15-853:Algorithms in the Real World

Cryptography 3 and 4

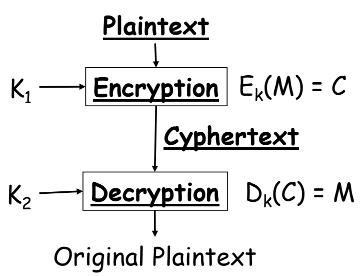
Cryptography Outline

Introduction: terminology, cryptanalysis, security
Primitives: one-way functions, trapdoors, …
Protocols: digital signatures, key exchange, ..
Number Theory: groups, fields, …
Private-Key Algorithms: Rijndael, DES
Public-Key Algorithms:
- Diffie-Hellman Key Exchange
- El-Gamal, RSA, Blum-Goldwasser
- Quantum Cryptography

Case Studies: Kerberos, Digital Cash

Public Key Cryptosystems

Introduced by Diffie and Hellman in 1976.



Public Key systems
K_1 = public key
K_2 = private key

Digital signatures
K_1 = private key
K_2 = public key

Typically used as part of a more complicated protocol.

One-way trapdoor functions

Both Public-Key and Digital signatures make use of one-way trapdoor functions.

Public Key:
- Encode: $c = f(m)$
- Decode: $m = f^{-1}(c)$ using trapdoor

Digital Signatures:
- Sign: $c = f^{-1}(m)$ using trapdoor
- Verify: $m = f(c)$

Example of TLS (previously SSL)

TLS (**Transport Layer Security**) is the standard for the web (**https**), and voice over IP.

Protocol (somewhat simplified): Bob -> amazon.com

B->A: **client hello**: protocol version, acceptable ciphers
A->B: **server hello**: cipher, session ID, $|amazon.com|_{verisign}$
B->A: **key exchange**, $\{masterkey\}_{amazon's\ public\ key}$
A->B: **server finish**: $([amazon,prev\text{-}messages,masterkey])_{key1}$
B->A: **client finish**: $([bob,prev\text{-}messages,masterkey])_{key2}$
(hand-shake)

A->B: **server message**: $(message1,[message1])_{key1}$
B->A: **client message**: $(message2,[message2])_{key2}$
(data)

$|h|_{issuer}$ = Certificate
= Issuer, $\langle h,h\text{'s public key, time stamp}\rangle_{issuer\text{'s private key}}$

$\langle\ldots\rangle_{private\ key}$ = Digital signature $\{\ldots\}_{public\ key}$ = Public-key encryption
[..] = Secure Hash $(\ldots)_{key}$ = Private-key encryption

key1 and key2 are derived from masterkey and session ID

Public Key History

Some algorithms

- **Diffie-Hellman,** 1976, key-exchange based on discrete logs
- **Merkle-Hellman**, 1978, based on "knapsack problem"
- **McEliece**, 1978, based on algebraic coding theory
- **RSA**, 1978, based on factoring
- **Rabin**, 1979, security can be reduced to factoring
- **ElGamal**, 1985, based on discrete logs
- **Blum-Goldwasser,** 1985, based on quadratic residues
- **Elliptic curves**, 1985, discrete logs over Elliptic curves
- **Chor-Rivest,** 1988, based on knapsack problem
- **NTRU**, 1996, based on Lattices
- **XTR,** 2000, based on discrete logs of a particular field

Diffie-Hellman Key Exchange

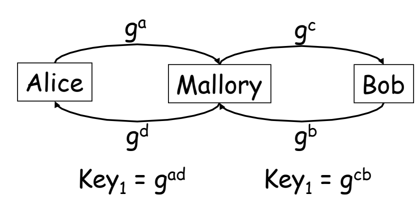
A group (G,*) and a primitive element (generator) g is made public.

- Alice picks a, and sends g^a to Bob
- Bob picks b and sends g^b to Alice
- The shared key is g^{ab}

Note the shared key is easy for Alice or Bob to compute, but assuming discrete logs are hard is hard for anyone else to compute.

Can someone see a problem with this protocol?

Person-in-the-middle attack



Mallory gets to listen to everything.

ElGamal

Based on the difficulty of the discrete log problem.

Invented in 1985

Digital signature and Key-exchange variants

- Digital signature is AES standard
- Public Key used by TRW (avoided RSA patent)

Works over various groups

- Z_p,
- Multiplicative group $GF(p^n)$,
- Elliptic Curves

ElGamal Public-key Cryptosystem

(G,*) is a group

- α a generator for G
- $a \in Z_{|G|}$
- $\beta = \alpha^a$

G is selected so that it is hard to solve the discrete log problem.

