Asymmetric cryptography from discrete logarithms

Benjamin Smith

Summer school on real-world crypto and privacy
Sibenik, Croatia // June 17 2019

Inria + Laboratoire d'Informatique de l'École polytechnique (LIX)

Asymmetric crypto settings

It's time to look at **asymmetric cryptosystems**, especially **signatures** and **key exchange**.

Unlike symmetric systems, asymmetric cryptosystems almost always[1] have some **algebraic** object at their core, such as

- Cyclic **groups** (from finite rings and elliptic curves)
- **Codes** from coding theory
- Euclidean **lattices**
- Multivariate **polynomial systems**

Security comes from the computational difficulty of some algorithmic problem in the object.

[1] Hash-based signatures are a notable exception.

Groups

Asymmetric crypto: groups

Today we concentrate on the simplest option:
discrete-log-based crypto in a finite commutative group $\mathcal{G}$
(in the end, $\mathcal{G}$ will generally be cyclic of prime order).

We write the group law in $\mathcal{G}$ **additive**ly: eg. $P \oplus Q = R$

Scalar multiplication (exponentiation):

$$[m] : P \longmapsto \underbrace{P \oplus \cdots \oplus P}_{m \text{ copies of } P}$$

for any m in $\mathbb{Z}$ (with $[-m]P = [m](\ominus P)$).

Computing $(m, P) \mapsto [m]P$ is efficient: $O(\log m)$ operations in $\mathcal{G}$.

Naive scalar multiplication: double-and-add

Algorithm 1: Naive scalar multiplication via double-and-add

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$, $P \in \mathcal{G}$
Output: $[m]P$

1 $R \leftarrow 0_{\mathcal{G}}$
2 **for** $i := \beta - 1$ ***down to*** 0 **do** invariant: $R = [\lfloor m/2^i \rfloor]P$
3 $\quad R \leftarrow [2]R$
4 $\quad$ **if** $m_i = 1$ **then**
5 $\quad\quad R \leftarrow R \oplus P$
6 **return** R // $R = [m]P$

Virtually *all* scalar multiplications involve $m \sim \#\mathcal{G}$.
They are therefore relatively **intensive operations**.

The Discrete Logarithm Problem (DLP)

Inverting scalar mult. is the **Discrete Logarithm Problem** (DLP):

Given P and $Q = [m]P$ in $\mathcal{G}$, compute m.

Oversimplified picture of group-based cryptography:

Public keys are group elements

Private keys are scalars in $\mathbb{Z}/N\mathbb{Z}$

Security: breaking a keypair means solving a DLP instance

Discrete logarithms in generic groups

Concretely: the DLP in any $\mathcal{G}$ is in $O(\sqrt{N})$.

Well-known algorithms include:

- Shanks' **baby-step giant-step**: $O(\sqrt{N})$ time and space.
  A classic space-time tradeoff.
- **Pollard's** ρ algorithm: $O(\sqrt{N})$ time, low space.
  Probabilistic algorithm based on pseudorandom walks.

More efficient algorithms to attack DLP instances in $\mathcal{G}$ may exist, **depending on the concrete realization** of $\mathcal{G}$.

For example: the DLP in the additive group $(\mathbb{Z}/N\mathbb{Z}, +)$ *is solved by the extended Euclidean algorithm.*

Discrete logarithms in black-box groups

In the **abstract**, the DLP is **exponentially hard**.

Shoup's theorem[2]: if $\mathcal{G}$ is a **black-box** group, then solving random instances of the DLP in $\mathcal{G}$ requires **at least** $\Omega(\sqrt{p})$ operations in $\mathcal{G}$, where p is the largest prime divisor of N.

For $\mathcal{G}$ of prime order p, this means the DLP is in $\Theta(\sqrt{p})$.

[2] See the appendix for a more precise statement.

Pohlig–Hellman: reduction to the prime-order case

Theorem (Pohlig and Hellman)
Suppose we know the prime factorization $\#\mathcal{G} = N = \prod_{i=1}^{n} p_i^{e_i}$.
Then we can solve DLP *instances in* $\mathcal{G}$ *in*

$$O\Big(\sum_{i=1}^{n} e_i(\log N + \sqrt{p_i})\Big)$$

$\mathcal{G}$*-operations.*[3]

The vital observation is that the DLP in $\mathcal{G}$ is essentially only as hard as the DLP in the largest prime-order subgroup of $\mathcal{G}$:
or, $\mathcal{G}$ is **only as secure as its largest prime-order subgroup**.

[3] See the appendix for details

Keypairs

Asymmetric keys come in matching (Public,Private) **pairs**.

- a public key poses an individual mathematical problem;
- the matching private key gives the solution.

Here, keypairs present instances of the DLP in $\mathcal{G} = \langle P \rangle$:

$$(\text{Public}, \text{Private}) = (Q, x) \quad \text{where} \quad Q = [x]P\,.$$

Cryptanalysis can begin as soon as a public key is "bound to" (i.e. published), *not* once either key is actually used!

Note that it can be *much* easier to attack sets of keys than to attack individual keys.

The challenge

We want to construct **cryptographically efficient** groups, in the sense that they are

compact: lots of group per bit;

fast: easy to compute scalar multiplications; and

secure: hard DLPs relative to their size.

Natural candidates: **algebraic groups** over finite fields $\mathbb{F}_q$.

- Elements are tuples of elements of $\mathbb{F}_q$,
- Group operations are defined by polynomial functions.

Examples: finite fields, elliptic curves, ...

Concrete groups

For k-bit security against generic algorithms, prime $\#\mathcal{G} \sim 2^{2k}$.

More efficient algorithms to attack DLP instances in $\mathcal{G}$ may exist, **depending on the concrete realization** of $\mathcal{G}$; parameters must be adjusted accordingly.

Example: Suppose $\mathcal{G} \subset \mathbb{F}_p^\times$, targeting 128-bit security. Then

1. $\#\mathcal{G}$ must be (a multiple of) a ~ 256-bit prime to defeat generic discrete log algorithms
2. p must be a ~ 3072-bit prime to defeat the finite-field-specific Number Field Sieve algorithm

Elliptic curves

Elliptic curves

Elliptic curves are a convenient source of groups that can **replace multiplicative groups** in asymmetric crypto.

Classic **"short" Weierstrass model**:

$$\mathcal{E}/\mathbb{F}_p : y^2 = x^3 + ax + b \quad \text{with} \quad a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0\,.$$

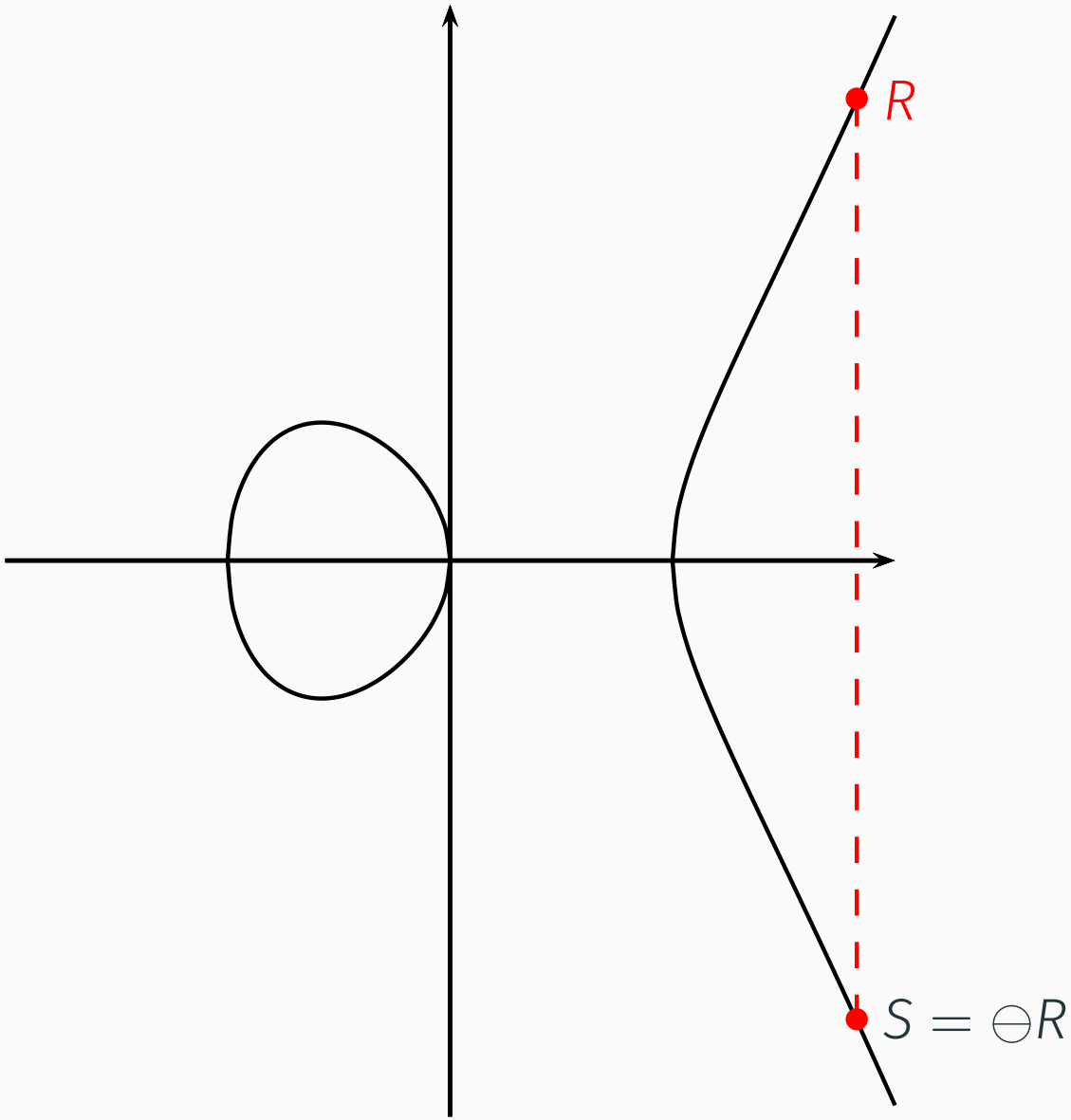
The **points** on $\mathcal{E}$ are

$$\mathcal{E}(\mathbb{F}_p) = \left\{(\alpha, \beta) \in \mathbb{F}_p^2 : \beta^2 = \alpha^3 + a \cdot \alpha + b\right\} \cup \{\mathcal{O}_\mathcal{E}\}$$

