Today

Cryptography 101.

- Alice, Bob & the Adversary
- The 1-time pad
- pseudorandom pads
- public-key crypto, RSA

Cryptography 101 (01101001011...)

Cryptography 101

Alice, Bob.

Alice wants to send Bob a message

But not plaintext, which anyone can read.

Alice wants to send ciphertext, an encoding of the plaintext.

Adversary

The adversary sees the ciphertext, and tries to recover the plaintext.

Adversary

Goal: “Break” the code.

What does this mean?

Different possibilities:

1: with nothing but ciphertext, recover the plaintext.

2: with some access to the coding function, learn to decode

3: learn to send “spoof” messages, etc.

Adding Noise

One way to encode a message:

Take plaintext:

hi dude

Add some random stuff:

abcabca

Output result (looks random)

ikcewgf

To decode: subtract same random stuff.

One-Time Pad

Invincible!

1) Roll dice to make a truly random “noise” sequence (no patterns allowed)

2) Alice & Bob each write it down (on a pad)

3) Alice sends “message + noise”. Every possibility equally likely, so no help to adversary.

4) Bob decrypts “codeword - noise”

5) Don’t re-use the same pad again.

What if we re-use the pad

Safety can be lost.

Adversary: get 2 encodings. Subtract them. (m1 + pad) - (m2 + pad) == m1-m2.

How does this help?

m1+pad looked random. no info.

m1-m2 looks less random. For instance, going to see bunch of 0's everywhere the same word occurs in same place in both messages. Also, "d". "e-a" or "z-v"?

More on pad re-use

Other dangers:

- Adversary gets hold of m1 plaintext.
- Uses this to recover pad. Can now decrypt all further messages.

Fine, so don't re-use the pad.

- Need fresh random bits for every message. May be expensive & inconvenient.
- Pads must be guarded from theft/spying.

Pseudo-random functions

We don't want the expenses associated with one-time pads, do want the security.

Idea: Agree on a function, f, whose output LOOKS "kinda like" random noise. Only Alice & Bob know f. Use f(i) for the i'th letter on the pad.

Advantage: f can be shared with much less communication than a pad can.

Disadvantages: 1) patterns. exploitable? 2) still need to share f secretly.

Public-key crypto

Previous schemes have the disadvantage of forcing Alice & Bob to start with some shared secret information.

Public-key cryptosystems don't need this. Leads to many new applications.

Basic idea: Alice makes up an encoding function, f, together with its inverse g (the decoding function). She tells everyone how to compute f, but keeps g secret. Now, anyone can send her coded messages f(m).

Interception

Suppose the Adversary gets a coded message from Alice to Bob, C.

Knows C = f(m) for some m, and knows f.

Can enumerate $f(s_1)$, $f(s_2)$, $f(s_3)$, ... for all possible message strings. If find $C=f(s_i)$, then know $m==s_i$. But: 2^n possible strings of length n, so not feasible for large n.

Also know: Alice can compute inverse, g, fast. Can we use structure of f to find g?

A Thought Experiment

Suppose encoding algorithm f is: compute a pseudorandom string s, of the same length as m. Then f(m) = m+s.

We can break this! After intercepting C, we know the length of m. Know f, so we can compute s. But now, m = C-s, so done!

Moral: f needs to be harder to invert. "Pad" must depend on message, m, in more complicated way.

RSA cryptosystem

In examples so far, have informally used + to mean “mod 26” arithmetic. So letters are numbers. We’ll do this more carefully now, for the RSA cryptosystem.

First: “letters” will come from a much larger alphabet, say, strings of length 100.

Second: we will do modular arithmetic using a fancier base.

RSA (simplified version)

Alice: choose 2 large prime numbers, p, q.

Compute n=pq, and a random number e between 1 and pq.

Encoding instructions: Let $0 \leq m \leq pq-1$ be the message you want to send me. Compute $f(m) = m^e \pmod{n}$, and publish that. Alice will be able to read it, and nobody else.

RSA (actual)

Alice: choose 2 large prime numbers, p, q.

Compute n=pq, and a random number e between 1 and (p-1)(q-1). Make sure e is relatively prime to both (p-1) and (q-1).

“Relatively prime”: 2 numbers share no prime factors. 4 & 9 are r.p. 6 & 9 not.

Encoding instructions: Let $0 \leq m \leq pq-1$ be the message you want to send. Compute $f(m) = m^e \pmod{n}$, and publish that.

Fast exponentiation

How to compute $f(m) = m^e \pmod{n}$?

Fast exponentiation

How to compute $f(m) = m^e \pmod n$?

Naive: do e iterations: multiply by m.

Fast exponentiation

How to compute $f(m) = m^e \pmod n$?

Naive: do e iterations: multiply by m.

Faster: repeated squaring.

(reduce mod m after each time).

(see whiteboard)

Decoding RSA

Fact (of number theory): For any integers m and x:

$m^{x*(p-1)*(q-1)+1} == m \pmod{p*q}$

Alice: computes a "decoding exponent," d, such that d*e is of the form (x*(p-1)*(q-1)+1)

Then, given codeword $C == m^e \pmod{n}$, she computes $C^d \pmod{n} == m$.

Security & Practicality

Given n and e, it is (apparently) hard to compute d.