Public Key: (α, β) and some description of G

Private Key: a

Encode:

Pick random $k \in Z_{|G|}$

$E(m) = (y_1, y_2)$
$= (\alpha^k, m * \beta^k)$

Decode:

$D(y) = y_2 * (y_1^a)^{-1}$
$= (m * \beta^k) * (\alpha^{ka})^{-1}$
$= m * \beta^k * (\beta^k)^{-1}$
$= m$

You need to know a to easily decode y!

ElGamal: Example

$G = Z_{11}^*$

- $\alpha = 2$
- $a = 8$
- $\beta = 2^8 \pmod{11} = 3$

Public Key: $(2, 3), Z_{11}^*$

Private Key: $a = 8$

Encode: 7

Pick random k = 4

$E(m) = (2^4, 7 * 3^4)$
$= (5, 6)$

Decode: (5, 6)

$D(y) = 6 * (5^8)^{-1}$
$= 6 * 4^{-1}$
$= 6 * 3 \pmod{11}$
$= 7$

Merkle-Hellman

Gets "security" from the **Subet Sum** (also called **knapsack**) which is NP-hard to solve in general.

Subset Sum (Knapsack): Given a sequence $W = \{w_0, w_1, \ldots, w_{n-1}\}$, $w_i \in Z$ of weights and a sum S, calculate a boolean vector B, such that:

$$\sum_{i=0}^{i<n} B_i W_i = S$$

Even deciding if there is a solution is NP-hard.

Merkle-Hellman

W is **superincreasing** if: $w_i \geq \sum_{j=0}^{i-1} w_j$

It is easy to solve the subset-sum problem for superincreasing W in O(n) time.

Main idea of Merkle-Hellman:

- Hide the easy case by multiplying each w_i by a constant **a** modulo a prime **p**
  $$w_i' = a * w_i \bmod p$$
- Knowing a and p allows you to retrieve the superincreasing sequence

Merkle-Hellman

What we need

- $w_1, \cdots, w_n$ superincreasing integers
- $p > \sum_{i=1}^{n} w_i$ and prime
- a, $1 \leq a \leq n$
- $w'_i = a\, w_i \bmod p$

Public Key: w'_i

Private Key: w_i, p, a,

Encode:

$y = E(m) = \sum_{i=1}^{n} m_i\, w'_i$

Decode:

$z = a^{-1}\, y \bmod p$

$= a^{-1} \sum_{i=1}^{n} m_i\, w'_i \bmod p$

$= a^{-1} \sum_{i=1}^{n} m_i a_i w_i \bmod p$

$= \sum_{i=1}^{n} m_i\, w_i$

Solve subset sum prob: $(w_1, \cdots, w_n, z)$

obtaining $m_1, \cdots m_n$

Merkle Hellman: Problem

Was broken by Shamir in 1984.

Shamir showed how to use integer programming to solve the particular class of Subset Sum problems in polynomial time.

Lesson: don't leave your trapdoor loose.

RSA

Name after Rivest, Shamir and Adleman (1978) but apparently invented by Clifford Cocks in 1973.

Based on **difficulty of factoring**.

Used to **hide** the size of a group Z_n^* since:

. $$|Z_n^*| = \phi(n) = n \prod_{p|n} (1 - 1/p)$$

Factoring has not been reduced to RSA

- an algorithm that generates m from c does not give an efficient algorithm for factoring

On the other hand, factoring has been reduced to finding the private-key.

- there is an efficient algorithm for factoring given one that can find the private key from the public key.

RSA Public-key Cryptosystem

What we need:
- p and q, primes of approximately the same size
- n = pq
  $\phi(n) = (p-1)(q-1)$
- $e \in Z_{\phi(n)}^*$
- $d = e^{-1} \bmod \phi(n)$

Public Key: (e,n)
Private Key: d

Encode:
$m \in Z_n$
$E(m) = m^e \bmod n$

Decode:
$D(c) = c^d \bmod n$

RSA continued

Why it works:

$D(c) = c^d \bmod n$
$= m^{ed} \bmod n$
$= m^{1 + k(p-1)(q-1)} \bmod n$
$= m^{1 + k\,\phi(n)} \bmod n$
$= m(m^{\phi(n)})^k \bmod n$
$= m$

Why is this argument not quite sound?

What if $m \notin Z_n^*$ then $m^{\phi(n)} \neq 1 \bmod n$

Answer 1: Not hard to show that it still works.