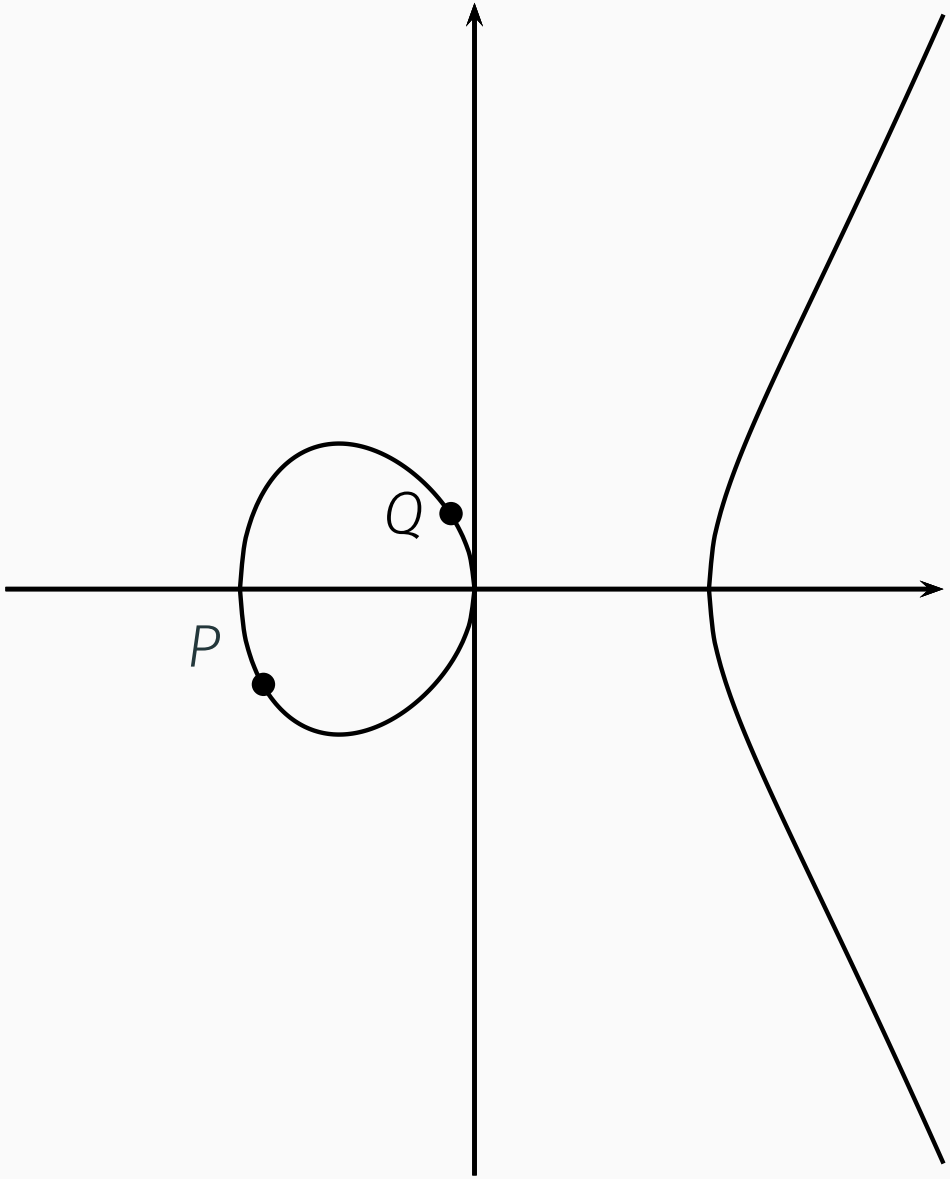
where $\mathcal{O}_\mathcal{E}$ is the unique **"point at infinity"**.

$\mathcal{E}(\mathbb{F}_p)$ is an algebraic group, with $\mathcal{O}_\mathcal{E}$ the identity element.

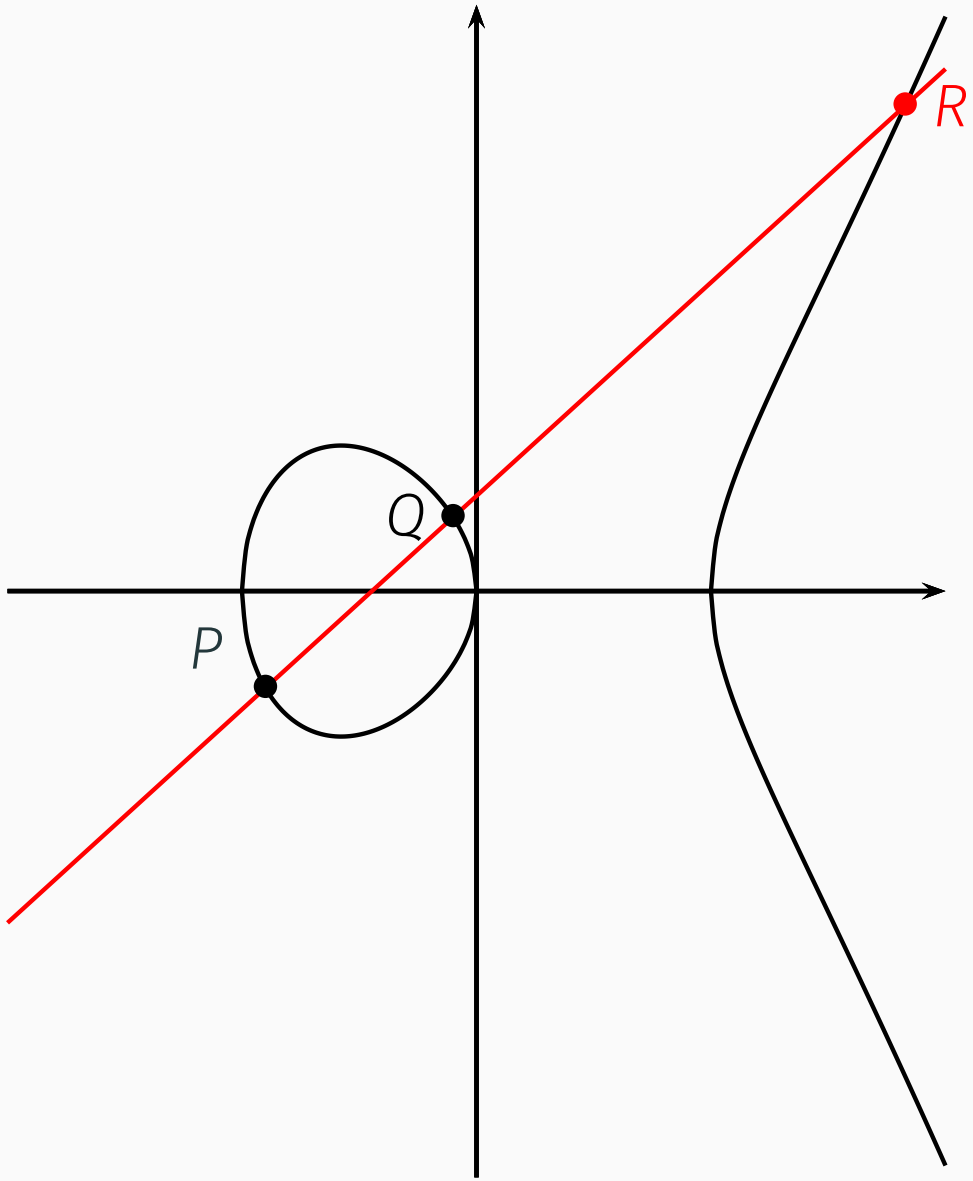
Elliptic curve negation: $\ominus R = S$



