Lecture 10: Key Exchange

MIT - 6.1600
Fall 2023
Corrigan-Gibbs &
Zeldovich

Plan

- Anonymous key exchange
- DH Protocol
- Pub-key encryption
- Elliptic curve crypto

Midterm exam

10/25 - Same room

Review recitation

10/20 - 11am
(2-105)

So far...

MAC - Auth w/ shared key

Sigs - Auth w/o shared key

Secret-key enc - Enc w/ shared key
(CCA)

Key exchange & - Enc w/o shared key
public-key enc } Today!

DH key exchange is (arguably) the simplest & most beautiful crypto protocol there is

- Beautiful theory
- Works in practice
- Solves a problem people care about

Turned cryptography upside down in 1970s

↳ Hard to imagine what e-commerce would look like w/o key exchange

↳ The DH paper introduced PKC → RSA, etc.

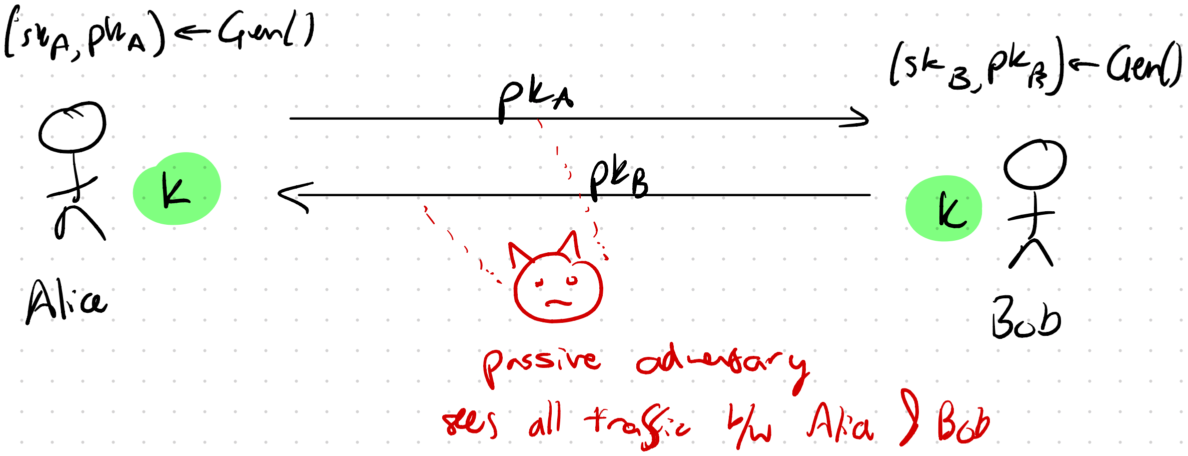
↳ Inspired by Merkle's final project in undergrad class

Can build all crypto primitives we've seen so far from OWF - old-school crypto

↳ Key ex seems to require stronger assumptions

Anonymous key exchange

N.B. This is a toy protocol — see book for details on auth key exchange.



In practice, A & B are cell phone & server, etc.

Key ex. used everywhere: SSH, HTTPS, TLS, WhatsApp, ...
- ↳ essentially whenever using encryption
- ↳ Runs on essentially every new TLS connection

In practice, need authenticated key ex.
- ↳ Not as easy as adding sigs.
- ↳ 68 (!) pages in Boneh-Shoup about it (Ch. 21)

⇒ Just use TLS — takes care of most details for you

Formally, anon key exchange over key space $\mathcal{K}$

- $\text{Gen}() \rightarrow (sk, pk)$
- $\text{Derive}(sk_A, pk_B) \longrightarrow k \in \mathcal{K}$

Correctness: A & B agree on shared secret.

$$\forall \; (sk_A, pk_A) \leftarrow \text{Gen}()$$
$$(sk_B, pk_B) \leftarrow \text{Gen}()$$
$$\text{Derive}(sk_A, pk_B) = \text{Derive}(sk_B, pk_A)$$

Security: Adv can't guess key k given pks.

