

Introduction to cryptology (GBIN8U16)



Public-Key Cryptography: Discrete logarithm-based schemes

Pierre Karpman

pierre.karpman@univ-grenoble-alpes.fr

<https://membres-ljk.imag.fr/Pierre.Karpman/tea.html>

2022-03

How to get a key

So far we assumed the presence of a shared secret between participants, but how do you get there?

Some possibilities

- ▶ Meet in person (impractical)
- ▶ Use secure message transmission (not so practical, hard to evaluate assumptions? But very nice!)
- ▶ Use asymmetric “public-key” schemes (quite practical) ← our focus now!

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

Group definitions

Finite cyclic group (*multiplicative notation*)

A finite group \mathbb{G} of *order* (or cardinality) N is *cyclic* if $\exists g \in \mathbb{G}$ s.t. $\forall x \in \mathbb{G}, \exists i \in \llbracket 0, N - 1 \rrbracket$ s.t. $x = g^i$. Such an element g is called a *generator* (or primitive element) of the group.

Properties

- ▶ Any element h of \mathbb{G} generates a subgroup $\mathbb{H} := \langle h \rangle$. The order $\text{ord}(h)$ of h is defined as the order (or cardinality) of \mathbb{H} . If $\mathbb{H} = \mathbb{G}$, h is a generator of the full group \mathbb{G} .
- ▶ A group may have several generators.
- ▶ (Lagrange Theorem) If \mathbb{H} is a subgroup of \mathbb{G} , $\#\mathbb{H} \mid \#\mathbb{G}$
(Corollary: if $\#\mathbb{G}$ is prime, all elements except 1 are primitive)

Group examples

An additive group:

- ▶ $(\mathbb{Z}/512\mathbb{Z}, +)$, $g = 1$, $\text{ord}(g) = 512$

Any multiplicative group of a finite field (and more):

- ▶ \mathbb{F}_{257}^\times , $g = 3$, $\text{ord}(g) = 256$
- ▶ $(\mathbb{F}_2[X]/\langle X^8 + X^4 + X^3 + X^2 + 1 \rangle)^\times$, $g = X$, $\text{ord}(g) = 255$
- ▶ $(\mathbb{Z}/n\mathbb{Z})^\times$, of order $\varphi(n)$ ($= n - 1$ when n is prime)
 - ▶ Cf. the extended Euclid algorithm... later!

Today's focus: Diffie-Hellman

A simple protocol:

- ▶ Let $\mathbb{G} = \langle g \rangle$ be a cyclic finite group with a generator g
- ▶ A picks $a \leftarrow \llbracket 0, \text{ord}(g) - 1 \rrbracket$, sends g^a to B
- ▶ B picks $b \leftarrow \llbracket 0, \text{ord}(g) - 1 \rrbracket$, sends g^b to A
- ▶ A computes $(g^b)^a = g^{ba} = g^{ab}$, sets $k = \text{KDF}(g^{ab})$
- ▶ B computes $(g^a)^b = g^{ab}$, sets $k = \text{KDF}(g^{ab})$

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

Why this works?

Functionality

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

⇒ Public-key key exchange

Security: necessary conditions

- ▶ Given g , g^a , g^b , it must be hard to compute g^{ab}
- ▶ $k = \text{KDF}(g^{ab})$ must be “random-looking” when a , b are random
- ▶ (Related: there must be many possible values for k)

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 in base g* of $h = g^a$, $a \in \llbracket 0, N - 1 \rrbracket$ is defined as a

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

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^\times : usually hard, not *maximally* hard (needs much less work 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 $a \in \llbracket 0, N - 1 \rrbracket$
- ▶ 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 $hg^{a_0} = g^{ra_1}$

\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 cost equal to $\sqrt{\text{ord}(\mathbb{G})} \Rightarrow$ generically optimal (up to the memory cost)!
- ▶ It can easily be parallelised
- ▶ It can easily be adapted when the logarithm is known to lie in a “small” interval
- ▶ Other collision-based algorithms exist with constant or small memory cost (such as Pollard’s ρ (also parallelisable) or kangaroos)!
- ▶ 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

⇒ 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 (Idea: use a Chinese Remainder Theorem-like decomposition; no details)

Are we done? Not quite

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

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

⇒ 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 $k_a = \text{KDF}(g^{ac})$
- ▶ B and C share a key $k_b = \text{KDF}(g^{bc})$
- ▶ 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*

⇒ 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!

