CNS Lecture 8

Security through mathematics

Number theory

Diffie-Hellman

Public key crypto

RSA

El Gamal

DSA

Assignments

midterm cs594 paper

Lectures

1. Risk, viruses
2. UNIX vulnerabilities
3. Authentication & hashing
4. Random #s classical crypto
5. Block ciphers DES, RC5
6. AES, stream ciphers RC4, LFSR
7. MIDTERM ☹
8. Public key crypto RSA, D-H
9. ECC, ssh/pgp
10. PKI, SSL

spring break ☺

11. Network vulnerabilities
12. Network defenses, IDS, firewalls
13. IPsec, VPN, Kerberos, secure OS
14. Crypto APIs
15. review

In the news

- Dept of Commerce machines broken into multiple times (root kits) by Chinese hackers – going to replace all hardware and software!
- McAfee epolicy buffer overflow
- AOL's "you've got pictures" buffer overflow
- 120 new vulnerabilities reported this week (SANS)

You are here …

Attacks & Defenses

- Risk assessment✓
- Viruses✓
- Unix security✓
- authentication✓
- Network security
  - Firewalls,vpn,IPsec,IDS
- Forensics

Cryptography

- •Random numbers✓
- •Hash functions✓
  - MD5, SHA,RIPEMD
- •Classical + stego✓
- •Number theory
- •Symmetric key✓
  - DES, Rijndael, RC5
- •Public key
  - RSA, DSA, D-H,ECC

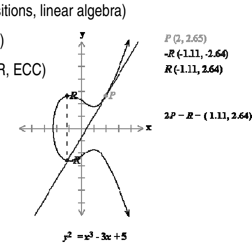
Applied crypto

- •SSH
- •PGP
- •S/Mime
- •SSL
- •Kerberos
- •IPsec

The mathematics of cryptography

Finite (discrete) mathematics

- •Modular arithmetic (shift ciphers, polyalphabets, Hill cipher)
- •Primes and prime factors, greatest common divisor, BIG integer libraries
- •Linear transforms (row/column transpositions, linear algebra)
- •Exponentiation/discrete logs (D-H, RSA)
- •Polynomial arithmetic (CRC, AES, LFSR, ECC)
- •Elliptic curves (ECC)



Number theory

Security through mathematics

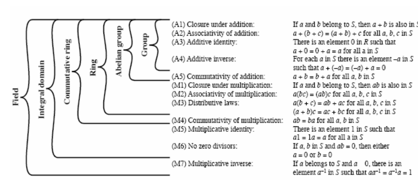
why? -- basis for public key cryptography

not permutations or substitutions

modular arithmetic

hard problems (factoring, discrete logs)

Number theory



prime -- number only divisible by 1 or itself

any integer can be expressed as the product of prime powers (fundamental theorem of arithmetic)

$$a = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$$

$3600 = 2^4 \times 3^2 \times 5^2$

Modular arithmetic

- mod (congruence), remainder after dividing
- if a mod n = b then a = kn + b
- 11 mod 3 = 5 mod 3 = 2
- modular arithmetic (+,*) on non-negative integers Z_n
  - associative   a+ (b +c) mod n = (a+b) +c mod n
  - commutative   ab mod n = ba mod n
  - distributive   a(b+c) mod n = (ab + ac) mod n
  - reducible   ab mod n = ((a mod n)(b mod n)) mod n
  - At least a commutative ring
- Additive identity: 5+0 = 5
- additive inverse: (5 + ?) mod 8 = 0

lets us work with smaller numbers

multiplication

- multiplicative inverse $(x^{-1} x)$ mod n = 1
  3*9 mod 26 = 1
  4*x mod 8 = 1 ?
- only numbers relatively prime to n have multiplicative inverse (Z_{n^*})
- **relatively prime → don't share any common factors**
- if p is prime, all elements have multiplicative inverse (except 0)
  Z_p is a field

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

(a) Addition modulo 8

w	$-w$	w^{-1}
0	0	—
1	7	1
2	6	—
3	5	3
4	4	—
5	3	5
6	2	—
7	1	7

(c) Additive and multiplicative inverses modulo 8

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

(b) Multiplication modulo 8

Euler's totient

totient function
  $\phi(n)$ how many numbers relatively prime to n

$\phi(6)=2$ (1 and 5)
$\phi(7)=6$ (1 thru 6) since 7 is prime,
$\phi(8)=4$ (1,3,5,7)

If p is prime, $\phi(p)$ = p-1

What is $\phi(n)$ if n= pq, and p and q are prime?
have to factor n to calculate totient
$\phi(n) = \phi(p)\ \phi(q) = (p-1)(q-1)$

Table 8.2 Some Values of Euler's Totient Function $\phi(n)$

n	$\phi(n)$	n	$\phi(n)$	n	$\phi(n)$
1	1	11	10	21	12
2	1	12	4	22	10
3	2	13	12	23	22
4	2	14	6	24	8
5	4	15	8	25	20
6	2	16	8	26	12
7	6	17	16	27	18
8	4	18	6	28	12
9	6	19	18	29	28
10	4	20	8	30	8

Euler's theorem: If a and n are relatively prime, $a^{\phi(n)}$ = 1 mod n

a=3; n=10; $\phi(10)$=4; 3^4 mod 10 = 81 mod 10 = 1

Modular shortcuts

- ab mod n = ((a mod n)(b mod n)) mod n
- exponentiation a^b mod n,   use squaring
  x^{16} 4 multiplications
  $x\,x$ $x^2\,x^2$ $x^4\,x^4$ $x^8\,x^8$
  - both: a^8 mod n = $((a^2 \bmod n)^2 \bmod n)^2 \bmod n$
    rather than 7 multiplies, and one huge division
- Chinese Remainder Theorem (CRT)
  - do arithmetic mod factors of n (faster)
  - handy in RSA (example later)