$\forall$ eff adv A

$$\Pr\left[A(pk_A, pk_B) = \text{Derive}(sk_A, pk_B) : \begin{matrix} (sk_A, pk_A) \leftarrow \text{Gen}() \\ (sk_B, pk_B) \leftarrow \text{Gen}() \end{matrix} \right] \leq \text{negl}.$$

Often want stronger sec prop: $\{k\} \approx \{\text{random}\}$; see notes

[Can get by using Hash(k) as key, modelling hash fn as a random oracle]

Can you build key exchange from OWF?

↳ seems unlikely but no one knows.

We can build from
- RSA (trapdoor OWF)
- DH problem

Theoretical quantum attacks

- Lattice problems ("learning with errors")

The Diffie-Hellman protocol...

Parameters: * big (≈ 2048-bit) prime p

↳ $\approx \lambda$-bit prime, as in RSA

* "generator" $g \in \mathbb{Z}_p \leftarrow \{0, 1, 2, \dots, p-1\}$

Can sometimes just be $g=2$

* integer q ("order") s.t.

$$\{\underbrace{g, g^2, g^3, \dots, g^q}_{\text{order}}, g, g^2, g^3, \dots\}$$

NOTE: Order q can be $\ll p$. often $q \approx 2^{256}$ and $p \approx 2^{2048}$

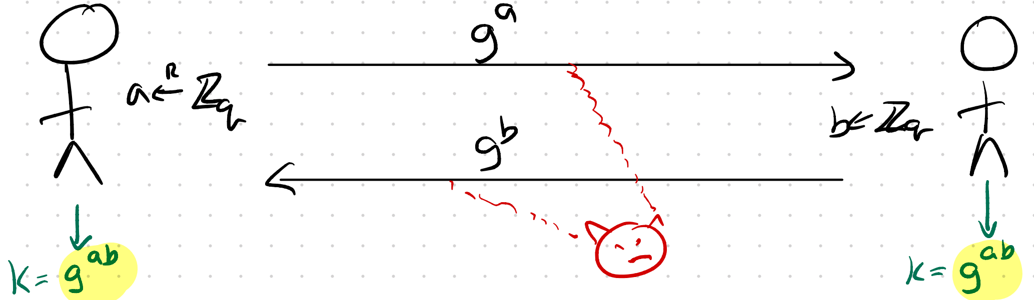
Must choose (p, q, g) carefully! Listed in NIST specs, etc.

$$\mathrm{Gen}() \to \begin{cases} a \xleftarrow{R} \mathbb{Z}_q \\ A \leftarrow g^a \in \mathbb{Z}_p^* \\ \text{return } (sk, pk) \leftarrow (a, A) \end{cases}$$

Notation $g^a \in \mathbb{Z}_p^* \equiv g^a \bmod p$

$$\mathrm{Derive}(sk_A = a,\ pk_B = B \in \mathbb{Z}_p^*)$$

output $k \leftarrow B^a \in \mathbb{Z}_p^*$



Important Detail: Computing $g^x \pmod p$.

$x \approx 2^{2048}$, $p \approx 2^{2048}$

Bad alg: 1) Compute $g^x = 2^{2^{2048}}$ … exponentially large

2) Reduce mod p

Better alg: "Repeated squaring"

$$\begin{tikzcd} g \arrow[r, bend right] & g^2 \arrow[r, bend right] & g^4 \arrow[r, bend right] & g^8 \arrow[r, bend right] & g^{16} \arrow[r, bend right] & g^{32} \arrow[r, bend right] & \cdots \arrow[r, bend right] & g^{2^{2048}} \end{tikzcd} \pmod p$$

To compute $g^{1011_2} = g^{16} \cdot g^2 \cdot g^1 \pmod p$.

Common trick: * When g fixed, precompute powers of g.

* Can use bigger table to get a slight speedup

Correctness: $B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b$

Security:

Attacker's job... "Computational DH problem" (CDH)

Given params (p, g, q) and $\underbrace{(g, g^a, g^b)}$ for $a, b \xleftarrow{R} \mathbb{Z}_q$

Compute g^{ab} ⟵ all mod p

$$\boxed{\text{CDH Assumption: No eff adv can solve CDH problem.}}$$

Related to the simpler discrete-log problem (dlog)

