Introduction to cryptology (GBIN8U16)

✧

Symmetric recap, Asymmetric start

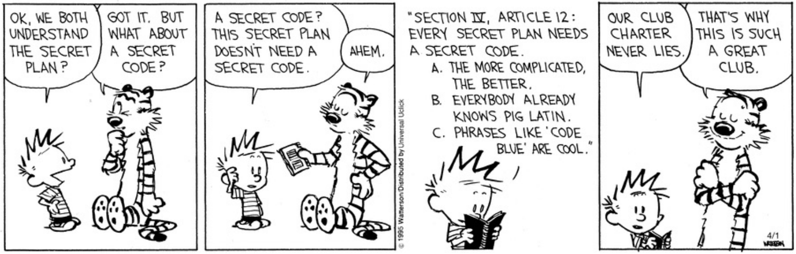
Pierre Karpman
pierre.karpman@univ-grenoble-alpes.fr
https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html

2018–02–28

Important information

- No lecture next week and in three weeks
- Two lectures in two and four weeks
- Contrôle continu lab session in two and three weeks

We always need secret codes



Part of the solution: symmetric crypto

Goal: given a shared secret k, establish a secure channel

- Provides confidentiality of communications
- Authenticity
- Integrity

against active adversaries not knowing k

A main security property for confidentiality: IND-CPA

1. Submit messages to an *oracle* $\mathfrak{O}$ to be encrypted, & get the result
2. Choose, m_0, m_1, send both to $\mathfrak{O}$
3. Receive $\mathfrak{O}(m_b)$ for a random $b \in \{0, 1\}$
4. Goal: determine the value of b (better than by guessing)

(Erratum: m_0 and m_1 have to be of equal length)

How to achieve IND-CPA

A “good” $\mathfrak{O}$: a block cipher w/ a randomized mode of operation.
But:

- “Only” computational security $\Rightarrow$ can always find the key, spending enough time
- Cannot encrypt too many (or too long) messages without changing the key

Examples of modes: CTR, CBC

How to achieve authenticity+integrity

Use a MAC (by definition). But again:

- "Only" computational security $\Rightarrow$ can always find the key, spending enough time
- Cannot authenticate too many (or too long) messages without changing the key
- May or may not be randomized

A good type of MAC

Polynomial MACs:

- Messages are polynomials (over a finite field)
- Evaluate a message on a secret point
- Encrypt the result (e.g. with a block cipher) to break linearity

(Erratum: n-block messages need to be mapped to degree-n polynomials, w/o a constant term)

MAC + Encryption

Authenticated encryption AE: jointly provide confidentiality and auth/integrity. One way: combine a MAC $\mathcal{M}$ and an encryption scheme Enc:

- $\mathrm{AE}(m) = \mathrm{Enc}(m) \| \mathcal{M}(m)$ → bad (as in "not always good")
- $\mathrm{AE}(m) = x := \mathrm{Enc}(m) \| \mathcal{M}(x)$ → good
- $\mathrm{AE}(m) = \mathrm{Enc}(m \| \mathcal{M}(m))$ → also good

AE: the most efficient way to communicate securely.

⇒ But we need a shared key!

How to get a key

Some possibilities

- Meet in person (impractical)
- Use secure message transmission (not so practical (but very nice!))
- Use asymmetric “public-key” schemes (quite practical)

Public-key algorithms

Some major examples:

- Asymmetric encryption (one key to encrypt, another to decrypt), e.g. RSA (+ some randomized padding)
- Digital signature (one key to sign, another to verify), e.g. DSA
- Public-key key exchange, e.g. Diffie-Hellman

Note: RSA can be used to implement both a key-exchange and a signature

Today's focus: Diffie-Hellman

A simple protocol:

- Let $\mathbb{G} = \langle g \rangle$ be a cyclic finite group with a generator g
  - Example: $(\mathbb{Z}/512\mathbb{Z}, +)$, $g = 1$, $\operatorname{ord}(g) = 512$
  - Example: $\mathbb{F}_{257}^*$, $g = 3$, $\operatorname{ord}(g) = 256$
  - Example: $(\mathbb{F}_2[X]/X^8 + X^4 + X^3 + X^2 + 1)^*$, $g = X$, $\operatorname{ord}(g) = 255$
- A picks $a \xleftarrow{\$} \{0, \ldots, \operatorname{ord}(g) - 1\}$, sends g^a to B
- B picks $b \xleftarrow{\$} \{0, \ldots, \operatorname{ord}(g) - 1\}$, sends g^b to A
- A computes $(g^b)^a = g^{ba} = g^{ab}$, sets $k = \mathrm{KDF}(g^{ab})$
- B computes $(g^a)^b = g^{ab}$, sets $k = \mathrm{KDF}(g^{ab})$

With KDF some *key derivation function* (e.g. a ~ hash function)

Why this works?