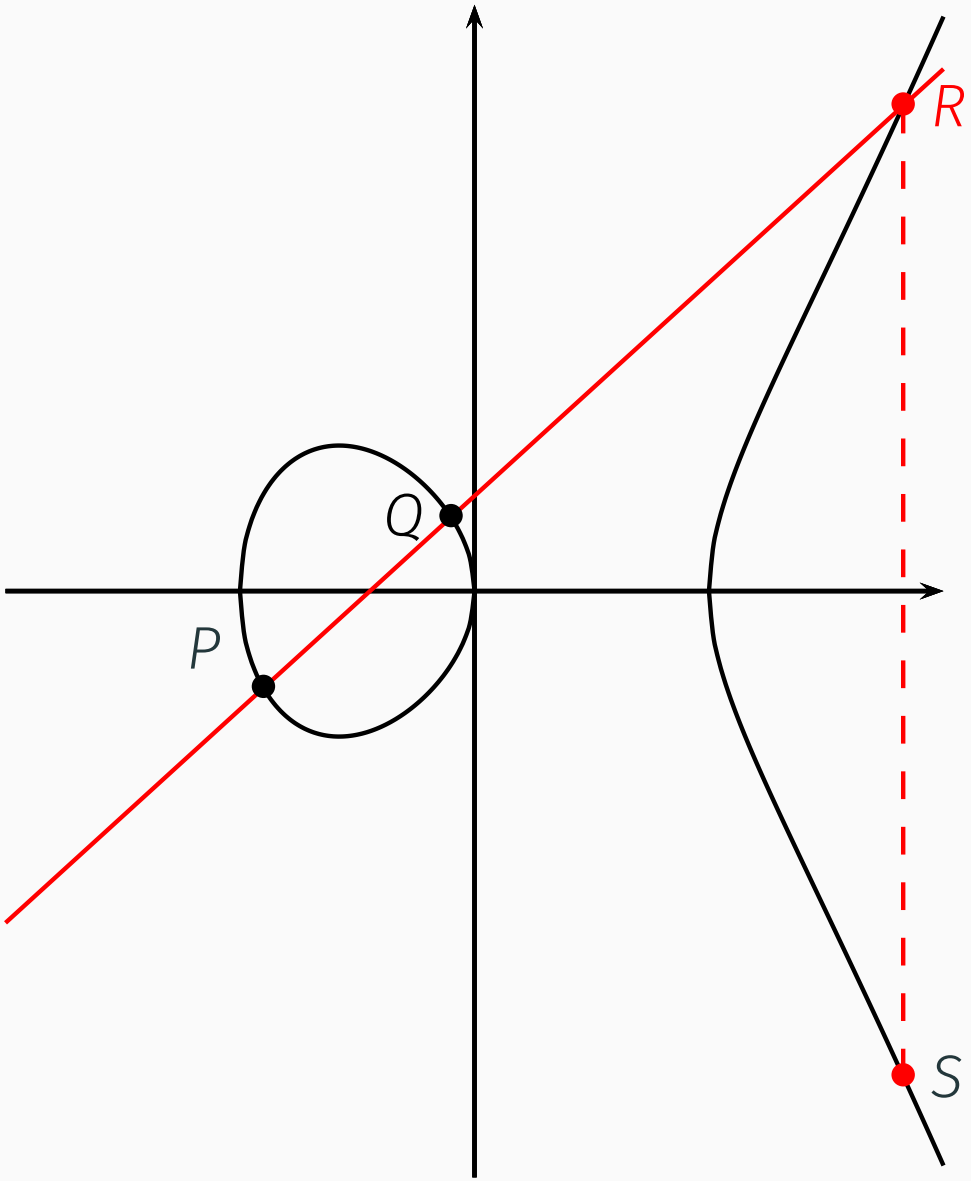
Elliptic curve addition: $P \oplus Q =?$



Elliptic curve addition: $P \oplus Q \oplus R = 0$



Elliptic curve addition: $P \oplus Q = \ominus R = S$



Elliptic curve group operations

If $P = Q$, the **chord** through P and Q degenerates to a **tangent**.

The important thing is that elliptic curve group operations, being geometric, have **algebraic expressions**.

$\Longrightarrow$ They can be computed as a series of $\mathbb{F}_p$-operations, which can in turn be reduced to a series of machine instructions.

Operations on $\mathcal{E}/\mathbb{F}_p : y^2 = x^3 + ax + b$:

Negation: $\ominus(x, y) = (x, -y)$ and $\ominus\mathcal{O}_\mathcal{E} = \mathcal{O}_\mathcal{E}$

Addition (special cases):

$$(x, y) \oplus \mathcal{O}_\mathcal{E} = (x, y) \qquad \text{and} \qquad (x, y) \oplus (x, -y) = \mathcal{O}_\mathcal{E}\,.$$

Elliptic curve point addition

General addition: write $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$,

For $P \neq \pm Q$, we have $P \oplus Q = (x_\oplus, y_\oplus)$ where

$$x_\oplus = \lambda^2 - (x_P + x_Q) \quad \text{and} \quad y_\oplus = -\lambda(x_\oplus + \mu)$$

where

$$\lambda = (y_P - y_Q)/(x_P - x_Q)$$

is the "slope" of the line through P and Q, and

$$\mu = (x_P y_Q - x_Q y_P)/(x_P - x_Q)\,.$$

Observe: the curve constants a and b do not appear!

Elliptic curve point doubling

Doubling is an extremely important special case.

We have

$$[2]P = P \oplus P = (x_{[2]P}, y_{[2]P})$$

where

$$x_{[2]P} = \frac{(3x_P^2 + a)^2 - 8x_P(x_P^3 + ax_P + b)}{4(x_P^3 + ax_P + b)}$$

and

$$y_{[2]P} = \frac{x_P^3 - ax - 2b - (3x_P^2 + a)x_{[2]P}}{2y_P}.$$

In practice we do all this using **projective coordinates** to avoid expensive divisions in $\mathbb{F}_p$ (see the appendix).

Group orders and structures

Intuitively: $\mathcal{E}$ is 1-dimensional over $\mathbb{F}_p$, so it should have $O(p)$ points. In fact, **Hasse's theorem** tells us that

$$\#\mathcal{E}(\mathbb{F}_p) = p + 1 - t \quad \text{where} \quad |t| < 2\sqrt{p}\,.$$

The possible group structures are limited:

$$\mathcal{E}(\mathbb{F}_p) \cong \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z} \quad \text{where} \quad m \mid \gcd(n, p-1)\,.$$

The **Hasse interval** $(p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p})$ contains many primes. Generating prime/near-prime order curves is routine[4].

Outside research, use **standardized** secure curve parameters.

[4] Though this requires some highly nontrivial algorithms!

The Elliptic Curve Discrete Logarithm Problem (ECDLP)

Amazing fact: for subgroups $\mathcal{G}$ of **general**[5] **elliptic curves**, we still do not know how to solve discrete logs significantly faster than by using **generic black-box group algorithms**.

In particular: currently, for prime-order $\mathcal{G} \subseteq \mathcal{E}(\mathbb{F}_p)$, we can do no better than $O(\sqrt{\#\mathcal{G}})$.

Apart from improvements in distributed computing, and a constant-factor speedup of about $\sqrt{2}$, there has been **absolutely no progress** on general ECDLP algorithms. Ever.

Current world record for prime-order ECDLP: in a 112-bit group, which is a *long* way away from the 256-bit groups we use today!

[5] That is, for all but a very small and easily identifiable subset of curves.

Why do we use elliptic curves?

Targeting **k bits of security:**

- Let p be a $2k$-bit prime.
- Let $\mathcal{E}/\mathbb{F}_p$ be an (almost)-prime order elliptic curve over $\mathbb{F}_p$.
- Let $\mathcal{G} \subseteq \mathcal{E}(\mathbb{F}_p)$ be the prime-order subgroup, $\#\mathcal{G} \sim p \sim 2^{2k}$.

Now **public and private keys** only require $\sim$ **2k bits each**.

Beats 3072-bit public keys in $\mathbb{F}_p^\times$.
The group operations are also *much* faster.

The take-home: elliptic curves simply offer **the shortest keys** at any given security level.

Identification

Identity

Identity means

- being distinguishable from everyone else
- **holding the private key** corresponding to a public key

We want **authentication**: cryptographically **identifying** the other participant(s) in a protocol, by verifying a proof that they hold the secret x corresponding to a given public $Q = [x]P$.

In **symmetric** crypto, MACs and AEAD can authenticate **data**, but **not communicating parties**, because *both sides hold the same secret*—and a shared identity is no identity.

How do you prove your identity?