Euclid's algorithm

greatest common divisor

- gcd(a.b) = gcd(b, a mod b)
- gcd(12,8)=4 and gcd(12,25)=1
- relatively prime if gcd is 1
- **extended gcd** can be used to find multiplicative inverse (if it exists)
  - used in IDEA and CRT for RSA

gcd(124,16) = gcd(16, 124 mod 16)
124 mod 16 = 12
16 mod 12 = 4
12 mod 4 = 0
gcd is 4

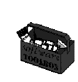
see gcd.c (extended euclid)

primes

Prime Number theorem:
  number of primes < n asymptotic to n/(ln n)
- primes < 1,000,000 = 78,498   < 1 billion = 51 million
- primes < $2^{512} \cong 10^{150}$ -- not that many atoms in the universe

generating primes
- need large primes for key generation (RSA,DSA, D-H)
- need to do only once (usually)
- usually part of crypto library (not your problem)
- "Is n prime?" easier than "What are the factors of n?"

Crypto Toolkit
secret-key crypto ✓
public-key crypto
big-number math
random numbers✓
prime numbers
hash functions✓



Primality testing ***Is p prime?***

- infeasible to check all factors for really BIG integer
- can't determine absolutely if big number is prime, but can test for "highly probable"
- **Fermat's theorem**
  if p is prime and *a is not divisible by p* (relatively prime) then $a^{p-1} = 1 \bmod p$
- **Lehman variation** of Fermat's theorem
  choose random a, if p is prime, $a^{p-1} = 1 \bmod p$
  exceptions: Carmichael numbers, e.g., p= 561 = 3 x 11 x 17 (pseudo-primes)
- **Rivest variation** of Fermat's $2^{p-1} = 1 \bmod p$
  true if p is prime, but there are pseudo-primes n that meet the test.
  For 256-bit number (2^{256}), 'bout 10^{74} primes and 10^{52} pseudo-primes,
  so chance of 1 in 10^{22} that p satisfies the test and is not prime
- **Miller-Rabin**
  if there is a solution to $x^2 = 1 \bmod p$ other than 1 and -1, then p is NOT prime
  So try lots of random x's
  probability p is NOT prime after k successful tests, $(1/4)^k$

See Schneier, *Applied Cryptography*

Finding a prime

1. generate random n-bit number p
2. set hi-bit to 1, low-bit to 1
3. verify p is not divisible by first 2048 primes
4. perform Miller-Rabin for some random *a*. If p passes, generate another *a*, and repeat test (5 times?). If it fails, generate a new p and go back to step 1.

roughly what OpenSSL lib does in BN_generate_prime ()

density of primes:

- proportion of positive integers < x that are prime is roughly 2/ ln x
- for 512-bit n (2^{512}) can find a prime in 177 tries
- this is what takes time when you first generate your PGP keys

Problems with symmetric key crypto

- Symmetric key crypto (DES, AES) needs pre-shared key
- How do Alice and Bob get their key?
- For N people, need N^2 sets of shared keys

- Mathematics will give us some other choices
  - Diffie-Hellman ('76) will allow Alice and Bob to establish a shared key
  - Public key crypto (RSA, ECC) will allow Alice to publish her public key

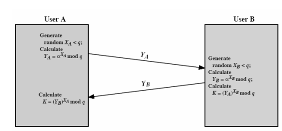
Diffie-Hellman

- *method for two parties to establish a secret key with no previous shared secrets -- cool!*
- *doesn't do encryption or signatures*

algorithm

- publish a **generator** g and prime modulus q
- Alice generates random x, and calculates
  $X = g^x \bmod q$
- Bob generates random y, and calculates
  $Y = g^y \bmod q$
- keep x and y secret (private keys)
- they exchange X and Y (public keys)
- they calculate X^y and $Y^x \bmod q$ to get k
  $k = X^y \bmod q = g^{xy} \bmod q$
  $k = Y^x \bmod q = g^{yx} \bmod q$

- now they can use k for DES or whatever
- Need good randomness



generators primitive roots
If p is prime, then g is a generator for Z_{p^*}, if g^i generates all the members of Z_{p^*} $(0 \le i \le \phi(p) - 1)$
There are $\phi(\phi(p))$ generators.
Guess and test to find them.

a = 6 is a generator for Z_{13^*}

```
i            0  1  2  3  4  5  6  7  8  9 10 11
6^i mod 13   1  6 10  8  9  2 12  7  3  5  4 11
```


Primitive roots/generators

- Primitive roots of prime number, 19
  - There should be $\phi(\phi(19)) = \phi(18) = 6$ of them
  - Roots are 2, 3, 10, 13, 14, and 15
  - When doing exponentiation mod p, using a primitive root insures full period (D-H, ElGamal)

Table 8.3 Powers of Integers, Modulo 19

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

D-H examples

- p =11 and g =2
- Alice gets random x = 4, Bob y=6
- Alice X = 2^4 mod 11 = 5
- Bob Y = 2^6 mod 11 = 9
- exchange 5 and 9 (Eve can observer this)
- Alice Y^x mod 11 = 9^4 = 6561 = 5 mod 11
- Bob X^y mod 11 = 5^6 = 15625 = 5 mod 11
- k =5

UNIX example with *bc* using
p=97 g=5 a=36 b=58

```
(5^36)%97
50
(5^58)%97
44
(50^58)%97
75
(44^36)%97
75
```

