CNS Lecture 8

Security through mathematics
Number theory
Diffie-Hellman
Public key crypto
RSA
El Gamal
DSA
Assignments
midterm cs614 paper

Cryptography
• Classical + Classical + Classical + Classical + stego stego stego stego
• Viruses + Viruses + Viruses + Viruses

In the news
• Dept of Commerce hack into multiple times (root kits) by Chinese hackers – going to replace all hardware and software!
• McAfee spdy buffer overflow
• ADL’s “you’ve got pictures” buffer overflow
• 120 new vulnerabilities reported this week (BANS)

You are here ...

Attacks & Defenses
• Risk assessment✓
• Viruses✓
• Link security✓
• Authentication✓
• Network security
Firewalls, syms, IDS ✓
• Forensics✓

Cryptography
• Random numbers
• Hash functions
• Classical + stego✓
• Symmetric key✓
• Public key✓

Applied crypto
• S/SH
• PGP
• S/MIME
• SSL
• Kerberos
• IPsec✓

RSA, DSA, D-H, ECC

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, RSA, 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)

data

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)

\[ a = p_1^{e_1} p_2^{e_2} \ldots p_n^{e_n} \]

3600 = 2^3 \times 3 \times 5^2
Modular arithmetic

- $\mod$ (congruence), remainder after dividing
- if $a \mod n = b \mod n$, then $a = km + b$
- $11 \mod 3 = 2 \mod 3 = 2$
- modular arithmetic ($\mod$) on non-negative integers $\mathbb{Z}_n$
  - associative: $(a + b) + c \mod n = a + (b + c) \mod n$
  - commutative: $a + b \mod n = b + a \mod n$
  - distributive: $(a + b) \mod n = (a \mod n) + (b \mod n)$
  - reducible: $a \mod n = (a \mod b) \mod b \mod n$
- commutative ring
- additive identity: $0 + a = a$
- additive inverse: $(5 + 7) + (5 \mod 7) = 0$

Let us work with smaller numbers

Euler’s totient

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

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

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

What is $\phi(n)$ if $m = np$, and $p$ and $q$ are prime?

- have to factor $n$ to calculate totient

$\phi(n) = \phi(p) \phi(q) = (p-1)(q-1)$

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

$a^{\phi(n)} = 10 \mod 11; a^{10} = 1 \mod 11$

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 $gcd = 1$
- extended $gcd$ can be used to find multiplicative inverse (if it exists)

Used in IDEA and CRT for RSA

see gcd.c (extended euclid)

Table 5.1: Some Values of Euler’s Totient Function $\phi(n)$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\phi(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Multiplication

- multiplicative inverse $(a^{-1} \mod n)$
- $3 \times 9 \mod 26 = 1$
- $4 \times 9 \mod 17 = 1$
- only numbers relatively prime to $n$ have multiplicative inverse ($\mathbb{Z}_n$)
- relatively prime $\Rightarrow$ don’t share any common factors
- if $p$ is prime, all elements have multiplicative inverse (except 0)

$\mathbb{Z}_p$ is a field

Modular shortcuts

- $ab \mod n = ((a \mod n)((b \mod n)) \mod n$
- $a \times a \mod n = ((a^2 \mod n) \times a \mod n)$
  - faster: $a^{30} \mod n = ((a^2 \mod n)^2 \times a^2 \mod n)$

- Chinese Remainder Theorem (CRT)
  - do arithmetic mod factors of $n$ (faster)
  - handy in RSA (example later)

Primes

Prime number theorem:
- number of primes $\leq n$ asymptotic to $n/\ln n$
- primes $< 1,000,000,000 = 78,498$
- primes $< 2^{32} = 10^{10} -- not that many atoms in the universe

Generating primes
- need $p$ and $q$ large primes for key generation (RSA, DSS, D-I)
- 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 integers.
- Can't determine absolutely if big number is prime, but can test for "highly probable".