More formally, for “typical” choices of e, it is exactly as hard as factoring n. This is widely believed to be really hard, but probably not NP-hard.

But, if you know p, q, it is easy to find d (Euclid’s algorithm for gcd).

Euclid's Algorithm

INPUT: (a,b) non-negative integers, not (0,0)

OUTPUT: greatest common divisor, d.

code:

```
gcd(a,b):
    if (a==0) return b.
    else if (a > b) return gcd(b,a)
    else return gcd(a,b-a)     // b%a is faster.
```

Extended Euclidean Alg

INPUT: (a,b) integers ≥ 0, not both 0.

OUTPUT: integers x,y, such that x*a+b*y == gcd(a,b).

algorithm: work backwords up the sequence of equations in Euclid's algorithm.

Full code in Wikipedia article. Let's do an example by hand.

Extended Euclidean Alg

Example: a=27, b=5.

Solve: a*x + b*y = gcd(a,b) = 1.

gcd(27,5) -> gcd(5,2) -> gcd(2,1) -> gcd(1,0) == 1

Back-solve: 1 == 1.

27 = 5*5 + 2

5 = 2*2 + 1

2 = 2*1 + 0

Extended Euclidean Alg

Example: a=27, b=5.

Solve: a*x + b*y = gcd(a,b) = 1.

gcd(27,5) -> gcd(5,2) -> gcd(2,1) -> gcd(1,0) == 1

Back-solve: 1 == 1.

27 = 5*5 + 2 27 - 5*5 = 2

5 = 2*2 + 1 5 - 2*2 = 1

2 = 2*1 + 0

Extended Euclidean Alg

Example: a=27, b=5.

Solve: a*x + b*y = gcd(a,b) = 1.

gcd(27,5) -> gcd(5,2) -> gcd(2,1) -> gcd(1,0) == 1

Back-solve: 1 == 1.

27 = 5*5 + 2 27 - 5*5 = 2

5 = 2*2 + 1 5 - 2*2 = 1 = 5 - 2*(27-5*5)

2 = 2*1 + 0 = (-2)*27 + 11*5

Extended Euclidean Alg

A bigger example: a=46, b=32.

46 - 32 = 14

32 - 2*14 = 4

14 - 3*4 = 2

4 - 2*2 = 0.

gcd(42,16) = 2.

Extended Euclidean Alg

A bigger example: a=46, b=32.

46 - 32 = 14

32 - 2*14 = 4

14 - 3*4 = 2 = 14 - 3*(32-2*14) = (-3)*32+7*14

4 - 2*2 = 0.

gcd(42,16) = 2.

Extended Euclidean Alg

A bigger example: a=46, b=32.

46 - 32 = 14

32 - 2*14 = 4

14 - 3*4 = 2 = 14 - 3*(32-2*14) = (-3)*32+7*14

4 - 2*2 = 0. =(-3)*32+7*(46-32)=7*46-10*32

gcd(42,16) = 2.

RSA example

Let p=3, q=11 be the secret primes.

Possible encoding/decoding exponents:

Must have (e*d)%((p-1)*(q-1))==1.

First: e must be relatively prime to 20.

encoding/decoding pairs: (1,1). (3,7).

(9,9), (11,11), (13,17), (19,19).

RSA example

Let p=13, q=11 be the secret primes.

Possible encoding/decoding exponents:

Must have (e*d)%((p-1)*(q-1))==1.

First: e must be relatively prime to 120.

Say we pick e=7.

120 - 17*7 = 1. So d=17.

How about e=13? 120-9*13=3.

13 - 4*3 = 1 = 13 - 4*(120)-9*13 = 4*120+37*13.
So d=37.

RSA example

Let p=3, q=11 be the secret primes.

e = 17. d = 13. n = 33.

Encryption: (m^e) % 33.

Example: m=2.

Repeated squaring, mod 33.

m^2 = 4. m^4 = 16. m^8 = 256 = 25.

m^16 = 625 = 31. m^17 = 62 = 29. Sent!

Decoding: (C^d)%33.

RSA example

Let p=3, q=11 be the secret primes.
e = 17. d = 13. n = 33.
Encryption: (m^e) % 33.
Example: m=2.
Repeated squaring, mod 33.
m^2 = 4. m^4 = 16. m^8 = 256 = 25.
m^16 = 625 = 31. m^17 = 62 = 29. Sent!
Decoding: (C^d)%33.
C^2 = 841 = 16. C^4 = 256 = 25. C^8 = 625 = 31. C^12 = 775 = 16. C^13 = 464 = 2 = m. (all arithmetic done mod 33).

Secure Signatures

Want to be able to “sign” documents sent over the internet.

Signature == proof of identity

Use RSA! Alice takes her document, m, and sends m^d (mod n). Everyone else can read it, by computing $(m^d)^e$ (mod n) == m. But only Alice should know d, so only Alice could have sent the message (without brute force search).

Secure Signatures

Secure signature can be combined with encryption, so that messages can be sent privately, and the sender verified.

Idea: Alice encodes message m using Bob's public key, then signs the whole bundle using her private key. Bob decodes the bundle using Alice's public key, to verify that it came from Alice. Then he decodes the actual message using his private key.