D-H uses

Used to generate session keys between two parties in

- Netscape (SSL)
- Secure PVM
- stel (secure TELNET)
- SKIP, ISAKMP, GKMP
- Cisco encrypting routers
- nautilus
- Sun secure RPC (keyserv)

- variation: publish your "public key" Y, then Bob wouldn't have to be online when Alice wants to send him a message encrypted with Y (SKIP)

D-H strengths

- easy to calculate g^x mod p
- Eve can capture X and Y, but infeasible to do discrete log, find x given X
- choose big (1024 bits) prime p
- nice if (p-1)/2 is also prime (**strong prime**)
- find g that is generator mod p, that is, g^a will generate all elements (1 to p-1)
  - e.g., 2 is a generator mod 11
  - –given p, there are ways to find g
- you can use published p, g pairs
- need good random number generator (big integers) for secrets x and y
- great for session keys, perfect forward secrecy
- strong against passive (offline) attacks

D-H weakness

- vulnerable to active attack (Trudy in the middle)
- no authentication (not sure who you're talking to)

Alice, Bob, Trudy know p and g
Alice and Bob think they are exchanging with each other
Trudy, in the middle, does exchange with each
Trudy establishes K_{AY} and K_{BX}
Trudy can decrypt, re-encrypt and relay, or make changes!
loss of privacy and integrity

Alice $\underset{Y}{\overset{A}{\rightleftarrows}}$ Trudy $\underset{B}{\overset{X}{\rightleftarrows}}$ Bob

countermeasures

- shared secret (STEL, .stel_secret), e.g., use it to encrypt K in a message
- sign key exchange with private key (GKMP, ISAKMP, Cisco's encrypting routers), use RSA or DSA
- Bellovin-Merritt encrypt D-H exchange with shared secret (EKE)
- SPEKE
- Assignment 8?

EKE (mutual authentication)
•Alice and Bob share password S
•Alice calculates X, sends X encrypted with S
•Bob calculates Y, receives X, decrypts with S, and calculates K
•Bob sends Y encrypted with S and a challenge R encrypted with K
•Alice decrypts Y, calculates K, decrypts challenge R
•Alice generates challenge T and sends R and T encrypted with K
•Bob decrypts challenges, and sends back T encrypted with K

SPEKE – licensed, mutual authentication shared password S, huge prime p where (p-1)/2 also prime
•Alice sends $A=S^{2x}$ mod p
•Bob sends $B=S^{2y}$ mod p
•each calculate $K = A^{2y} = B^{2x}$
•Bob sends hash(hash(K))
•Alice sends hash(K)
•each verify hashes

Implementing D-H

- need random numbers
- may also need to find prime and generator (use known ones)
- want big prime (1024 bits)
- need multiprecision integer library (UNIX mp, -lssl, GNU gmp)
  - see example dhtest.c

or perl BigInt, or C++ Integer, or Java BigInteger

Crypto Toolkit
secret-key crypto ✓
public-key crypto
big-number math
random numbers✓
prime numbers ✓
hash functions✓

BIG integer arithmetic

- Need software to do arithmetic on 1000-bit (100's of digits) numbers
- CPU's like to do 32-bit integer arithmetic
- Need data structures and functions for big integers
  - –Vectors of 32-bit words
  - –Routines for allocate/free, convert, print, read/write
  - –Routines for arithmetic (+ - * / mod exp compare)
  - –See HAC chapter 14
- Libraries for C/FORTAN, classes/methods for C++/Java

BIG integer software

```
UNIX bc command

perl
   use Math::BigInt;

   a = Math::BigInt->new("4324567832342");
   c = a +1;
   print c;

C++
   #include <Integer.h>

   Integer bigi, bigj, bigmod;
   bigi = (bigj*bigi)%bigmod +55;
   cout << bigi;
```

operator overloading is nice
no mod exponentiation (means slow)
really need IntegerMod class ? left to the reader

```
Java
   import java.math.BigInteger;
   BigInteger includes mod exponentiation
```


DHTest.java

```java
import java.io.*;
import java.security.*;
import java.math.BigInteger;

public class DHTest {
    public static void main(String[] args) throws IOException {
    BigInteger prime,generator, sharedsecret, mysecret,
      mypublic, hispublic, hissecret;

    generator = BigInteger.valueOf(3L);
    prime = new
  BigInteger("10600793837847095502437519478711009735280641408663083351958564847666239803449310577396474102435915795190171045667853582308109442172042019236265927024041691035249779046021175308158513569425084317265969515883790351094287231289936792710938093381122200287344747549868584453794340074848422757005577516103310507631 9");

   mysecret = new BigInteger(1024, new SecureRandom());
   mysecret = mysecret.mod(prime);
   mypublic = generator.modPow(mysecret,prime);

   hissecret = new BigInteger(1024, new SecureRandom());
   hissecret = hissecret.mod(prime);
   hispublic = generator.modPow(hissecret,prime);

   sharedsecret = hispublic.modPow(mysecret,prime);
   System.out.println("secret " + sharedsecret.toString());
   sharedsecret = mypublic.modPow(hissecret,prime);
   System.out.println("secret " + sharedsecret.toString());
    }
}
```


BIG integer software (C)