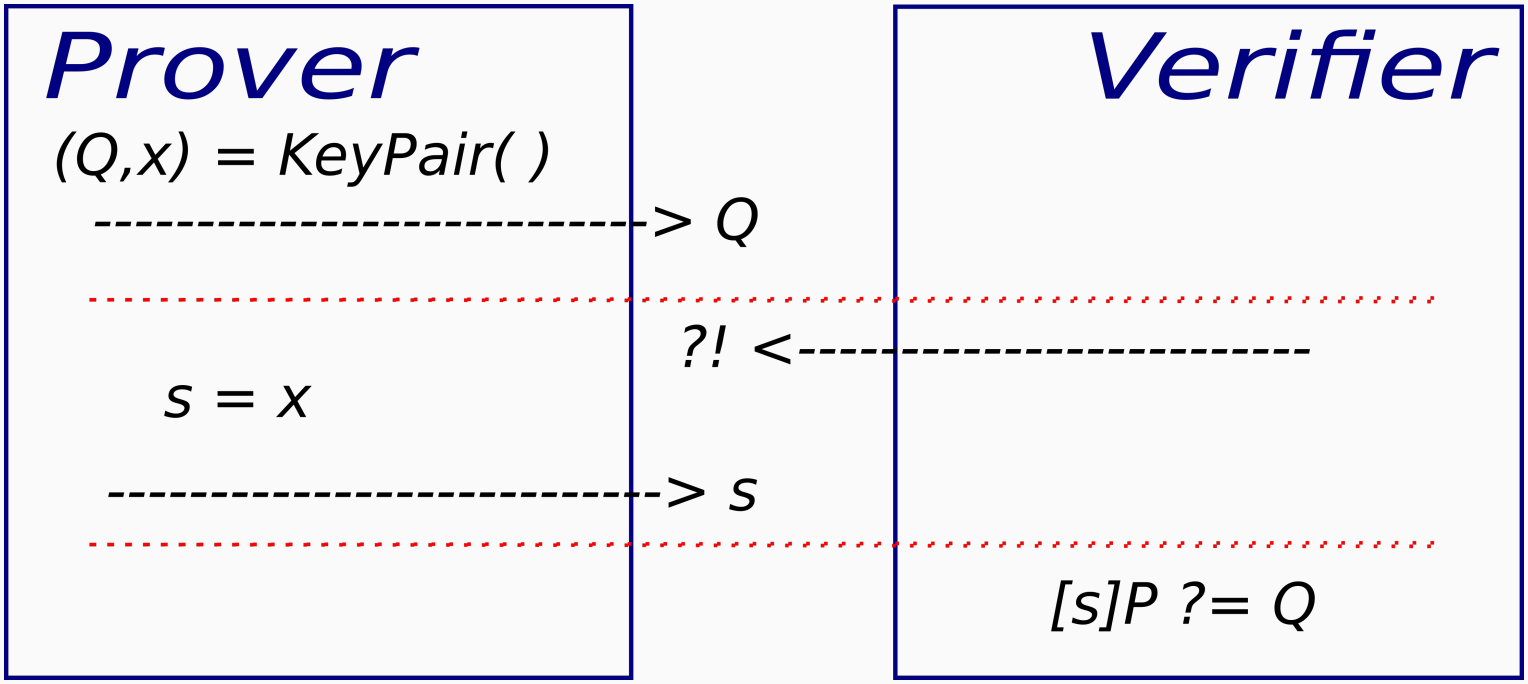
In our setting, you assert or claim an identity by binding to (that is, publishing and committing to) a public key Q from a keypair $(Q = [x]P, x)$.

Prove your identity $\iff$ prove you know x.

To formalize this, we introduce three characters:

Prover	wants to *prove* their identity
Verifier	wants to *verify* the identity of Prover
Simulator	wants to impersonate Prover

Ineffective identification

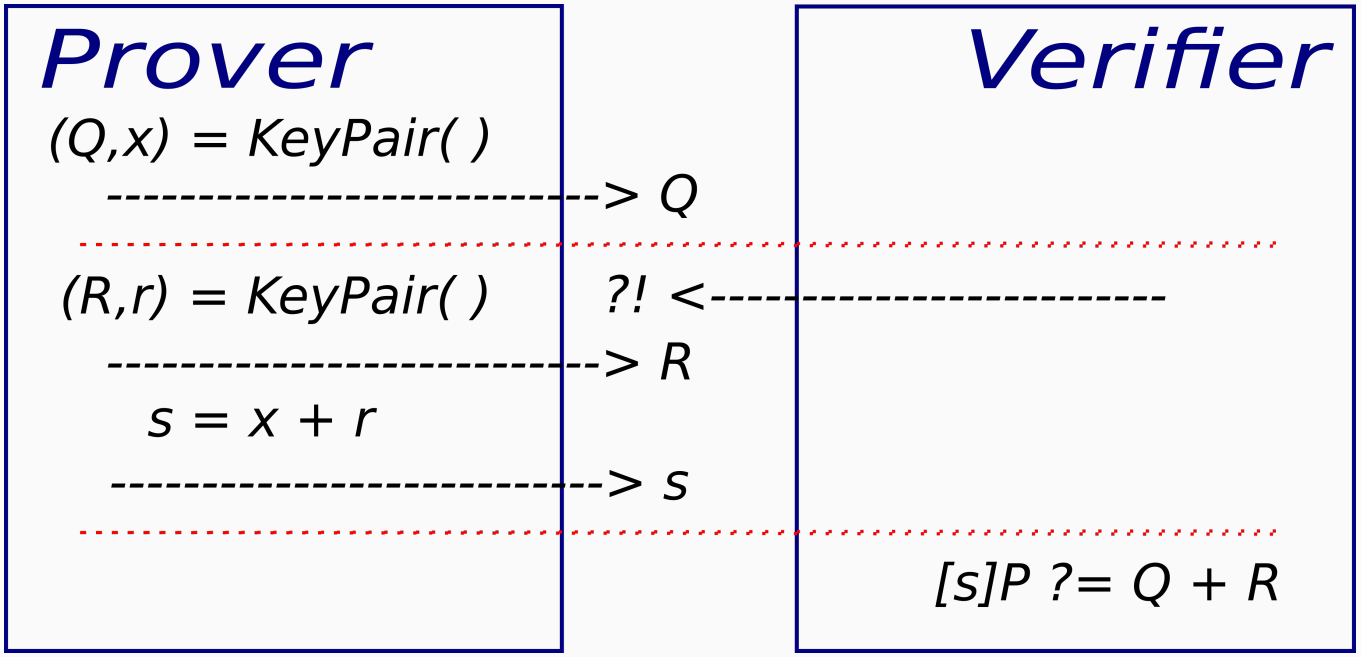


1. Verifier challenges;
2. Prover returns x as s;
3. Verifier accepts iff $[s]P = Q$.

Problem: Prover no longer has an identity, because they gave away their secret x.

Using ephemeral keys

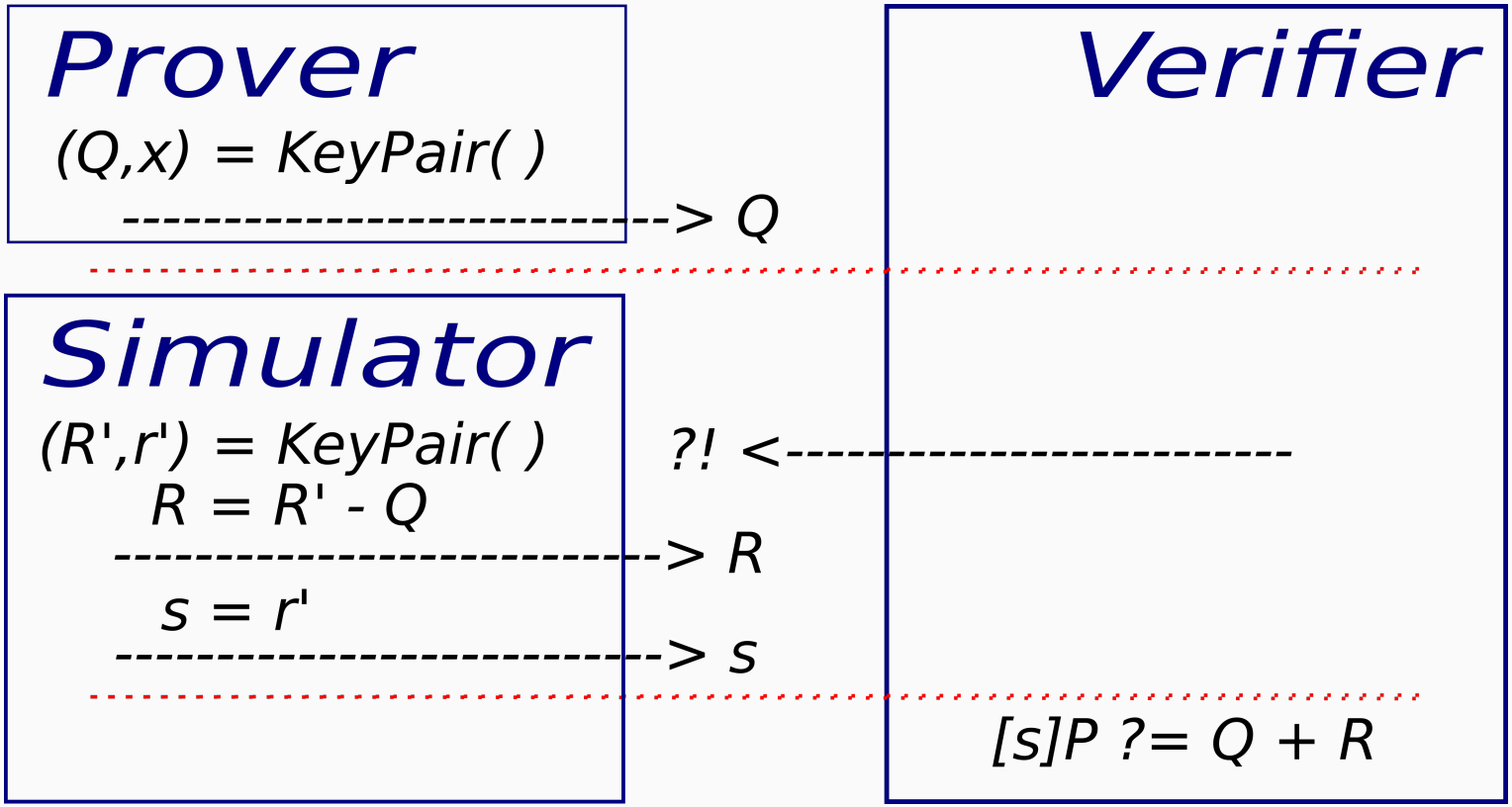
Trick: hide long-term secrets with disposable one-shot secrets.