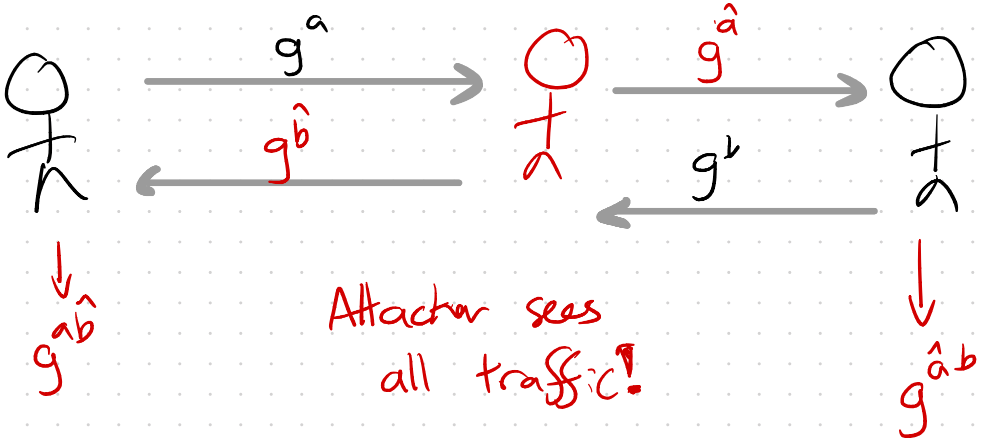
Given: (g, g^a) for $a \xleftarrow{R} \mathbb{Z}_q$ + params (p, g, q)

Compute: $a \in \mathbb{Z}_q$

* If you can solve dlog, can solve CDH.

* In general, we don't know whether solving CDH is enough to solve dlog.... (we do in some special cases)

Warning: Anonymous key exchange provides no guarantees against active attack.



Attacker sees all traffic!