GNU's MP library

```c
#include "gmp.h"

mpz_t x, y, z;

void mpz_init (mpz_t integer)
void mpz_set_ui (mpz_t rop, unsigned long int op)
int mpz_set_str (mpz_t rop, char *str, int base)
void mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)
void mpz_random (mpz_t rop, mp_size_t max_size)
char * mpz_get_str (char *str, int base, mpz_t op)
```

link with *libgmp.a*

OpenSSL's Big Number API

```c
#include <openssl/bn.h>

 BIGNUM *BN_new(void);
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_sub(BIGNUM *r, const BIGNUM *a, const BIGNUM *b);
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
int BN_mod_mul(BIGNUM *ret, BIGNUM *a, BIGNUM *b, const
 BIGNUM *m, BN_CTX *ctx);
int BN_rand(BIGNUM *rnd, int bits, int top, int bottom);
BIGNUM *BN_generate_prime(BIGNUM *ret, int bits,int
 safe,BIGNUM *add, BIGNUM *rem, void (*callback)(int, int,
 void *), void *cb_arg);
BN_hex2bn() BN_bn2hex()
```

- Native assembly language can speed these up

Diffie Hellman example (OpenSSL lib)

```c
/* gcc -o dhtest dhtest.c -lssl */
#include <stdio.h>
#include <openssl/bn.h>
/* 1024 bits modulus */
char *defaultmodulostring = "\
96F5D737535D8BC982698A80AB91DAE28E84E2982071998880C736B5\
695AF14D015401A942186B865496ECBECAFE964A5E70B7031F5756C0\
60AD53528687F4FBFF059150D529638C11FA6FAB6A58785DBD62D73D\
1001014BE3EA53E97D43944F1EF83885196E5BCDD4098744B7673B54\
BA2EB1FDB29ED2C3BA3AD8644FAFF05F";

main()
{
        BIGNUM *mod, *three, *sharedsecret, *mysecret, *mypublic, *hispublic,
                *hissecret;
        BN_CTX *ctx;
        char *p;

        mod = BN_new();
        three = BN_new();
        sharedsecret = BN_new();
        mysecret = BN_new();
        mypublic = BN_new();
        hispublic = BN_new();
        hissecret = BN_new();
        ctx = BN_CTX_new();
```



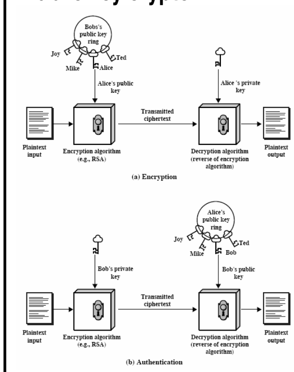
```c
        BN_set_word(three,3);
        BN_hex2bn(&mod,defaultmodulostring);
        BN_print_fp(stdout,mod); printf("\n");

        BN_rand(mysecret,1024,1,1);
        BN_nnmod(mysecret,mysecret,mod,ctx);
        BN_mod_exp(mypublic,three,mysecret,mod,ctx);

        BN_rand(hissecret,1024,1,1);
        BN_nnmod(hissecret,hissecret,mod,ctx);
        BN_mod_exp(hispublic,three,hissecret,mod,ctx);
        BN_mod_exp(sharedsecret,mypublic,hissecret,mod,ctx);
          printf("his key\n");
        BN_print_fp(stdout,sharedsecret); printf("\n");

        BN_mod_exp(sharedsecret,hispublic,mysecret,mod,ctx);
         p= BN_bn2hex(sharedsecret);
        printf("my key\n %s \n",p);
         OPENSSL_free(p);
```

See ~dunigan/cns06/dhtest.c

Diffie-Hellman ’76 paper

- described a key agreement system
- described public key structure
  - private/public key (algorithms E_k and D_k)
    $E_k(D_k(m)) = D_k(E_k(m)) = m$
  - easy to compute
  - can’t determine E_k from D_k (nor reverse)
  - no need for shared secret
  - sign documents (non-repudiation)
  - avoid symmetric key problems -- scalability, secret key distribution
- suggested looking for one-way trap-door functions
  $Y = f_k(X)$ easy if know X, k
  $X = f_k^{-1}(Y)$ easy if know k and Y
  $X = f_k^{-1}(Y)$ infeasible if know only Y and not k

Public key crypto



Conventional Encryption

Needed to Work:

1. The same algorithm with the same key is used for encryption and decryption.
2. The sender and receiver must share the algorithm and the key.

Needed for Security:

1. The key must be kept secret.
2. It must be impossible or at least impractical to decipher a message if no other information is available.
3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.

Public-Key Encryption

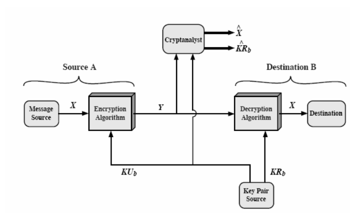
Needed to Work:

1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption.
2. The sender and receiver must each have one of the matched pair of keys (not the same one).

Needed for Security:

1. One of the two keys must be kept secret.
2. It must be impossible or at least impractical to decipher a message if no other information is available.
3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

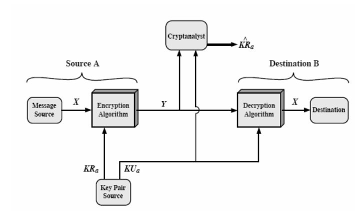
Public key secrecy



Note that you have Alice’s public key, so you can make unlimited chosen plain-text attacks to guess private key.

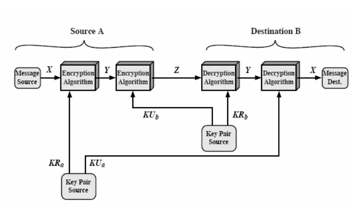
Or steal private key file and do dictionary attack on the passphrase used to encrypt the private key file.

Public key authentication



Sign with your private key, verified by receiver with your public key

Public key: sign and encrypt



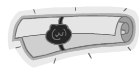
Public key can be published (emailed, LDAP, web page), solves symmetric key distribution problem!