1. Prover generates an *ephemeral* keypair (R, r), **commits** R;
2. Verifier **challenges**;
3. Prover **responds** by sending R and $s = x + r$ to Verifier.
   s reveals nothing about x, because r is random
4. Verifier accepts iff $[s]P = Q + R$ (which is $[x]P + [r]P$).

Cheating

Problem: Simulator can easily impersonate Prover.



Verifier accepts because $[s]P = [r']P = R' = Q + R$

Note: *Simulator never knows x—nor the log of R, because otherwise they would know x!*

Detecting cheating

How can Verifier detect this cheating, and thus distinguish between Prover and Simulator?

Prover
- sends $s = x + r = \log(Q + R)$,
- knows *both* $x = \log(Q)$ and $r = \log(R)$.

Simulator
- sends $s = \log(Q + R)$,
- knows *neither* $x = \log(Q)$ *nor* $r = \log(R)$.

The difference: knowledge of x, and knowledge of r.

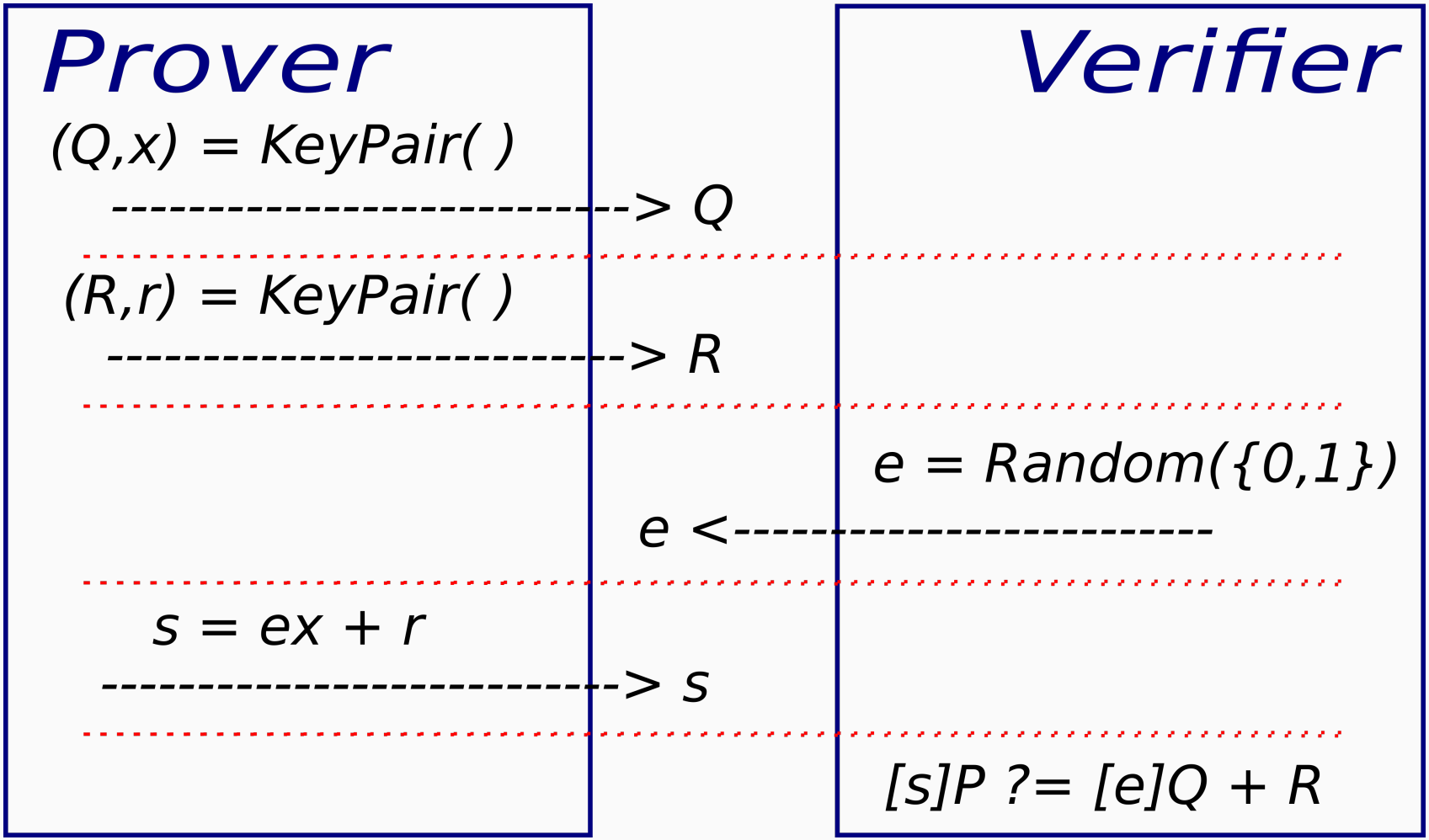
- Verifier can't ask for x.
- Verifier can't ask for the ephemeral secret $r = \log(R)$ because that would also reveal x (since she knows s).

Detecting cheating

Solution: let Verifier ask for **either** s **or** r,
and check either $[s]P = Q + R$ or $[r]P = R$ accordingly.

- correct $s \implies$ I know x, *if* I am honest
- correct $r \implies$ I was honest, but *not* that I know x

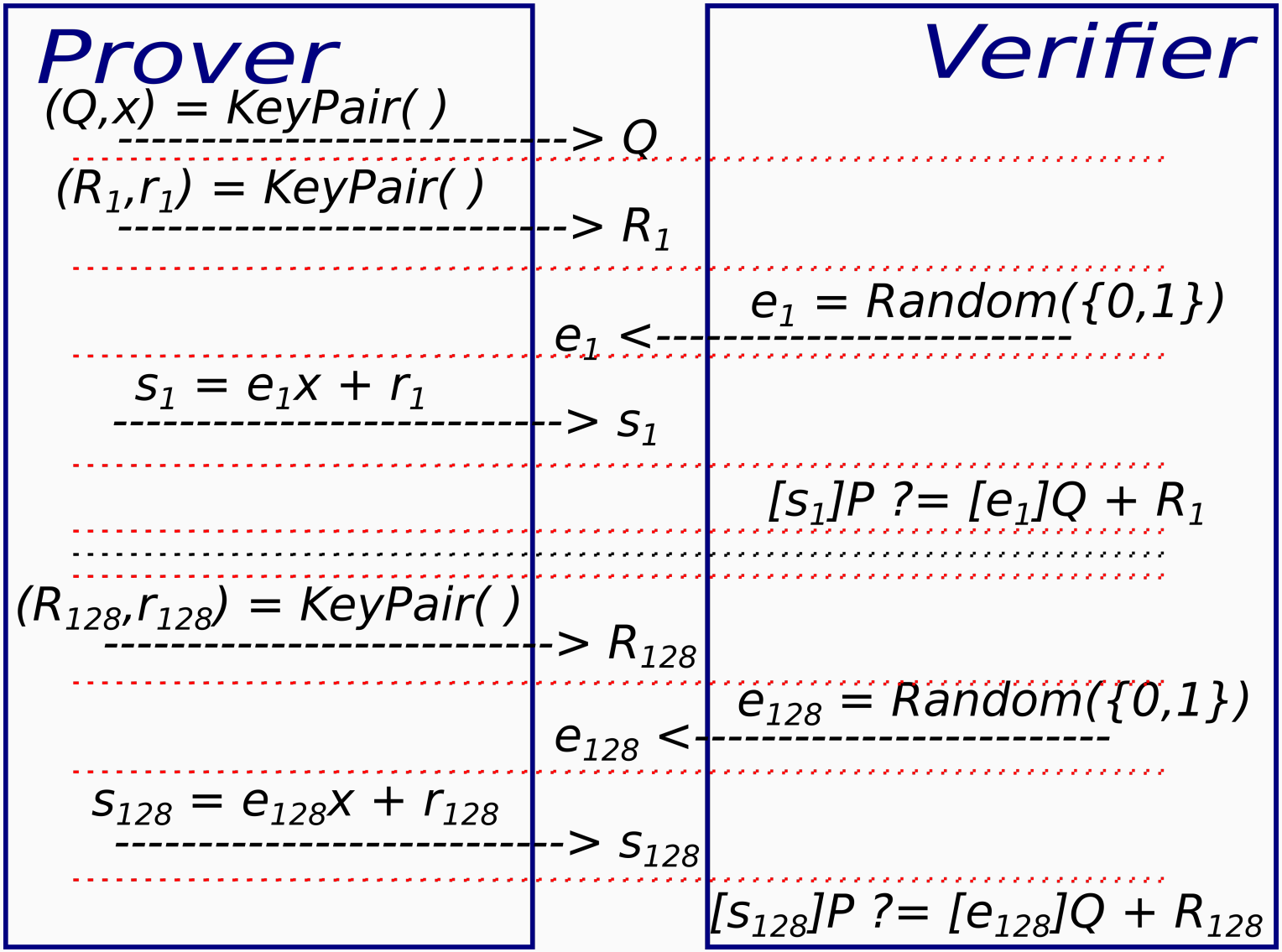
Chaum–Evertse–Graaf (1988)



To cheat, Simulator must guess/anticipate *e*: 50% chance.