Functionality

- A and B only need public information to perform the exchange
- They get the same k

$\Rightarrow$ Public-key key exchange

Security: necessary conditions

- Given g, g^a, g^b, it must be hard to compute g^{ab}
- $k = \mathrm{KDF}(g^{ab})$ must be "random-looking" when a, b are random
- There must be many possible values for k

Security focus

A necessary condition: computing *discrete logarithms* in $\mathbb{G}$ must be "hard"

Discrete logarithm

Let $\mathbb{G} = \langle g \rangle$ be a finite group of order N, the *discrete logarithm* of $h = g^a$, $a \in \{0, \ldots, N-1\}$ is equal to a

How hard is the "discrete logarithm problem" (DLP) for various groups?

DLP hardness

Proposition

It is always possible to compute the discrete logarithm in a group of order N in time $O(\sqrt{N})$

So one must *at least* pick N s.t. $2^{\log(N)/2}$ is large. But:

- $(\mathbb{Z}/n\mathbb{Z}, +)$: DLP always easy (logarithm $\equiv$ division)
- $\mathbb{F}_q^*$: usually hard, not *maximally* hard (needs much less than $\sqrt{N}$)
- $E(\mathbb{F}_q)$: usually maximally hard (needs about $\sqrt{N}$)

A simple generic algorithm

Idea: use *collisions* to reveal the solution. One way to do this: baby-step/giant-step

- Let $\mathbb{G}$ be of order N, $h = g^a$ for some $a \in \{0, \dots, N-1\}$
- Let $r = \lceil \sqrt{N} \rceil$, then $a = ra_1 + a_0$, with a_0, a_1 less than r
- We have $h = g^{ra_1 + a_0}$, so $h/g^{a_0} = g^{ra_1}$

$\Rightarrow$

1. Compute $L_0 = [hg^{-x}, x < r]$, $L_1 = [g^{ry}, y < r]$
2. Find i, j s.t. $L_0[i] = L_1[j]$
3. Return $a = rj + i$

Baby-step/giant-step: Comments

- The baby-step/giant-step algorithm works *with any* group
- It has time and memory complexity equal to $\sqrt{\mathrm{ord}(\mathbb{G})} \Rightarrow$ generically optimal!
- Other collision-based algorithms exist with constant memory complexity
- Depending on $\mathbb{G}$, better algorithms may be available (we've seen some examples)

More on how to pick a group

If the order N of $\mathbb{G}$ is not prime, $\mathbb{G}$ has *subgroups*

- Let $N = pN'$, then g^p generates a group of order N'

Proposition (Pohlig-Hellman)

It is possible to solve the DLP in $\mathbb{G}$ subgroup-by-subgroup

$\Rightarrow$ For the DLP to be hard, $\mathbb{G}$ must be of order N s.t. DLP is hard in a subgroup of order p, the largest prime factor of N (But no details for now)

Are we done? Not quite

- Hardness of the DLP cannot be "proven", but a reasonable assumption for some groups
- We also need g^x to be random-looking (ditto)

But regardless, Diffie-Hellman as presented only protects againts *passive* adversaries

$\Rightarrow$ Not very useful in practice

Diffie-Hellman with a man in the middle

- A sends g^a to B
  - C intercepts the message, sends g^c to B
- B sends g^b to A
  - C intercepts the message, sends g^c to A
- A and C share a key $k_a = \mathrm{KDF}(g^{ac})$
- B and C share a key $k_b = \mathrm{KDF}(g^{bc})$
- Anytime A sends a message to B with key k_a, C decrypts and re-encrypts with k_b (and vice-versa)

One way to solve this: signatures

A wants to be sure it is talking to B

- Find B's public verification key for a signature algorithm
- Ask B to sign g^b
- Only accept it if the signature is valid

Works well, but A needs to know B's public key *beforehand*

$\Rightarrow$ We again have a bootstrapping issue

So are we back to square one?

Public-key infrastructures can help

Public keys still help compared to private ones:

- Possibly long term (v. have to be changed after a while (although not a real limitation))
- Scales linearly w/ the number of participants (v. quadratically)
- Trusting only one key is enough, if it signs all the ones you need

Example: TLS certificates

The simple picture:

- Web browsers are pre-loaded with “certificates” (~ public keys) of certification authorities (CAs)
- CAs sign the certificates of websites using secure connections (possibly using intermediaries)
- When connecting to a website, check the entire chain of certificates
- If everything’s fine, use the website’s public key to authenticate the exchange

So how do we sign?

Signature possibilities

- Use a discrete logarithm based protocol
- Or RSA
- But in both cases, also need a hash function!

$\Rightarrow$ Details in two weeks!