Public key crypto applications

- **Encryption/decryption** sender encrypts message with the recipient’s public key
  Note: attacker can do unlimited chosen-plaintext attacks,
  and try various guesses at private key to decipher
- **Digital signature** sender “signs” message with his private key
  - Actually encrypts a hash (MD5/SHA) of the message
  - **Non-repudiation** only private key owner could have signed the message
- **Key exchange** two parties establish a session key

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange	Strength
RSA	Yes	Yes	Yes	Factoring
Elliptic Curve	Yes	Yes	Yes	Elliptics
Diffie-Hellman	No	No	Yes	Discrete log
DSS	No	Yes	No	Discrete log

Digital signatures



- like written signature
  - –verifies signer and date
  - –authenticate contents, tamperproof
  - –legally binding (verifiable by third party) – non-repudiation
- can encrypt whole message, hash is better
  - –smaller, faster, privacy
- can use shared key (keyed hash) -- but lacks non-repudiation

Digital signatures

- criteria
  - –signature must depend on message being signed
  - –must be unique to signer
  - –easy to generate, verify, and store
  - –infeasible to forge
    - can't construct new message for existing signature
    - can't forge signature for a new message
- sign: encrypt hash with private key
- verify: decrypt hash with public key, re-hash and compare
- legally binding (non-repudiation)
- used for authenticating messages, documents, and keys

RSA

Rivest, Shamir, Adleman

- discovered a trap-door function
- '77 MIT tech report and *Mathematical Games* in '77 Scientific American, and '78 CACM
- simple public key cryptography
- strength based on difficulty in factoring large numbers
- patented/licensed

later revealed NSA/British may have already "done that"

RSA algorithm

Security through mathematics

- public modulus n and public key e
  - n is product of two secret primes p and q, n=pq
  - e is chosen relatively prime to
    - $\phi(n) = (p-1)(q-1)$
- Private key, d
  - $d = e^{-1} \mod (p-1)(q-1)$
- encryption: $c = m^e \mod n$
- decryption: $m = c^d \mod n$

- must keep p, q, d secret

Key Generation	
Select p, q	p and q both prime, $p \quad q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calculate d	$d \equiv e^{-1} \mod \phi(n)$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Encryption	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod n$

Decryption	
Ciphertext:	C
Plaintext:	$M = C^d \pmod n$

RSA details

Key generation

- generate huge (1000+ bits) random primes p and q
- n = pq
- choose random e relatively prime to (p-1)(q-1) $\phi(n)$
- use extended Euclidean algorithm to compute d multiplicative inverse of e
  - $d \equiv e^{-1} \mod (p-1)(q-1)$
  - d is your private key (KEEP IT SECRET)
- public key: n, e

encryption $c \equiv m^e \mod n$ (message represented as n-bit number m)

decryption

$c^d \mod n = (m^e)^d = m^{ed}$

Recall $d = e^{-1} \mod (p-1)(q-1)$ so $ed = 1 \mod (p-1)(q-1)$ ➔ $ed = k\,\phi(n) + 1$

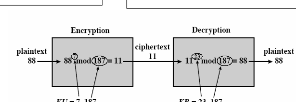
So $c^d \mod n \equiv m^{k\phi(n)+1} \equiv m m^{k\phi(n)} \equiv m$

recall Euler's theorem: $a^{\phi(n)} = 1 \mod n$

RSA examples

p=7, q=17 then n=119
find an e, relatively prime to
(p-1)(q-1)=96
e=5, calculate
$d = 5^{-1} \mod 96 = 77$
public (5,119), private (77,119)
if plaintext m=19, then
$c = 19^5 \mod 119 = 66$
decryption yields
$66^{77} \mod 119 = 19 = m$

p=47, q=71 then n=3337
find an e, relatively prime to
(p-1)(q-1)=3220
e=79, calculate (extended euclid)
$d = 79^{-1} \mod 3220 = 1019$
public (79,3337),
private (1019,3337)
say m = 688 (plaintext), then
$c = 688^{79} \mod 3337 = 1570$
to decrypt,
$1570^{1019} \mod 3337 = 688 = m$
use UNIX *bc* -- can take a while



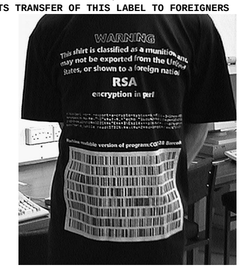
RSA software

- used in Netscape/SSL, Lotus Notes, ssh, PEM, PGP, esm
- reference implementation RSAREF 2.0 (C library)
  - RSA key generation
  - RSA sign/verify
  - RSA encrypt/decrypt
  - DES CBC
  - MD5, MD2
  - Diffie-Hellman
  - multiprecision arithmetic
  - random pool mgt (no source)
- primality
  - small prime test (3,5,7,11)
  - Fermat test $2^{p-1} = 1 \bmod p$

Crypto Toolkit
- secret-key crypto ✓
- public-key crypto
- big-number math ✓
- random numbers✓
- prime numbers ✓
- hash functions✓

RSA in perl

```
        WARNING -- THIS LABEL IS CLASSIFIED AS A MUNITION
              RSA IN THREE LINES OF PERL
HAVE YOU EXPORTED A CRYPTO SYSTEM TODAY? --> http://www.cypherspace.org/~adam/rsa/

 #!/usr/local/bin/perl -s-- -export-a-crypto-system-sig -RSA-in-3-lines-PERL
  ($k,$n)=@ARGV;$m=unpack(H.$w,$m."\0"x$w),$_=`echo "16do$w 2+4Oi0$d*-^1[d2%
  Sa2/d0<X+d*La1=z\U$n%0]SX$k"[$m*]\EszlXx++p|dc`,s/^.|\W//g,print pack('H*'
 ,$_)while read(STDIN,$m,($w=2*$d-1+length($n||die"$0 [-d] k n\n")&~1)/2)

TRY: echo squeamish ossifrage | rsa -e 3 7537d365 | rsa -d 4e243e33 7537d365
            FEDERAL LAW PROHIBITS TRANSFER OF THIS LABEL TO FOREIGNERS
```