Signatures: what?

Objectives of a signature algorithm:

- ▶ Given (sk, pk) a key pair
- ▶ message m + secret key $sk \rightsquigarrow$ signature $s = S_{sk}(m)$
- ▶ message m + signature s + public key $pk \rightsquigarrow$ verified message $V_{pk}(m, s)$

Informal security objectives

- ▶ Given pk , it should be hard to find sk
- ▶ Given pk , it should be hard to forge signatures
- ▶ (Variant: given access to a signing oracle $\mathcal{O}_{(sk, pk)}$, it should be hard to forge signatures)
- ▶ Formalised as *Existential unforgeability under chosen-message attacks* (EUF-CMA)

EUFCMA for Public-Key signatures

EUFCMA for (S, V) : An adversary cannot forge a valid signature σ for a message m such that $V(pk_C, \sigma, m)$ succeeds, when given (restricted) oracle access to $S(sk_C, \cdot)$:

- 1 The Challenger chooses a pair (pk_C, sk_C) and sends pk_C to the Adversary
- 2 The Adversary may repeatedly submit queries m_i to the Challenger
- 3 The Challenger answers a query with $\sigma_i = S(sk_C, m_i)$
- 4 The Adversary tries to forge a signature σ_f for a message $m_f \neq m_i$, s.t. $V(pk_C, \sigma_f, m_f) = \top$

Related: interactive proof of identity

Objective of a proof of ID scheme:

- ▶ Publish public identification data α
- ▶ When challenged, prove knowledge of a secret related to α

Example of a one-time scheme:

- 1 Let \mathcal{H} be a preimage-resistant hash function, \mathcal{R} a large set
- 2 The prover draws $x \leftarrow \mathcal{R}$, computes and publishes $X = \mathcal{H}(x)$
- 3 When challenged, reveals x

Many-time variant:

- 1 Draw $x \leftarrow \mathcal{R}$, compute and publish $X = \mathcal{H}^N(x)$
- 2 When challenged, reveal $\mathcal{H}^{N-1}(x)$, reset $X = \mathcal{H}^{N-1}(x)$

This can be generalised even more, leading eventually to *hash-based signatures* (no details)

A discrete-log based PoID scheme

- 1 Let $\mathbb{G} = \langle g \rangle$ be a group with a hard DLP
- 2 The prover draws $x \leftarrow \mathcal{R}$, computes and publishes $X = g^x$
- 3 When challenged; draws r , sends $R = g^r$
- 4 The verifier picks c and sends it
- 5 The prover computes $a = r + cx$ and sends it
- 6 The verifier checks that $RX^c = g^a$

This can be run many times, BUT r 's should be *uniformly* random and never repeat!

From PoID to signature

Differences between PoID and signatures:

- ▶ PoIDs are interactive (in the verification), signatures are not
- ▶ Signatures also involve a message

One major observation:

- ▶ If the prover can guarantee that it doesn't control *both* R and c , interaction is unnecessary
- ▶ Otherwise, nothing is proved \Leftarrow This is actually nice, since it allows to show that the protocol is *zero-knowledge* (cf. TD)

\Rightarrow Fiat-Shamir transformation: generate c from R with a hash function

Schnorr signatures

To sign a message m with the key pair (sk, pk) ($x, X = g^x$)

- 1 Pick $r \leftarrow \mathcal{R}$ and compute $R = g^r$
- 2 Compute $c = \mathcal{H}(R, m)$
- 3 Compute $a = r + cx$ and output (c, a) as the signature of m

To verify a signature:

- 1 Compute $\hat{R} = g^a / X^c = g^a / g^{cx}$
- 2 Check that $c = \mathcal{H}(\hat{R}, m)$

Important: r must (again) be **uniformly** random and not repeat!
(Why?)