So repeat until Verifier is satisfied it's Prover (say 128 rounds).

128 rounds later...



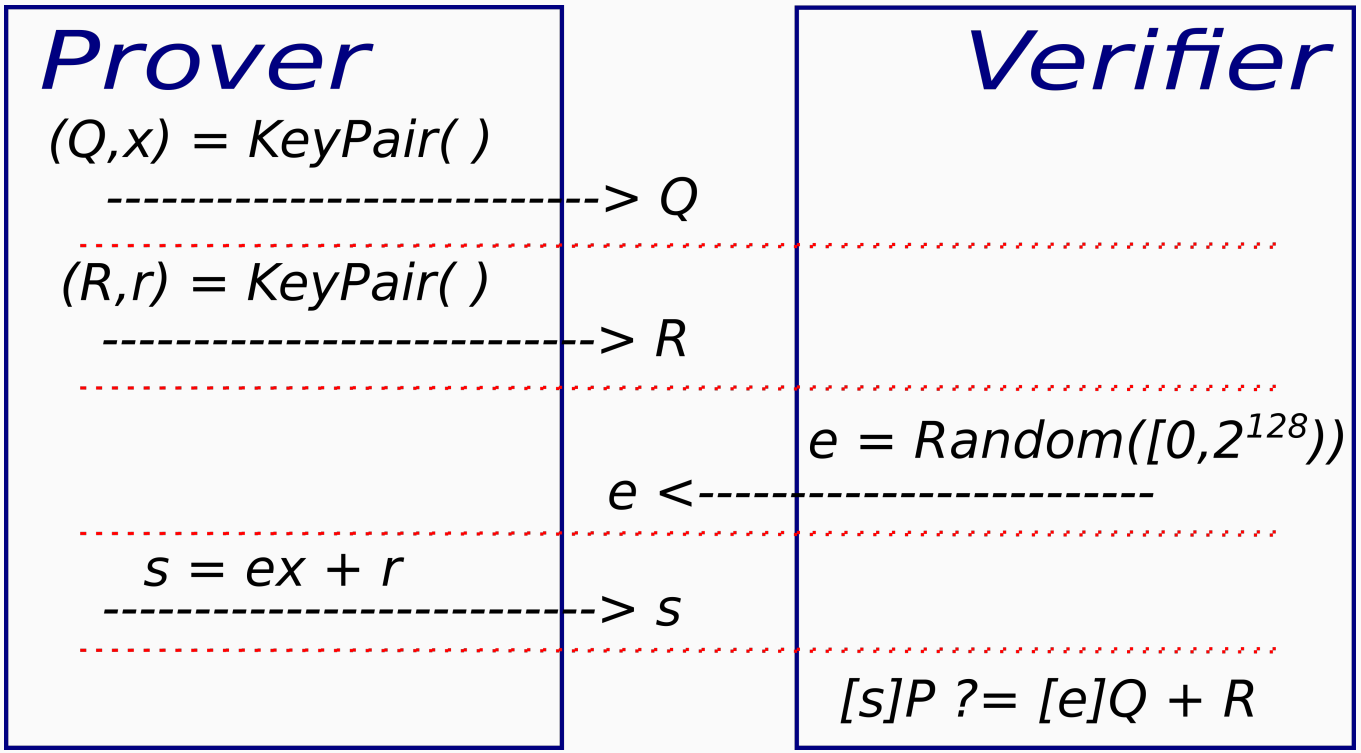
Schnorr ID (1991)

It is extremely inconvenient to run 128 rounds of the Chaum–Evertse–Graaf ID protocol:

1. too many **interactive rounds** of communication (128 challenges and responses),
2. too much **bandwidth** (128× 256-bit group elements and 128 × 256-bit scalars)
3. too much **computation** on each side (128× 256-bit scalar multiplications for both parties!)

Schnorr identification (1991): "**parallelise**" the 128 rounds, replacing 128 one-bit challenges with one 128-bit challenge.

Schnorr ID



Note: s reveals nothing about x, because r is random

Only one round. Prover does one 256-bit scalar multiplication, Verifier does one 256-bit and one 128-bit scalar multiplication.

Signatures

Signatures

A **digital signature** is a **non-interactive proof** that
the Signer witnessed (created, saw) some data.

Authenticity, *message integrity*, *non-repudiability*:

- only the Signer could have created it;
- the Signer could not have created it from any other data; and
- only the Signer's public key is needed to *verify* it.

The Fiat–Shamir transform

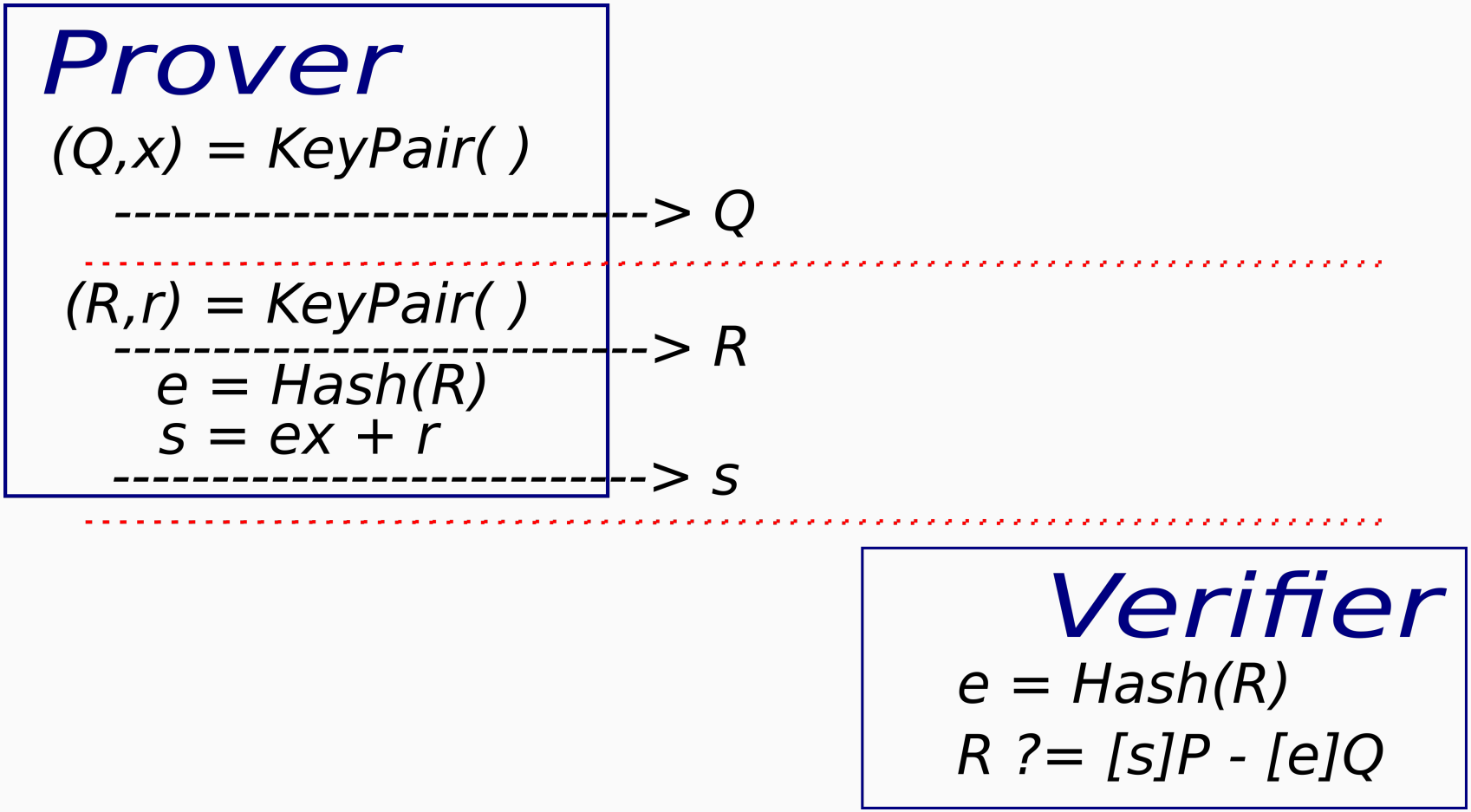
We build **Schnorr signatures** from the Schnorr ID scheme by applying the **Fiat–Shamir transform**:

1. make the ID scheme non-interactive, and
2. have the signer identify themself *to the data* (!)

Formally: **Fiat–Shamir** transforms an interactive proof with public randomness into a non-interactive proof, by replacing the verifier with a cryptographic hash function applied to the protocol's transcript.