RSATest.java

```java
import java.io.*;
import java.security.*;
import java.math.BigInteger;

public class RSATest {
    public static void main(String[] args) throws IOException {
    BigInteger n,d,e,m,c,m1;

// rsa pub (e) and private (d) keys mod n
  n  = new BigInteger(
  "12467899317123168996938478195368007977523029551389083332205993261431214145017901246238
  8955199881800247896622113260636305952118642184220610072971688712560157547953370625529343
  8031445235671966391197094158868849414392581683905371530307896151726649818252362048413805
  6115643133052630577199833387088308265430407691");
  e = BigInteger.valueOf(37L);
  d = new
  BigInteger("1516366133163628661789814915652865835104152242736510135121424242368758009871
  8393394812838644322886759771715674350467198315633875377916196608995601619477080259016270
  1500942793034560109614997119625041170116994399696468106879360767967276183442800667287336
  90010290995392440062128773122006698090773675868324510565");


   m = new BigInteger(1024, new SecureRandom());
   m = m.mod(n);
   c = m.modPow(e,n);

   m1 = c.modPow(d,n);
   System.out.println("msg m " + m.toString());
   System.out.println("msg m1 " + m1.toString());
    }
}
```


RSA with OpenSSL API

```c
#include <stdio.h>
#include <openssl/rsa.h>
#include <openssl/objects.h>

static const unsigned char tmp16[16]=
               {0x12,0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,
                0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,0x12};

main()
{
        RSA *rsa;
        int res,lth;
        char *to, buff[4096];

        rsa = RSA_generate_key(1024,RSA_3,NULL,NULL);
        to = malloc(RSA_size(rsa));
        lth = RSA_public_encrypt(sizeof(tmp16),tmp16,to,rsa,RSA_PKCS1_PADDING);
        printf("lth %d\n",lth);
        lth = RSA_private_decrypt(lth,to,buff,rsa,RSA_PKCS1_PADDING);
        printf("lth %d %d\n",lth,memcmp(buff,tmp16,sizeof(tmp16)));

        RSA_sign(NID_md5,tmp16,sizeof(tmp16),to,&lth,rsa);
        printf("lth %d\n",lth);
        res=RSA_verify(NID_md5,tmp16,sizeof(tmp16),to,lth,rsa);
        printf("res %d\n",res);
}
```


OpenSSL RSA

- The RSA key struct

```c
struct RSA
       {
       BIGNUM *n;              // public modulus
       BIGNUM *e;              // public exponent
       BIGNUM *d;              // private exponent
       BIGNUM *p;              // secret prime factor
       BIGNUM *q;              // secret prime factor
       BIGNUM *dmp1;           // d mod (p-1)
       BIGNUM *dmq1;           // d mod (q-1)
       BIGNUM *iqmp;           // q^-1 mod p
       // ...
       };
```

- RSA from the openssl command line

  genrsa -- generate RSA key (**genrsa –des3 –out ca.key 1024**)
  rsautil -- encrypt/decrypt sign/verify

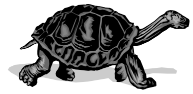
RSA performance

- key generation slow (e.g., PGP)
- encryption/decryption 1000 times slower than DES
  300 bytes/sec (see OpenSSL speed command)
- e=3 will speed up software
- hardware: 10 Mb/sec

- Use RSA to encrypt session key, then use DES or AES
- Use RSA to encrypt hash of message (signature)

  both also result in a plaintext (m) less than n bits $c = m^e \bmod n$

RSA optimizations



for 512-bit numbers and larger, slow
- speed up for exponentiation (square and multiply)
- e often set to 3 or 65537 (pub key fast)
- keep p and q with d, do computations mod p and mod q, use Chinese Remainder Theorem to compute answer mod n
- Calculate the following and keep (secret) with d
  - d mod (p-1)
  - d mod (q-1)
  - p^{-1} mod q
  - q^{-1} mod p

Chinese Remainder Theorem (CRT)

Allows us to manipulate BIG numbers in terms of tuples of smaller numbers (➔ faster). Handy in RSA with private key calculations where we know p and q

$n = pq$, want to calculate $b = a^k \bmod n$

precalculate $z_0 = q^{-1} \bmod p$ $\quad z_1 = p^{-1} \bmod q$ $\quad c_0 = qz_0 \bmod n$ $\quad c_1 = pz_1 \bmod n$

$b_0 = a^k \bmod p$

$b_1 = a^k \bmod q$

$b = (b_0 c_0 + b_1 c_1) \bmod n$

```
Example:  calculate (678 + 973) mod 1813 using CRT
n= pq   n=1813 = 37*49   p=37 q=49  (p and q just need to be relatively
  prime for CRT to work)
973 mod 1813 equivalent to (973 mod 37, 973 mod 49) = (11,42)
678 mod 1813 equivalent to (678 mod 37, 678 mod 49) = (12,41)
So: (11,42) + (12,41) = (23,34)

To verify, convert back, need (multiplicative inverse)
u = p-1 mod q = 34  and v= q-1 mod p = 4
(a,b) = ( auq + bvp ) mod n
(23,34) =(23*34*49 + 34*4*37 ) mod 1813
        = (245 + 1406) mod 1813
        = 1651
```


Number theory – hard problems

factoring a number
- simple, but time consuming
- use quadratic sieve or number field sieve
- complexity $e^{(1.9 \ln(n))^{\{2/3\}}}$
- strength of RSA
- RSA factoring challenge -- 174-digit factored

discrete log

- find x where $a^x = b \bmod p$
- basis of ElGamal, DSS, Diffie-Hellman
- if you can solve discrete log, you can factor
- complexity $e^{((\ln p)^{\{1/3\}}(\ln(\ln p))^{\{2/3\}})}$

square roots mod n

$y = x^2 \bmod n$ given y, find x

easy if you know prime factors p and q where n = pq

someone still might find a better way ...