Application: Public-key encryption

* Encryption w/o shared secret over msg space $\mathcal{M}$

$\text{Gen}() \to (sk, pk)$ } Separate keygen alg

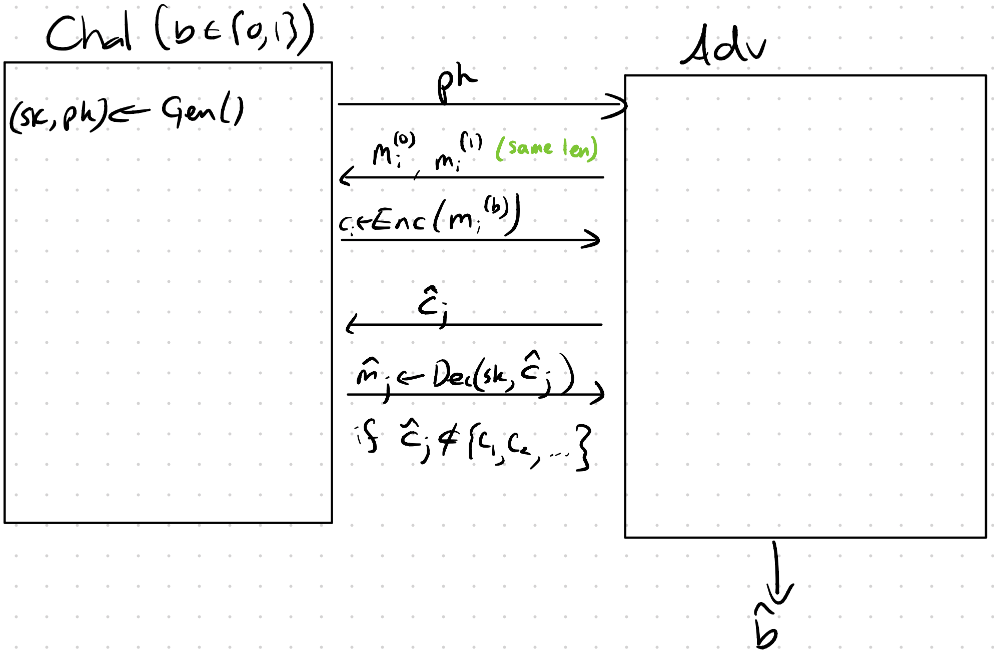
$\text{Enc}(pk, m) \to ct$

$\text{Dec}(sk, ct) \to m \text{ or } \perp$

} Different key for enc & dec – big idea

As before, we want correctness (not shown) & security

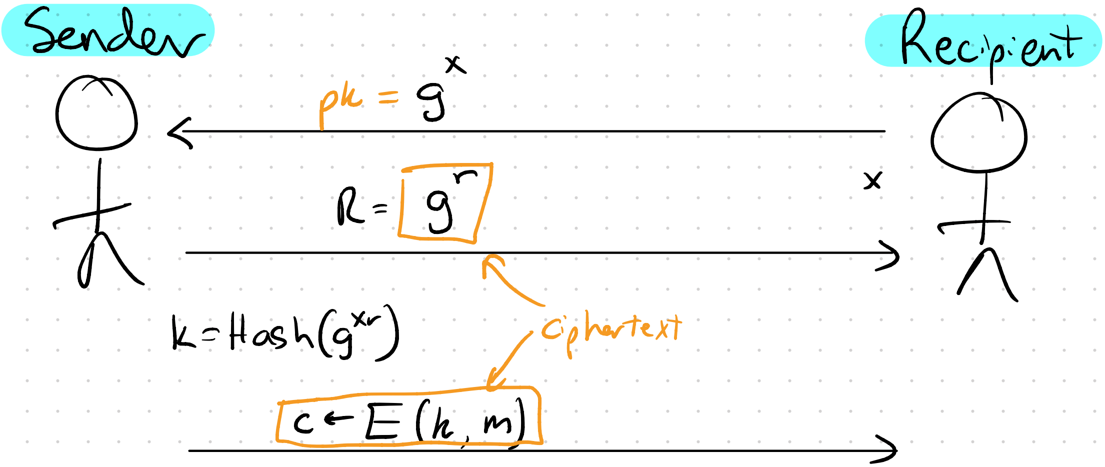
* CCA security defn is as in symmetric-key setting except that adv gets public key



PKE from Key exchange - ElGamal Encryption

Uses:

- CCA-secure sym-key enc scheme (E, D), keyspace $\mathcal{K}$
- Hash fn $\text{Hash}: \mathbb{Z}_p^* \to \mathcal{K}$, model as random oracle



$$\text{Gen}() \to (x, g^x) \quad \text{for } x \overset{R}{\leftarrow} \mathbb{Z}_q$$

$$\text{Enc}(pk = g^x, m) := \begin{cases} r \overset{R}{\leftarrow} \mathbb{Z}_q \\ \text{Output } (R = g^r, E(\text{Hash}(g^{xr}), m)) \end{cases}$$

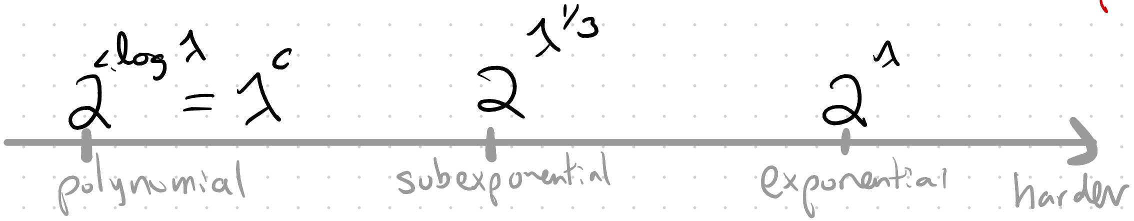
$$\text{Dec}(sk = x, (R, c)) := \begin{cases} k \leftarrow \text{Hash}(R^x) = \text{Hash}(g^{xr}) \\ \text{output } D(k, c) \end{cases}$$

How hard is dlog in $\mathbb{Z}_p^*$?

- When order q is "$\underbrace{\text{smooth}}_{\text{factors into small primes}}$" ⇒ dlog easy (poly time)

- When order q is λ-bit prime, best classical alg runs in "subexponential" time $2^{\lambda^{1/3}}$ - - "index calculus"

- Shor's alg solves dlog in poly time on theoretical quantum computer



Much faster than brute force 2^{λ} attack

⇒ Have to set $\lambda \gg 128$ to make best attack take time $\gg 2^{128}$.