- **Format's theorem**
  - If $p$ is prime and $a$ is not divisible by $p$ (relatively prime) then $a^{p-1} \equiv 1 \mod p$.

- **Lehman variation** of Format's theorem
  - Choose random $a$, if $p$ is prime, $a^{(p-1)/2} \not\equiv \pm 1 \mod p$ with high probability.

- **Rivest variation** of Format's theorem
  - If $p$ is prime but there are pseudo-primes in that test, for 256-bit number $(2^{256})$, test $10^{17}$ primes and $10^{25}$ pseudo-primes.

- **Miller-Rabin**
  - If there is a solution to $a^2 \equiv 1 \mod p$, other than 1 and $p-1$, then $p$ is NOT prime.

- **Probability** $p$ is NOT prime after $k$ successful tests: $\left(\frac{1}{4}\right)^k$.


### 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 (775) 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.
- Doesn't do encryption or signatures.

**Algorithm**

- Select $g$ and prime module $q$.
- Alice: generates $a$ and calculates $A \equiv g^a \mod q$.
- Bob: generates $b$ and calculates $B \equiv g^b \mod q$.
- Two exchange $A$ and $B$ (public keys).
- They calculate $X^y \equiv B^a \equiv g^{ab} \equiv g^{xy}$ and $Y^x \equiv A^b \equiv g^{ab} \equiv g^{xy}$ to get $k \equiv g^{xy} \equiv g^{ab} \equiv g^{ab}$.
- Now they can use $k$ for DES or whatever.

### Primitive roots/generators

- **Primitive roots of prime number, 19**
  - There should be $\phi(19) = 18$ of them.
  - Roots are 2, 3, 5, 10, 13, 14, and 15.
  - When doing exponentiation mod $p$, using a primitive root insures full period (D-H, ElGamal).

### D-H examples

- $p = 11$ and $g = 2$.
- Alice chooses random $x = 4$, Bob $y = 6$.
- Alice $X = 2^4 \mod 11 = 5$.
- Bob $Y = 2^6 \mod 11 = 9$.
- Exchange and $9g = 6561 \equiv 5 \mod 11$.
- Bob $X^y \mod 11 = 5^6 = 15625 \equiv 5 \mod 11$.
- $k = 5$. 

### Finding a prime

1. **Generate random n-bit number $p$**
2. **Set low (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.**

### Density of primes:
- **Proportion of positive integers $x$ that are prime is roughly $2/e \ln n$**.
- **For 512-bit $n$ (2^512) can find a prime in 177 tries**.
- **This is what takes time when you first generate your RSA key**.

### UNIX example with bc using primes 5, 13, 17

- $(5 \cdot 13) \cdot 17 = 1155$
- $(5 \cdot 13) \cdot 17 = 6561$
- $(5 \cdot 13) \cdot 17 = 15625$
- $(5 \cdot 13) \cdot 17 = 25503$
D-H uses

- Used to generate session keys between two parties in
  - Netscape (SSL)
  - Secure PVM
  - ssh (secure TELNET)
  - SKIP, ISAKMP, GREMP
  - Cisco encrypting routers
  - nautilus
  - Sun secure RPC (keyserver)

  - 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 $p$ big (1024 bits) prime $p$
- $x$ and $g$ is also prime (advisory prime)
- find $g$ that is generator $\mod p$, that is, $g^a \mod p$ will generate all elements $1$ to $p-1$
- $g^a \mod p$, there are ways to find $g$
- you can use published $p$, $g$
- need good random number generator (big integers) for secrets $x$ and $y$
- good 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 $K_{AB}$ and $K_{AT}$
Trudy can decrypt, re-encrypt and relay, or make changed loss of privacy and integrity