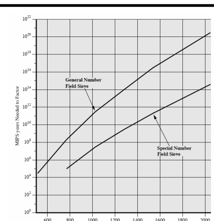
Computational complexity big-O

- Time complexity of an algorithm
- Number of operations as a function of size of input
  - Matrix multiply for NxN matrix is $O(n^2)$
  - Brute force of DES is $O(2^{56})$
- Algorithm with input size n is:
  - Linear if running time is $O(n)$
  - Polynomial if $O(n^k)$
  - Exponential if $O(k^{h(n)})$
  - computationally feasible if polynomial or linear or if n is small

Table 9.4 Level of Effort for Various Levels of Complexity

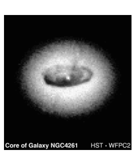
Complexity	Size	Operations
$\log_2 n$	$2^{10^{12}} = 10^{3\times10^{11}}$	10^{12}
n	10^{12}	10^{12}
n^2	10^6	10^{12}
n^6	10^2	10^{12}
2^n	39	10^{12}
$n!$	15	10^{12}

RSA strength



factoring n

- operations $e^{\mathrm{sqrt}\{\ln(n)\ln(\ln(n))\}}$
- Time
  - 200 digit n, 665 bits, 10^{23} ops @ 10^9 ops/sec, 380,627 years
  - 400 digits, 1400 bits would take 10^{15} years -- longer than age of universe
- Space
  - precalculate factors of all 200 digit numbers
  - $9 \times 10^{200} \times 665$ bits
  - store on 100 GB drives, each weighing one millionth of a gram
  - weight: 10^{177} tons
    earth: 10^{21} tons, black hole 10^{27}



RSA vulnerabilities

- finding factors of n, or computing ϕ(n), or finding d

Number of Decimal Digits	Approximate Number of Bits	Date Achieved	MIPS-years	Algorithm
100	332	April 1991	7	quadratic sieve
110	365	April 1992	75	quadratic sieve
120	398	June 1993	830	quadratic sieve
129	428	April 1994	5000	quadratic sieve
130	431	April 1996	1000	generalized number field sieve
140	465	February 1999	2000	generalized number field sieve
155	512	August 1999	8000	generalized number field sieve
160	530	April 2003		lattice seive
200	663	May,2005		lattice sieve

- guess
- off-line dictionary attacks (steal your key file, guess your passphrase)
  - keyboard/net sniffer
  - rubber-hose
- brute force
  easy to generate messages with someone's public key, then try different d to get back plain text
- some other way ? new math?

crypto-card best protection (USB, PCMCIA, smartcard)
keys and crypto software on the card
do sign/encrypt via card API

Timing attacks

- note that big integer multiplies take a long time
- if calculating c^d mod n, then some modular multiplications may take longer than others, depending on whether a bit in d is set or not
- need a hi-resolution timer and lots of samples
- discover secret d bit by bit

countermeasures

- quantize all operations (cryptolib)
- add random delay
- blinding

  instead of $m = c^d$ **mod** n   generate a random r and calculate

  $m = r^{-1} (cr^e)^d$ **mod** n

The RSA API's usually pad the message before encrypting to randomize the ciphertext and limit chosen-cipher text attacks (see OAEP) and PKCS standards.

ElGamal

- 1985 public-key scheme, based on discrete log
- like D-H,   public: generator g, prime p
- user A picks random X_A as private key

  calculate public key $Y_A = g^{X_A} \bmod p$
- to encrypt M for user B (public key Y_B)

  pick random k mod p

  encrypt M as a pair (C_1, C_2)

  $C_1 = g^k \bmod p$

  $C_2 = Y_B^k M \bmod p$
- B decrypts with

  computes $C_1^{p-1-X_B} = C_1^{-X_B} \bmod p$

  $M = C_2 C_1^{-X_B} \bmod p$

  because $C_2 C_1^{-X_B} = g^{kX_B} M g^{-kX_B}$

- requires random number for each encryption
- requires 2 exponentiations for each encryption
- message expansion (cipher text twice as big)
- DSS is a variant

ElGamal example

```
Using bc,  given p = 2357, g=2, B's private key = 1751     X_B

2^1751%2357
1185     B's public key         Y_B = g^X_B mod p

want to encrypt m=2035 using  random k=1520
2^1520%2357
1430    C_1 =  g^k

2035*1185^1520%2357  m * y_B^k mod p
697     C_2

send 1430 697   (C_1  C_2 ciphertext)

B decrypts, p-1-1751 = 605
1430^605%2357
872              C_1^-X_B  mod p

697*872%2357          M = C_2 C_1^-X_B mod p
2035    m
```

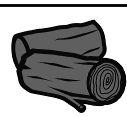

DSA

DSS   Digital Signature Standard (FIPS 186, 1994)

