a computer can perform 10^9 such operations per second
 - There are about $3 \cdot 10^7 < 10^8$ seconds in a year
 - So we would need something like 10^{33} computers to perform such a computation within a year

Factoring: State of the Art

- The fastest (general-purpose) factoring algorithm to date is the number field sieve algorithm of Buhler, Lenstra, and Pomerance
 - For d -bit numbers, the running time is

$$2^{\Theta(d^{\frac{1}{3}}(\log_2 d)^{\frac{2}{3}})}$$

- This is a huge improvement over the naive algorithm, which has a running time of $2^{\Theta(d)}$
- In 1999, an implementation of the number field sieve algorithm was used to factor a 155-digit (512 bit) number of the kind (product of two large primes) used in 512-bit implementations of RSA
 - The computation was spread across about 200 machines and required about 8000 MIPS years
 - This result demonstrates that 512-bit RSA is no longer secure
 - Okay, let's use 1024-bit RSA

RSA: Mathematical Preliminaries

- Fermat's Little Theorem
- Extended Euclid algorithm

Fermat's Little Theorem

- For any prime p , and any positive integer a such that p does not divide a ,

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

- Proof:
 - Note that if i and j are integers between 1 and $p - 1$ inclusive and $a \cdot i$ is congruent to $a \cdot j$ modulo p , then $i = j$; furthermore, $a \cdot i$ is not congruent to zero modulo p
 - Thus $a^{p-1} \cdot (p - 1)!$ is congruent to $(p - 1)!$ modulo p , i.e., p divides $(a^{p-1} - 1) \cdot (p - 1)!$
 - Since p does not divide $(p - 1)!$, p divides $a^{p-1} - 1$

Euclid's GCD Algorithm

- Euclid's algorithm computes the greatest common divisor of two nonnegative integers (at least one of which is nonzero)
- Here is an efficient implementation of Euclid's algorithm
 - What is the running time of this algorithm as a function of the input size (i.e., the total number of bits in the binary representations of x and y)?

$u, v := x, y$

$\{u \geq 0, v \geq 0, u \neq 0 \vee v \neq 0, \gcd(x, y) = \gcd(u, v)\}$

while $v \neq 0$ **do**

$u, v := v, u \bmod v$

od

$\{\gcd(x, y) = \gcd(u, v), v = 0\}$

$\{\gcd(x, y) = u\}$

Euclid's GCD Algorithm

- Here is a slight modification of the preceding algorithm

```
 $u, v := x, y$   
 $\{u \geq 0, v \geq 0, u \neq 0 \vee v \neq 0, \gcd(x, y) = \gcd(u, v)\}$   
while  $v \neq 0$  do  
     $q := \lfloor u/v \rfloor;$   
     $u, v := v, u - v \times q$   
od  
 $\{\gcd(x, y) = u\}$ 
```

A GCD-Like Problem

- Given nonnegative integers x and y , at least one of which is nonzero, our goal is to compute integers a and b such that $a \cdot x + b \cdot y = \gcd(x, y)$
 - Note that a and b need not be positive, nor are they unique
- We will now develop an extended Euclid algorithm that can be used to compute such a pair of integers a and b
 - The proof of correctness of the algorithm, which we develop along with the algorithm, provides a proof of the existence of such a pair of integers

Towards an Extended Euclid Algorithm

$u, v := x, y; a, b := 1, 0; c, d := 0, 1;$

while $v \neq 0$ **do**

$q := \lfloor u/v \rfloor;$

$\alpha : \{ (a \times x + b \times y = u) \wedge (c \times x + d \times y = v) \}$

$u, v := v, u - v \times q;$

$a, b, c, d := a', b', c', d'$

$\beta : \{ (a \times x + b \times y = u) \wedge (c \times x + d \times y = v) \}$

od

- It remains to determine expressions a', b', c', d' so that the given annotations are correct

Determining a' and b'

Using backward substitution, we need to show that the following proposition holds at program point α .

$$(a' \times x + b' \times y = v) \wedge (c' \times x + d' \times y = u - v \times q)$$

We are given that the proposition $(a \times x + b \times y = u) \wedge (c \times x + d \times y = v)$ holds at α . Therefore, we may set

$$a', b' = c, d$$

Determining c' and d'

$$\begin{aligned} & c' \times x + d' \times y \\ = & \{\text{from the invariant}\} \\ & u - v \times q \\ = & \{a \times x + b \times y = u \text{ and } c \times x + d \times y = v\} \\ & (a \times x + b \times y) - (c \times x + d \times y) \times q \\ = & \{\text{algebra}\} \\ & (a - c \times q) \times x + (b - d \times q) \times y \end{aligned}$$

So, we may set

$$c', d' = a - c \times q, b - d \times q$$

Extended Euclid Algorithm

$u, v := x, y; a, b := 1, 0; c, d := 0, 1;$

while $v \neq 0$ **do**

$q := \lfloor u/v \rfloor;$

$\alpha : \{ (a \times x + b \times y = u) \wedge (c \times x + d \times y = v) \}$

$u, v := v, u - v \times q;$

$a, b, c, d := c, d, a - c \times q, b - d \times q$

$\beta : \{ (a \times x + b \times y = u) \wedge (c \times x + d \times y = v) \}$

od

- What is the running time of this algorithm?

Extended Euclid Algorithm: Correctness

Upon termination

$$\begin{aligned} & a \times x + b \times y \\ = & \{\text{from the invariant}\} \\ & u \\ = & \{v = 0 \text{ and } \gcd(u, 0) = u, \text{ for } u \neq 0\} \\ & \gcd(u, v) \\ = & \{\gcd(x, y) = \gcd(u, v)\} \\ & \gcd(x, y) \end{aligned}$$

Extended Euclid Algorithm: Example

Running extended Euclid with $x = 157$ and $y = 2668$:

a	b	u	c	d	v	q
1	0	157	0	1	2668	
						0
0	1	2668	1	0	157	
						16
1	0	157	-16	1	156	
						1
-16	1	156	17	-1	1	
						156
17	-1	1	-2668	157	0	