Implementing D-H

- need random numbers
- may also need to find prime and generator (use known ones)
- use big prime (1024 bits)
- need multiprecision integer library (UNIX mp, -xlf, GNU gmp)
  see example.d, ext.c
  or perl bignum, or C++ Integer, or Java BigInteger

countermeasures

- shared secret (STEL, _stel_secret), e.g. use it to encrypt $K$ in a message
- sign key exchange with private key
  (SKMP, ISAKMP, Cisco’s encrypting routers), use RSA or DSA
- Bellavw-Merritt encrypt D-H
  exchange with shared secret (EKE)
- SPEKE
- Assignment 8

D-H depends on:
- prime numbers
  may also need to find prime and generator (use known ones)
- Trudy

big integer arithmetic

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

UNIX bc command

```
perl -e '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;
```

Java

```
import java.math.BigInteger;
```

BIG integer software (C)

GNU's MP library

```c
#include "gmp.h"

void mpz_init (mpz_t integer)
void mpz_set_ui (mpz_t rop, unsigned long int op)
int mpz_mod (mpz_t rop, 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_mod (BIGNUM *r, BIGNUM *a, BIGNUM *b, BIGNUM *c);
int BN_modl (BIGNUM *r, BIGNUM *a, BIGNUM *b, const BIGNUM *c);
int BN_rand (BIGNUM *r, int bits, int top, int bottom);
BIGNUM * BN_generate_prime (BIGNUM *q, int flag, int bits);
void * BN_num_bytes (BIGNUM *n, int *num);
```

Native assembly language can speed these up

Diffie Hellman example (OpenSSL lib)

```c
/* gcc -o dhtest dhtest.c -lssl */
```

```
BN_set_word(three,3);
BN_hex2bn(&mod,defaultmodulostring);
BN_print_fp(stdout,mod);
printf("n");
BN_set_word(mod,three);
BN_mod_exp(mod,three,mysecret,mod,ctx);
BN_set_word(three,mypublic);
BN_mod_exp(three,mypublic,three,mysecret,mod,ctx);
```

See ~dunigan/cns06/dhtest.c
Diffie-Hellman '76 paper

- **described a key agreement system**
- **described public key structure**
  - private/public key (algorithms $E$, $D$)
  - $E(D(m)) = D(E(m)) = m$
  - easy to compute
  - can't determine $E_m$ from $D$ (non-reversal)
  - 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**
  - $Y = f(x)$ easy if know $x$, $k$
  - $X = f(Y)$ easy if know $k$ and $Y$
  - $X = f(Y)$ infeasible if know only $Y$ and not $k$

---

Public key secrecy

Note that you have Alice's public key, so you can make unlimited chosen plaintext 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 crypto applications

- **Encryption/decryption** sender encrypts message with the recipient's public key.
  - Note that attacker can do unlimited chosen-plaintext attacks, and try various guesses at private key to decipher

- **Digital signature** sender (Alice) signs message with her private key.
  - Actually encrypts a hash (MD5/SHA) of the message
  - **Non-repudiation** only she or she alone could have signed the message

- **Key exchange** two parties establish a session key

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Encryption/Decryption</th>
<th>Digital Signature</th>
<th>Key Exchange</th>
<th>Non-repudiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Elliptic Curves</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Diffie-Hellman</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DSS</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

---

Public key authentication

Sign with your private key, verified by receiver with your public key.
Digital signatures

- like written signature
  - verifies signer and date
  - authenticates content, tamperproof
  - legally binding (verifiable by third party) -- non-repudiation
- can encrypt whole message, hash is better
  - smaller, faster, privacy
- can use shared key (byted 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 module \( n \) and public key \( e \)
  - 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

RSA details

Key generation

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

Encryption \( c = m^e \mod n \) (message represented as \( n \) bit number \( m \))

Decryption \( c^d \mod n = (m^e)^d \mod n = m \)

Recall \( a \mod b = a - \lfloor a/b \rfloor b \)