- in '82 US solicited public-key algorithms for a standard
- in '91 NIST proposed DSA for DSS
- controversy
  - many companies had licensed RSA
  - RSA de facto standard
  - slower than RSA
  - trap-door concern
  - not as well tested as RSA (test of time)
  - modulus too small (512) -- expanded later
  - requires unique secret for each message
  - only signature (can't do encryption) ➔ exportable ☺

- Gov't approved: DES, SHA, DSA, AES
- FIPS 186-2 (2000) digital signatures (DSS, RSA, ECC)

DSA details



- one of many discrete log signature schemes
- based on ElGamal ('85) and Schnorr ('89)
- strength based on discrete logs (like D-H)
- could be subject to Schnorr patent infringement
- uses hash function H(m), SHA in standard
- not intuitive like RSA

ElGamal signatures

- choose prime p and generator g
- generate random secret x, calculate public key $y = g^x \bmod p$
- public: p, g, y
- h = hash of message
- generate another random secret k relatively prime to p-1 (unique for each message)
- signature is (r,s)

  $r = g^k \bmod p$

  $s = (h - xr)k^{-1} \bmod (p-1)$
- Verify: calculate hash h, check if $y^r r^s = g^h \bmod p$ $y^r r^s = g^{xr} g^{k(h-xr)k\hat{}-1}$
- Note
  - similarity to D-H
  - r independent of message/hash
  - doesn't recover hash, just verifies
  - must have different random k for each message
  - calculating k^{-1} **mod** p   slow, but can pre-calculate

DSA algorithm

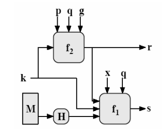
```
public p, q, g, y

generate prime p (512-1024 bits)
find q a prime factor of p-1 (160 bits)
find g = h^((p-1)/q) mod p  where  h < p and  h^((p-1)/q) mod p > 1
Generate random x, publish  y = g^x  mod p
        private: x < q (160 bits)
```

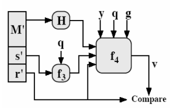
signing (signature (r,s))

```
     generate random k < q  (secret)
     r = (g^k mod p) mod q
     s = (k^-1 (H(m) + xr)) mod q
```

verifying (OK if v = r)

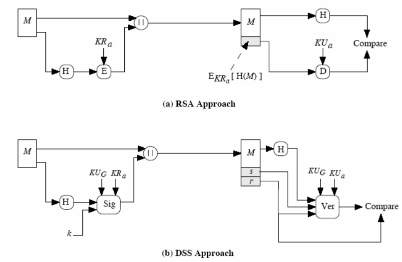
```
     w = s^-1 mod q
     a = (H(m)w) mod q
     b = (rw) mod q
     v = ((g^a y^b) mod p) mod q
```






```
verifies hash H(m) -- doesn't recover hash
```


RSA vs DSA signatures



DSA performance

- signature verify is hundred times slower than RSA
- faster than ElGamal since q is smaller than p
- calculating multiplicative inverses slow
- though can pre-calculate some things, r doesn't depend on message at all
- good random number generator is essential for k
- if Mallory ever figures out a k, she can recover the private key x
- Can you show what Eve can deduce if the same k is used for two different messages?

Time-stamp services

digital notary

- digital signatures assures who and what, but not when
- signed documents should contain "date"
  but signer could alter time before signing
- need trusted third party
- for privacy, submit hash (actually signed hash) to third party
- third party "publishes" hash log

- PGP time-stamp service
  - email document to service
  - service returns date/seqno signature
  - service publishes seqno/date

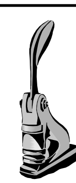
example

```
pgp -sba tst.c

verify: pgp tst.c.asc

mail pgp@stamper.itconsult.co.uk < tst.c.asc
```

get back new tst.c.asc (detached sig. file)

```
-----BEGIN PGP MESSAGE-----
Version: 2.6.3i
Comment: Stamper Reference Id: 0028848

iQEVAgUAOMPQC4GVnbVwth+BAQF0/wf6AhL1Jvm9Tb54VRs6RKmf16rs1ugqkeEO
oET6S7/cVAz2l2xKdrtrFenv2hksM1sHrKvF5yCECbwqQaLjVHc7fgvAg+/y9VsM
18aj3rzG0qAF3GGkWehjRIKdAVBczcVEHrkX/WfflLgU81+V1qLEfzwT+VkmuwRQ
bZOqphYKjx0geD//o2zBu2JQTFatb7TCdNEhgCEHDSxxMFCmcy1MOneC9veWmUOi
qmHiyBCx5XPXdohuoFaGOJx4CHTmHcaKumWydGkEXzgOknuNwJh1nlyLWHn01O3E
=KyBO
-----END PGP MESSAGE-----

pgp tst.c.asc   (verify)

File has signature.  Public key is required to check signature. .
Good signature from user "Timestamp Service <stamper@itconsult.co.uk>".
Signature made 2000/03/06 15:35 GMT
File has signature.  Public key is required to check signature.
File 'tst.c.01' has signature, but with no text.
Text is assumed to be in file 'tst.c'.
.
Good signature from user "Tom Dunigan <thd@ornl.gov>".
Signature made 2000/03/06 15:11 GMT
```


Digital notary



surety.com/RSA

- time-stamping service
- hash chaining with binary tree
- can't post-date or pre-date
- algorithm (patented)
  - user sends hash (288-bit) of document
  - service combines hash with other recent hashes
  - each minute creates superhash of aggregate hash with previous superhash
  - user gets certificate of hash path
  - service publishes time and superhashes each week in New York Times
  - third party can verify hash and superhash
- Digital notaries handy for business e documents, also for cyber forensics (hash/sign and time-notarize digital evidence, e.g. disk images, logs)

Next time …

ECC

PKCS

ssh and pgp