In practice $\lambda = 2048, 4096$.

$$T(\lambda) \approx \exp\left[1.923 \cdot (\ln 2^{\lambda})^{1/3} \cdot (\ln \ln 2^{\lambda})^{2/3}\right]$$

LKT '13 paper on "universal security"

⇒ Small improvement in $\mathbb{Z}_p^*$ dlog alg or factoring alg could require us to use gigantic keys to get 128-bit security. ☹

Elliptic Curve Crypto

- The existence of subexponential-time dlog algs in $\mathbb{Z}_p^*$ (+ factoring) motivates search for alternatives
- Idea: Generalize DH key agreement to setting in which dlog is much harder ($2^{n/2}$ time).

$$g^x \in \mathbb{Z}_p^* = \overbrace{\underbrace{g}_{} \cdot g \cdot g \underbrace{\cdot}_{} \ldots \cdot g}^{x \text{ times}} \pmod p$$

Instead of int, use another object

Instead of mul mod p, use same op on objects.

Koblitz & Miller proposed this generalization indep in 1985

- * Now is the standard.
  - ↳ All modern protocols use ECC for key ex & sigs

Arithmetic on elliptic curves

Public params
- prime $p \approx 2^{256}$
- group order $q \approx p$ — typically q is prime
- constants $A, B \in \mathbb{F}_p$ $\}$ ints mod p w/ + and *

As with $\mathbb{Z}_p^*$, need to pick params carefully

Object: points $(x,y) \in \mathbb{F}_p \times \mathbb{F}_p$

s.t. $y^2 = x^3 + Ax + B$

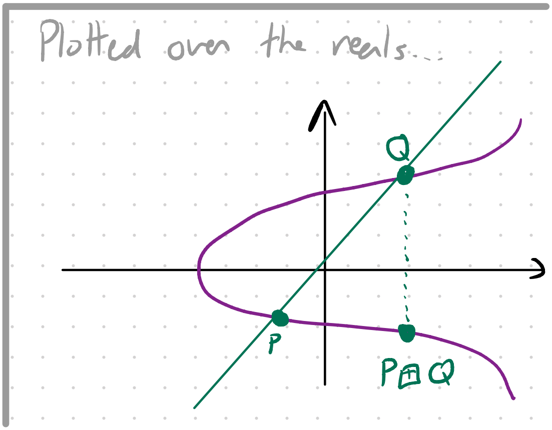
Operation on points

$P \boxplus Q$
- draw line through
- flip over x-axis

$\}$ Takes ≈ 15 $\mathbb{F}_p$ ops to compute

Lots of history & depth here.

Many clever optimizations and tweaks to improve security.


Plotted over the reals...

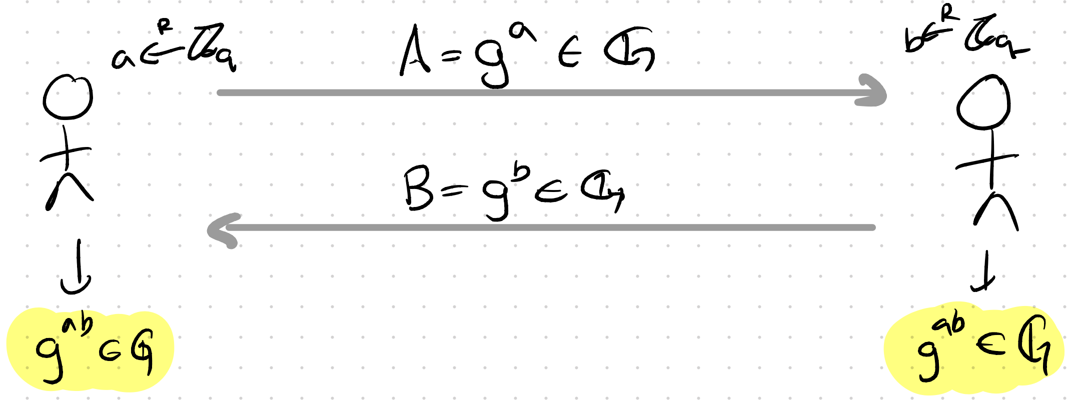
Point addition is "nice"

- assoc, commut, identity, inverse ⇒ Commutative group
- As with $\mathbb{Z}_p^*$, we define order q to be
- Let $\mathbb{G}$ be points on curve in $\mathbb{F}_p^2$.
- Define

$$g^x \in \mathbb{G} \equiv \underbrace{g \boxplus g \boxplus g \boxplus g \boxplus \cdots \boxplus g}_{x \text{ times}}$$

Order is smallest q s.t. $g^q = 1_{\mathbb{G}}$ for all $g \in \mathbb{G}$.