RSA examples

- \( p=47, \, q=71 \) then \( n=3357 \)
- find an \( e \) relatively prime to \( (p-1)(q-1)=3220 \)
- \( e = 5 \) calculate \( d = 5^{-1}\mod 3220 \)
- public \( (79,3357) \), private \( (1019,3337) \)
- \( c = 668^{3357} \mod 3357 = 1570 \)
- to decrypt:
  - \( 1570\mod 3357 = 668 \) (plaintext), then
  - \( c = 668^{1019} \mod 3337 = 1570 \)

Recall Euler's theorem: \( a^{\phi(n)} \mod n = 1 \)
**RSA software**

- used in Netscape/SSL, Lotus Notes, ssh, PEM, PGP, emacs
- reference implementation RSAKEF 2.0 (C library)
  - RSA key generation
  - RSA sign/verify
  - DSS CBC
  - DSS, MD2
  - Diffie-Hellman
  - multiprecision arithmetic
  - random pool mgt. (no source)
- primality
  - small prime test (3, 5, 7, 11)
  - Fermat test $2^m \equiv 1 \mod p$

**RSA in perl**

```perl
use strict; use warnings;

# RSA key struct

struct RSA {
    # public module
    BIGNUM n;
    BIGNUM e;
    BIGNUM d;
    # public exponent
    BIGNUM P;
    # secret prime factor
    BIGNUM Q;
    # secret prime factor
    BIGNUM D;
    # n mod (P-1)
    # n mod (Q-1)
    # n mod P
    # n mod Q

    # RSA key generation
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # RSA sign/verify
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # RSA encryption/decryption
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # Diffie-Hellman
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # multiprecision arithmetic
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # random pool mgt.
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # Diffie-Hellman
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    # RSA performance
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)
```

**OpenSSL RSA**

- The RSA key struct

```c
struct RSA {
    BIGNUM n;  // public module
    BIGNUM e;  // public exponent
    BIGNUM d;  // secret prime factor
    BIGNUM p;  // secret prime factor
    BIGNUM q;  // secret prime factor
    BIGNUM dp;  // n mod (P-1)
    BIGNUM dq;  // n mod (Q-1)
    BIGNUM invq;  // n mod 2

    // RSA key generation
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // RSA sign/verify
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // RSA encryption/decryption
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // Diffie-Hellman
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // multiprecision arithmetic
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // random pool mgt.
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)

    // Diffie-Hellman
    RSAooky(key, modulus, prime, factor, exponent)
    RSAooky(key, modulus, prime, factor, exponent)
```

**RSA with OpenSSL API**

```c
#include <openssl/bio.h>
#include <openssl/obj_mac.h>

#include <openssl/x509.h>

RSA *RSA_new;
RSA *RSA_generate_key_ex(BIGNUM *n, BIGNUM *e, BIGNUM *dh);
RSA *RSA_public_encrypt(const unsigned char *t, size_t tlen, RSA *rsa, const unsigned char *m, size_t mlen);
RSA *RSA_private_decrypt(const unsigned char *c, size_t clen, RSA *rsa, BIGNUM *p, size_t plen);

// RSA key generation
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// RSA sign/verify
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// RSA encryption/decryption
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// Diffie-Hellman
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// multiprecision arithmetic
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// random pool mgt.
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)

// Diffie-Hellman
RSAooky(RSA* rsa, RSA* key, RSA* modulus, RSA* prime, RSA* factor, RSA* exponent)
```
RSA optimizations

- For 512-bit numbers and larger, slow
- Speed up for exponentiation (square and multiply)
- A 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 (\( \bigotimes \) faster). Handy in RSA with private key calculations where we know \( p \) and \( q \).
- If \( m \equiv n \mod p \) and \( m \equiv q \mod p \), then \( m \equiv (n \cdot q^{-1} \mod p) \mod p \).
- Example: Calculate \((768 \cdot 973) \mod 1613\) using CRT.
- \( \text{gcd}(768, 1613) = 1 \), so \( 768 \cdot 973 \equiv 973 \mod 1613 \) equivalent to \((768 \cdot 973) \mod 1613 = (768 \cdot 973) \).