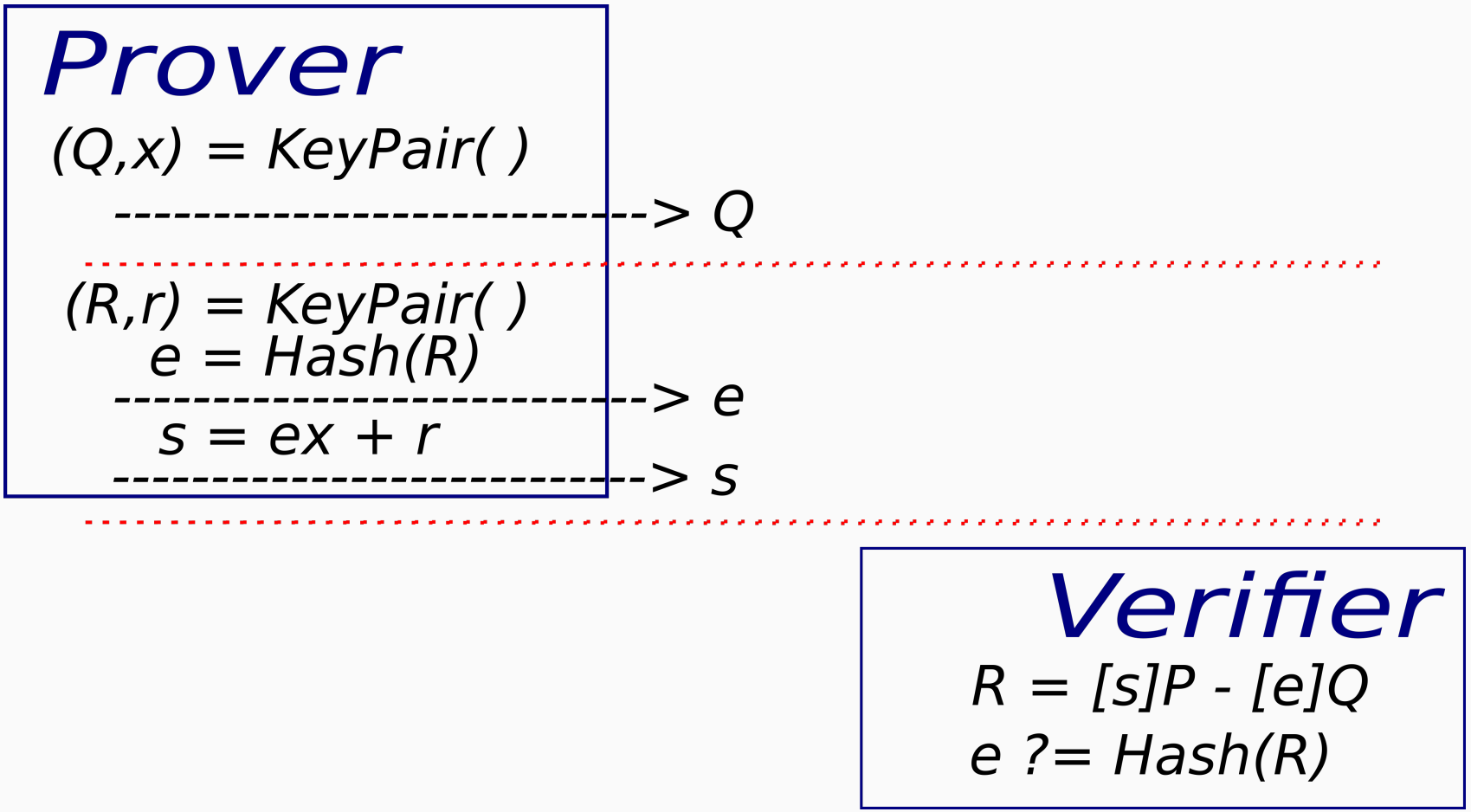
Fiat–Shamir: making Schnorr ID non-interactive

Intuition: the hash of *R* is unpredictable and random-looking, so it can stand in for a true random challenge.



Sending (e, s) instead of (R, s)

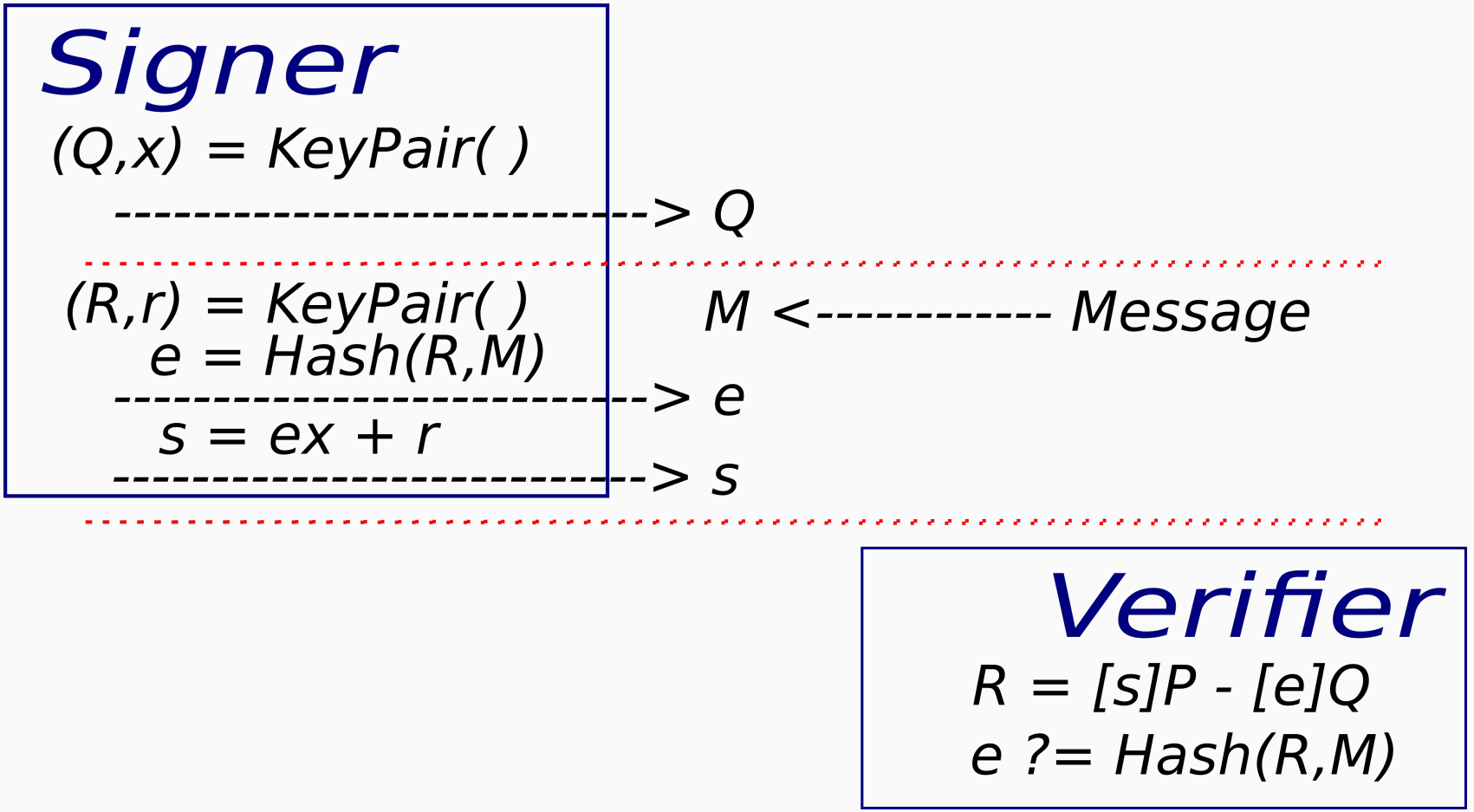
Generally the hash e is smaller than R (especially if $\mathcal{G} = \mathbb{F}^\times$), so we can send (e, s) instead of (R, s) to save some space.



Schnorr signatures (1991): sending (e, s) instead of (R, s)

Hash needs 128 bits of **prefix-second-preimage** resistance.

Traditionally, no need for collision resistance...



Schnorr signatures

Schnorr signatures are proven secure in the **random oracle model** (but not in the **standard model**).

Schnorr **patented** his signature scheme.

As a result, few people actually used it.

(Instead we had the inferior DSA and ECDSA protocols).

The patent **expired in 2008**...

EdDSA (Bernstein–Duif–Lange–Schwabe–Yang, 2012)

EdDSA: a contemporary Schnorr signature variant.
It is **deterministic**: same signer+message $\implies$ same signature.

Fix a 2β-bit hash function H and a secure elliptic curve $\mathcal{E}/\mathbb{F}_p$ with a β-bit prime-order subgroup $\mathcal{G} = \langle P \rangle \subset \mathcal{E}(\mathbb{F}_p)$.

Key gen. Choose a random β-bit string, k.
Let x and y be the β-bit strings s.t. $x \parallel y = H(k)$.
Public key: $Q = [x]P$. **Secret key:** k *(not x)*.

Sign a message M: let $x \parallel y = H(k)$,
$r = H(y \parallel M)$, $R = [r]P$, $s = r + H(R \parallel Q \parallel M)x$.
Signature: (R, s).

Verify a putative signature (R, s) on M under Q:
accept iff $R = [s]P - [H(R \parallel Q \parallel M)]Q$.

Key exchange

The need for key agreement

Key agreement is a fundamental operation in cryptography.