ECDH – DH over elliptic curve



$a \xleftarrow{R} \mathbb{Z}_q$

$A = g^a \in \mathbb{G}$

$b \xleftarrow{R} \mathbb{Z}_q$

$B = g^b \in \mathbb{G}$

$g^{ab} \in \mathbb{G}$

$g^{ab} \in \mathbb{G}$

EC Dlog Given (p, q, A, B, g) and g^x for $x \xleftarrow{R} \mathbb{Z}_q$

Find x.

Why we like elliptic curves.

- Group order q (size of secret) is $\simeq p$.
- Best ECDlog alg on certain curves runs in time $2^{\lambda/2}$ when q is a λ-bit prime
  - ↳ Can take $\lambda = 256$ (vs. $\lambda = 2048$ for $\mathbb{Z}_p^*$)
    - ↳ Ops are faster, keys are shorter.
      (as $\lambda \to \infty$)

Neat trick: Point Compression

- Naively, each EC element is (x,y) point 2×256 bits
- We have $y^2 = x^3 + Ax + B$.
  - ↳ Given an x, there are only 2 possible ys

$$y = \pm\sqrt{x^3 + Ax + B} \in \mathbb{F}_p$$

modular sq root

- Represent point (x,y) as $(x, \text{sign}(y)) \Rightarrow 256 + 1$ bits

Subtleties to elliptic curves

* EC-DSA signatures consist of pair (α, β) where β is the x-coord of EC point

$\Rightarrow$ If (α, β) is valid sig on m, so is $(\alpha, -\beta)$! ("malleability")

* When receiving an ostensible EC point from network, need to check that it's on curve before using it.

* Many mathematically equiv ways to represent curve. Some are algorithmically better than others.

Edwards form $\quad x^2 + y^2 = 1 + \underset{\uparrow \text{ parameter}}{d} x^2 y^2 \in \mathbb{F}_p$

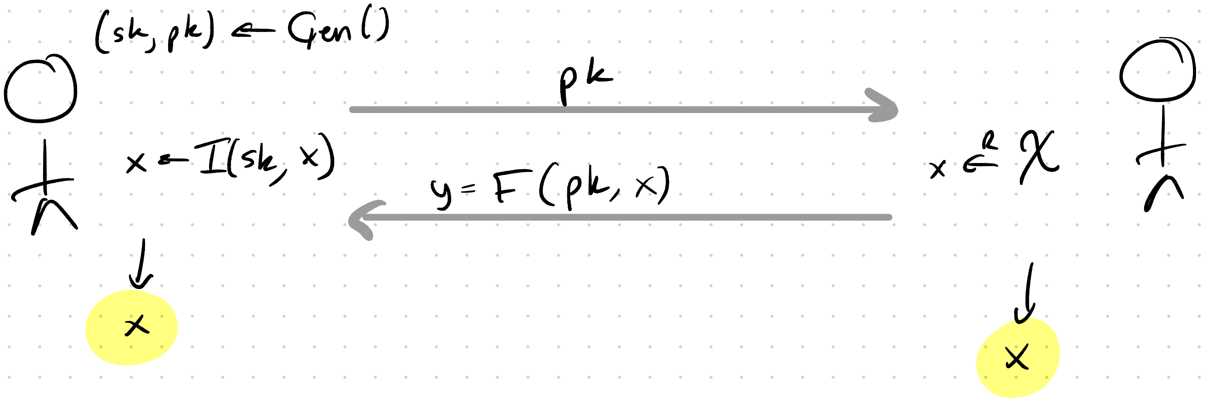
The end!

Can also build key exchange from trapdoor OWP (Gen, F, I) over $\mathcal{X}$

$Gen() \rightarrow (sk, pk)$

$F(pk, x) \rightarrow y \in \mathcal{X}$

$I(sk, y) \rightarrow x \in \mathcal{X}$



Not used as commonly as DH b/c of cost of generating keys:

RSA: λ^4

DH: λ^3

} when $\lambda \approx 2000$ this is a big gap.

2048-bit keygen DSA: 0.22 ms

RSA: 68 ms