Answer 2: jackpot – you've factored n

RSA computations

To **generate the keys**, we need to
- **Find two primes p and q.** Generate candidates and use primality testing to filter them.
- **Find e^{-1} mod (p-1)(q-1).** Use Euclid's algorithm. Takes time $\log^2(n)$

To **encode and decode**
- **Take m^e or c^d.** Use the power method. Takes time $\log(e) \log^2(n)$ and $\log(d) \log^2(n)$.

In practice e is selected to be small so that encoding is fast.

Security of RSA

Warning:
- Do not use this or any other algorithm naively!

Possible security holes:
- Need to use "safe" primes p and q. In particular p-1 and q-1 should have large prime factors.
- p and q should not have the same number of digits. Can use a middle attack starting at sqrt(n).
- e cannot be too small
- Don't use same n for different e's.
- You should always "pad"

Algorithm to factor given d and e

If an attacker has an algorithm that generates d from e, then he/she can factor n in PPT. Variant of the Rabin-Miller primality test.

Function TryFactor(e,d,n)

1. write **ed** – 1 as $2^s r$, r odd
2. choose w at random < n
3. $v = w^r \bmod n$
4. if v = 1 then return(**fail**)
5. while $v \neq 1 \bmod n$
6. $v_0 = v$
7. $v = v^2 \bmod n$
8. if $v_0 = n - 1$ then return(**fail**)
9. return(**pass**, $\gcd(v_0 + 1, n)$)

LasVegas algorithm
Probability of pass is > .5.
Will return p or q if it passes.
Try until you pass.

RSA Performance

Performance: (600Mhz PIII) (from: ssh toolkit):

Algorithm	Bits/key		Mbits/sec
RSA Keygen	1024	.35sec/key	
	2048	2.83sec/key	
RSA Encrypt	1024	1786/sec	3.5
	2048	672/sec	1.2
RSA Decrypt	1024	74/sec	.074
	2048	12/sec	.024
ElGamal Enc.	1024	31/sec	.031
ElGamal Dec.	1024	61/sec	.061
DES-cbc	56		95
twofish-cbc	128		140
Rijndael	128		180

RSA in the "Real World"

Part of many standards: PKCS, ITU X.509, ANSI X9.31, IEEE P1363

Used by: SSL, PEM, PGP, Entrust, …

The standards specify many details on the implementation, e.g.

- e should be selected to be small, but not too small
- "multi prime" versions make use of n = pqr… this makes it cheaper to decode especially in parallel (uses Chinese remainder theorem).

Factoring in the Real World

Quadratic Sieve (QS):

$$T(n) = e^{(1+o(n))(\ln n)^{1/2}(\ln(\ln n))^{1/2}}$$

- Used in 1994 to factor a 129 digit (428-bit) number. 1600 Machines, 8 months.

Number field Sieve (NFS):

$$T(n) = e^{(1.923+o(1))(\ln n)^{1/3}(\ln(\ln n))^{2/3}}$$

- Used in 1999 to factor 155 digit (512-bit) number. 35 CPU years. At least 4x faster than QS
- Used in 2003-2005 to factor 200 digits (663 bits) 75 CPU years ($20K prize)

Probabilistic Encryption

For RSA one message goes to one cipher word. This means we might gain information by running $E_{public}(M)$.

Probabilistic encryption maps every *M* to many *C* randomly. Cryptanalysists can't tell whether $C = E_{public}(M)$.

ElGamal is an example (based on the random k), but it doubles the size of message.

BBS "secure" random bits

BBS (Blum, Blum and Shub, 1984)

- Based on difficulty of factoring, or finding square roots modulo n = pq.

Fixed

- p and q are primes such that p = q = 3 (mod 4)
- n = pq (is called a Blum integer)

For a particular bit seq.

- **Seed:** random x relatively prime to n.
- **Initial state:** $x_0 = x^2$
- **i^th state:** $x_i = (x_{i-1})^2$
- **i^th bit:** lsb of x_i

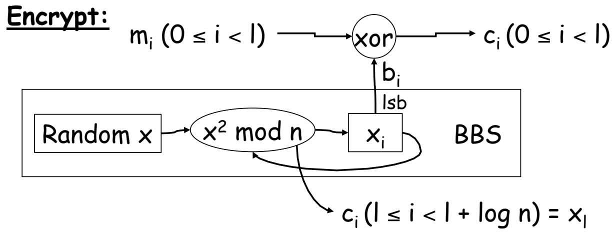
Note that: $x_0 = x_i^{-2^i \bmod \phi(n)} \pmod{n}$

Therefore knowing p and q allows us to find x_0 from x_i

Blum-Goldwasser: A stream cypher

Public key: n (= pq) **Private key**: p or q

Encrypt:



Decrypt:

Using p and q, find $x_0 = x_i^{-2^i \bmod (p-1)(q-1)} \pmod{n}$

Use this to regenerate the $\mathbf{b_i}$ and hence $\mathbf{m_i}$

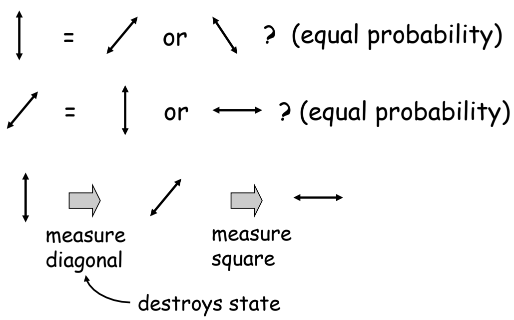
Quantum Cryptography

In quantum mechanics, there is no way to take a measurement without potentially changing the state. E.g.

- Measuring position, spreads out the momentum
- Measuring spin horizontally, "spreads out" the spin probability vertically

Related to Heisenberg's uncertainty principal

Using photon polarization



Quantum Key Exchange

1. Alice sends bob photon stream randomly polarized in one of 4 polarizations: ↕ ↔ ⤢ ⤡
2. Bob measures photons in random orientations
   e.g.: X + + X X X + X (orientations used)
   \ | - \ / / - \ (measured polarizations)
   and tells Alice in the open what orientations he used, but not what he measured.
3. Alice tells Bob in the open which are correct
4. Bob and Alice keep the correct values

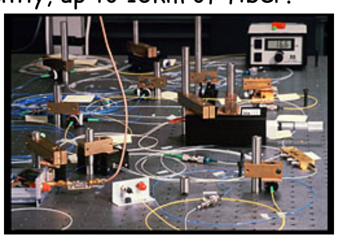
Susceptible to a **man-in-the-middle** attack

In the "real world"

Not yet used in practice, but experiments have verified that it works.

IBM has working system over 30cm at 10bits/sec.

More recently, up to 10km of fiber.



Cryptography Outline

Introduction: terminology, cryptanalysis, security

Primitives: one-way functions, trapdoors, …

Protocols: digital signatures, key exchange, ..

Number Theory: groups, fields, …

Private-Key Algorithms: Rijndael, DES

Public-Key Algorithms: Knapsack, RSA, El-Gamal, …

Case Studies:
- Kerberos
- Digital Cash (not this year)

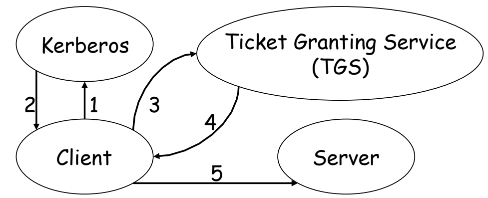
Kerberos

A key-serving system based on Private-Keys (DES).

Assumptions

- Built on top of TCP/IP networks
- Many "**clients**" (typically users, but perhaps software)
- Many "**servers**" (e.g. file servers, compute servers, print servers, …)
- User machines and servers are potentially insecure without compromising the whole system
- A **kerberos server** must be secure.

Kerberos



1. Request *ticket-granting-ticket* (TGT)
2. <TGT>
3. Request *server-ticket* (ST)
4. <ST>
5. Request service

Kerberos V Message Formats

C = client S = server K = key
T = timestamp V = time range
TGS = Ticket Granting Service A = Net Address

Ticket Granting Ticket: $T_{C,TGS} = TGS,\{C,A,V,K_{C,TGS}\}K_{TGS}$
Server Ticket: $T_{C,S} = S, \{C,A,V,K_{C,S}\}K_S$
Authenticator: $A_{C,S} = \{C,T,[K]\}K_{C,S}$

1. Client to Kerberos: $\{C,TGS\}K_C$
2. Kerberos to Client: $\{K_{C,TGS}\}K_C$, $T_{C,TGS}$
3. Client to TGS: $A_{C,TGS}$, $T_{C,TGS}$
4. TGS to Client: $\{K_{C,S}\}K_{C,TGS}$, $T_{C,S}$
5. Client to Server: $A_{C,S}$, $T_{C,S}$

(Steps 3–5: Possibly repeat)

Kerberos Notes

All machines have to have synchronized clocks
- Must not be able to reuse authenticators

Servers should store all previous and valid tickets
- Help prevent replays

Client keys are typically a one-way hash of the password. Clients do not keep these keys.

Kerberos 5 uses CBC mode for encryption Kerberos 4 was insecure because it used a nonstandard mode.