Remember randomness (always)!

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

Figure: Not good for Schnorr signatures

Where are we with dlog?

If $\mathbb{G} = \langle g \rangle$ is a prime-order group where the DLP is hard (on average \equiv in the worst case), then:

- ▶ Can do asymmetric key exchange
- ▶ Can do public-key signatures

For signatures we also need

- ▶ Good hash functions
- ▶ Good pseudorandom number generation (for “classical” signature algorithms)

What if I don't trust my PRNG?

- ▶ Typical dlog-based signatures break easily if r is not random enough
 - ▶ Vulnerable to bad implementations or government backdoors
- ▶ But one can tweak them to generate r from the message and the private key using a VIL/VOL-PRF (either completely deterministically or not)
 - ▶ Example: RFC6979
- ▶ N.B. It is indeed fine for a signature algorithm to be deterministic (cf. also later RSA examples)
- ▶ ... But in the case of dlog-based schemes, determinism may help physical attacks

Some comments on dlog attacks

When $\mathbb{G} \approx \mathbb{F}_p^\times$, the current dlog records are:

- ▶ $|p| \approx 795$ bits (Boudot et al., 2019), using a *Number Field Sieve* (NFS) algorithm
 - ▶ Took about 3100 core years
- ▶ $|p| \approx 1024$ bits for a *trapdoored* prime (Fried et al., 2017), using a *Special NFS* (SNFS) algorithm
 - ▶ Took about 385 core years

Note: it may be hard to decide if a prime is trapdoored

One nice (for an attacker) feature of (S)NFS:

- ▶ The largest part of the cost is a *precomputation*, then computing *individual dlogs* is *very fast*

Some more comments on dlog: small subgroup attack

Consider a *semi-static* key exchange,

- ▶ Where one of g^a or g^b (say g^b) is fixed

using $\langle g \rangle \subset \mathbb{F}_p^\times$ where \mathbb{F}_p^\times has many small subgroups

- ▶ Then B must check that “ \hat{g} ” sent by A is in the correct group
- ▶ Otherwise, if \hat{g}^b is in a small group of order N , a malicious A can learn $b \pmod N$
- ▶ ... Then $b \pmod{N'}$, etc.

One way to easily prevent this: use a “safe” prime $p = 2q + 1$, q prime

\Rightarrow Only a small subgroup of order 2 to check for in \mathbb{F}_p^\times

What about implementation, though?

- ▶ We need to compute g^x , for a large x (e.g. 256 bits)
- ▶ Cannot just do $g \times g \times g \times \dots \times g \approx 2^{256}$ times!
- ▶ Notice that $g \times g = g^2$, $g^2 \times g^2 = g^4$, $g^4 \times g^4 = g^{16}$, etc.
- ▶ Also: $g \times g^2 = g^3$, $g^2 \times g^{16} = g^{18}$, etc.

~> “Square & multiply” algorithm

Square & multiply

Square & multiply

Input x, g

Output g^x

- 1 $h = 1$
- 2 While $x \neq 0$
- 3 if ($x \& 1$)
- 4 $h \leftarrow h \times g$
- 5 $g \leftarrow g \times g$
- 6 $x \leftarrow x \gg 1$
- 7 Return h

\Rightarrow Only $\log(x)$ iterations needed!

(Problem here, runtime also depends on $\text{wt}(x)$)

Implementation: what else?

- ▶ We also need multiplication, addition in \mathbb{G}
- ▶ If $\mathbb{G} \subseteq \mathbb{F}_p^\times \Rightarrow$ modular arithmetic
- ▶ Require big number multiplication, (integer) division, remainders, addition
- ▶ \Rightarrow split f as e.g. $f_0 + 2^{64}f_1 + 2^{128}f_2 + \dots$
- ▶ Can use dedicated arithmetic for “efficient” primes (e.g. efficient Barrett reduction)