Number theory – hard problems

**Factoring a number**
- Simple, but time consuming
- Use quadratic sieve or number field sieve
- Complexity \( \exp((C_{quartic}) \sqrt{n}) \)
- Strength of RSA
- RSA factoring challenge – 174-digit factored

**Discrete log**
- Find \( x \) where \( y \equiv a^x \mod p \)
- Base of big primes, DSS, Diffie-Hellman
- If you can solve discrete log, you can factor
- Complexity \( \exp((C_{quartic}) \sqrt{n}) \)

**Square roots mod n**
- \( y \equiv x^2 \mod n \)
- Easy if you know prime factors \( p \) and \( q \) where \( n = p \cdot q \)
- Someone else 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 \( N \times N \) matrix is \( O(n^3) \)
  - Brute force of DES is \( O(2^{56}) \)
- Algorithm with input size \( n \):
  - Linear if running time is \( O(n) \)
  - Polynomial if \( O(n^k) \)
  - Exponential if \( O(2^n) \)
  - Computationally feasible if polynomial or linear or if \( n \) is small

RSA strength

- Factoring \( n \)
- \( \text{operations} \xrightarrow{\text{very long}} \)
- \( 1 \times 200 \text{ digit n}, 665 \text{ bits}, 10^{12} \text{ ops} @ 10^7 \text{ ops/sec,} \)
- \( 500,000,000 \text{ years} \)
- \( 400 \text{ digits}, 1400 \text{ bits would take} \)
- \( 10^{127} \text{ years} \)
- Space:
  - Precalculates factors of all 200 digit numbers
  - \( \times 10^{127} \times 665 \text{ bits} \)
  - Store on 100 GB drives, each weighing one millionth of a gram
  - Weighs: \( 10^{127} \) tons
  - Earth: \( 10^{30} \) tons

RSA vulnerabilities

- Finding factors of \( n \), or computing \( d \), or finding \( d \)
- Brute force
  - Easy to generate messages with someone's public key, then try different \( d \) to get back plain text
  - Some other prop? New math?
- Crypto-card theft protection (USB, PCKMA, smartcard)
- Keys and crypto software on the card
- Do not sign/encrypt via card API
Timing attacks

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

**countermeasures**
- quantize all operations (cryptoki)
- add random delay
- blinding
  instead of $m = c^r \mod n$, generate a random $r$ and calculate $m = r^x \cdot (c^r)^y \mod n$

The RSA APIs usually pad the message before encrypting to randomize the ciphertext and limit chosen-plaintext 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 $x$, private key calculate public key $y = g^x \mod p$
- to encrypt $m$ for user $B$ (public key $y_B$)
  - pick random $k \mod p$
  - encrypt $m$ as a pair $(C_1, C_2)$
    
  $C_1 = g^k \mod p$
  $C_2 = m \cdot y_B^k \mod p$
- $B$ decrypts with
  - computes $C_1^y \cdot C_2 = g^k \cdot m \cdot y_B^k \mod p$
  - modulus too small (512) -- expanded later
- requires random number for each encryption
- requires 2 exponentiations for each encryption
- message expansion (cipher text twice as big)
- **DSS** is a variant:

### DSA

**DSA**

- Digital Signature Standard (FIPS 186, 1994)
- in 1991 US adopted public-key algorithms as a standard
- in 1993 NIST proposed DSA for DSS
- controversy
  - many companies had licensed RSA
  - RSA gave free 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) • expensive to create
- didn't approve DSA, SHA, DSS, AIE
- FIPS 186-2 (2000) digital signatures (DSS, RSA, ECC)

### ElGamal example

- $p = 2357$, $q = 2$, $n$'s private key $= 1751$ $X_B$
- $2^{1751} \mod 2357 = 1430$ $n$'s public key $y = 2^{1751} \mod p$
- want to encrypt a message using random $k = 1520$
- $2^{1520} \mod 2357 = 1185$ $m = 2035$
- $1185^1 \cdot 1520 \mod 2357 = m \cdot y^k \mod p$
- $697 \cdot C_1 = 697 \cdot y_B^k \mod p$
- send $697 \cdot C_1$ ($C_1$, ciphertext)

#### 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

<table>
<thead>
<tr>
<th>ElGamal signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>choose prime $p$ and generator $g$</td>
</tr>
<tr>
<td>generate random secret $x$, calculate public key $= g^x \mod p$</td>
</tr>
<tr>
<td>public $p$, $g$, $y$</td>
</tr>
<tr>
<td>$h$ a hash of message</td>
</tr>
<tr>
<td>generate another random secret $k$ relatively prime to $p-1$ (unique for each message)</td>
</tr>
<tr>
<td>signature is $(r,s)$</td>
</tr>
<tr>
<td>$r = g^k \mod p$</td>
</tr>
<tr>
<td>$s = (h \cdot r \cdot y^{-1}) \mod (p-1)$</td>
</tr>
<tr>
<td>verify calculated hash, check if $y^s \cdot g^r \mod p$</td>
</tr>
<tr>
<td>note</td>
</tr>
<tr>
<td>independently of message/hash</td>
</tr>
<tr>
<td>doesn't recover hash, just verifies</td>
</tr>
<tr>
<td>must have different random $k$ for each message</td>
</tr>
<tr>
<td>calculating $k^*$ for $k_1$, should be pre-calculated</td>
</tr>
</tbody>
</table>
**DSA algorithm**

```
public p, q, g, h

generate primes p (1536-1024 bits)
find q prime factor of p - 1 (160 bits)
find a prime p and b = \( h^2 \mod p \) and a = \( h^2 \mod q \)

generate random r, compute g^r \mod p and g^r \mod q

private: a \( \in q \) (160 bits)

signing (signature, x, a)
generate random b, r, q (secret)
x = (g^r \mod p) \mod q

verifying (x, a, r, q)

- if x = r

  a = \( g^r \mod q \)

  b = \( (x - a \cdot r) \mod q \)

  verifys hash H(x) -- doesn't recover hash

-----BEGIN PGP MESSAGE-----
Version: 2.6.3i
Comment: Stamper

RefId: 0028848... to be in file 'tst.c'.

Good signature from user "Tom Dunigan <thd@ornl.gov>".
Signature made 2000/03/06 15:11 GMT

-----END PGP MESSAGE-----
```

**RSA vs DSA signatures**

```
\( \frac{p - 1}{q} \)
```

**DSA performance**

- Signature verify is hundreds of 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 \( a \) or \( k \), she can recover the private key \( x \)
- Can you show what Eve can deduce if the same \( k \) is used for two different messages?

**Example**

```
ping -t x x

verify: ping x x

mail ping霰erochastic x x x

get back new x x.asc (detached sig file)

-----BEGIN PGP SIGNED MESSAGE-----
Version: 2.6.3i
Comment: Stamper

RefId: 0028848... to be in file 'tst.c.asc'.

Good signature from user "Tom Dunigan <thd@ornl.gov>".
Signature made 2000/03/06 15:11 GMT

-----END PGP SIGNED MESSAGE-----
```

**Time-stamp services**

- Digital notary
  - Digital signatures assures who and what, but not when
  - Signed documents should contain "date"/"time"
  - But signer could alter time before signing
  - Need trusted blind 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/seal (signature)
  - Service publishes secpub.dat

**Digital notary**

- surety.com/RSA
  - Time-stamping service
  - Hash chaining with library tree
  - Can't post-date or pre-date
  - Algorithm (patented)
    - Uses signed hash (256-bit) of document
    - Service combines hash with other recent hashes
    - Each minute creates new hash of aggregate hash with previous
    - User gets certificate of hash path
    - Service publishes Time and supersigns each week in New York Times
    - Third party can verify hash and supersign

- Digital notaries 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