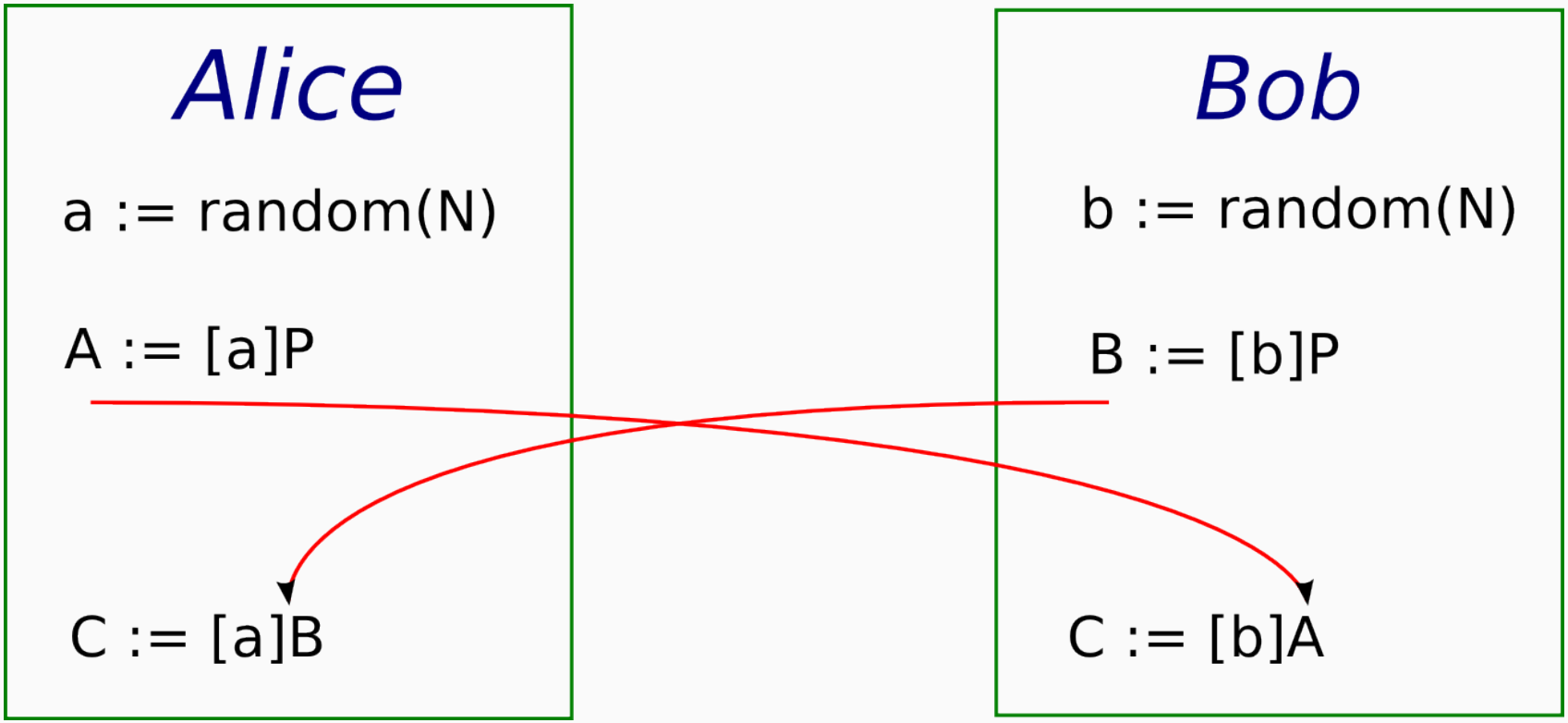
It allows two principals (“Alice” and “Bob”) to establish a shared secret key without prior contact.

The classic protocol for this is **Diffie–Hellman Key Exchange**, historically one of the first asymmetric crypto algorithms.

- **Public discovery:** Diffie and Hellman, 1976
- **Secret discovery:** GCHQ, UK, early 1970s.

More generally, we use **Key Encapsulation Mechanisms** (**KEM**s).

Diffie–Hellman key exchange ($\leq$ 1976)



Correctness: $[a][b] = [b][a] = [ab]$ for all $a, b \in \mathbb{Z}$.

Alice & Bob now use a **KDF** (Key Derivation Function, e.g. HKDF) to derive a shared cryptographic key from the shared secret S.

Warning: no authentication!

The Diffie–Hellman problem

Diffie–Hellman security depends not (directly) on the DLP, but rather on the **Computational Diffie–Hellman Problem (CDHP)**:

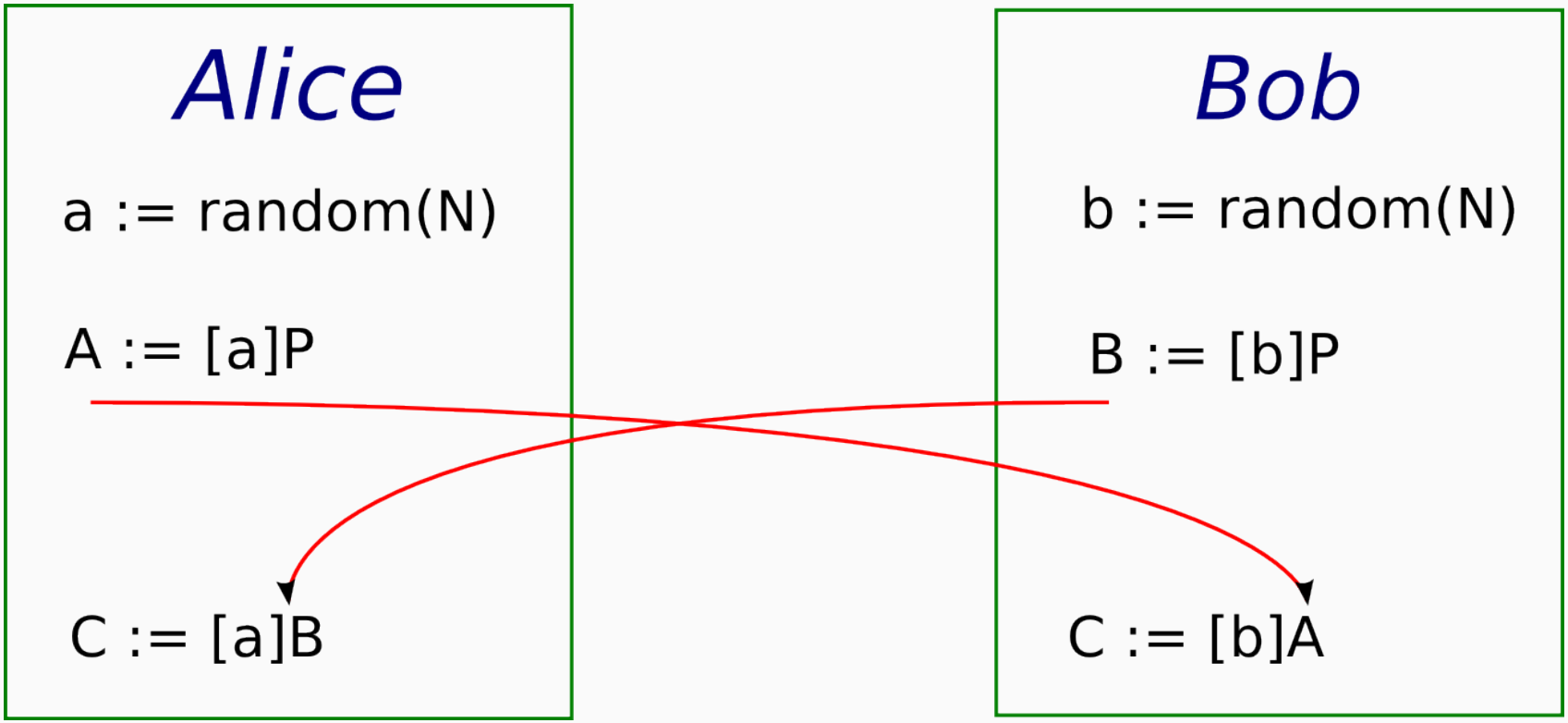
$$\text{Given } (P, Q_A = [x_A]P, Q_B = [x_B]P), \text{ compute } S = [x_A x_B]P.$$

Clearly DLP $\implies$ CDHP.
What about CDHP $\implies$ DLP? **Not obvious!**

- **Conditional polynomial-time** reduction (Maurer–Wolf, …)
- **Unconditional subexponential** reduction for the $\mathcal{G}$ we use in practice (Muzerau–Smart–Vercauteren).

More detail on Maurer: see the appendix

Modern Diffie–Hellman key exchange



Notice **DH never directly uses the group structure** on $\mathcal{G}$.

All we need for DH is a *set* $\mathcal{G}$, and big sets A, B of efficiently samplable and computable functions $\mathcal{G} \to \mathcal{G}$ such that $[a][b] = [b][a]$ for all $[a] \in A$ and $[b] \in B$, and the corresponding CDHP is believed hard.

Modern Diffie–Hellman

Diffie–Hellman does not need a group law, just scalar multiplication; so we can "drop signs" and work modulo $\ominus$.

Elliptic curves: work on x-line $\mathbb{P}^1 = \mathcal{E}/\langle \pm 1 \rangle$.

- The equivalence class $\{P = (x_P, y_P), \ominus P = (x_P, -y_P)\}$ is represented by the x-coordinate $\mathbf{x}(P) = x_P$.
- Projectively: $\mathbf{x}((X : Y : Z)) = (X : Z) \in \mathbb{P}^1$ when $Z \neq 0$, and $\mathbf{x}(\mathcal{O}_\mathcal{E})) = \mathbf{x}((0 : 1 : 0)) = (1 : 0)$.

Advantage: save time and space by ignoring y.

This is how we do DH in the real world today, using Curve25519/X25519.

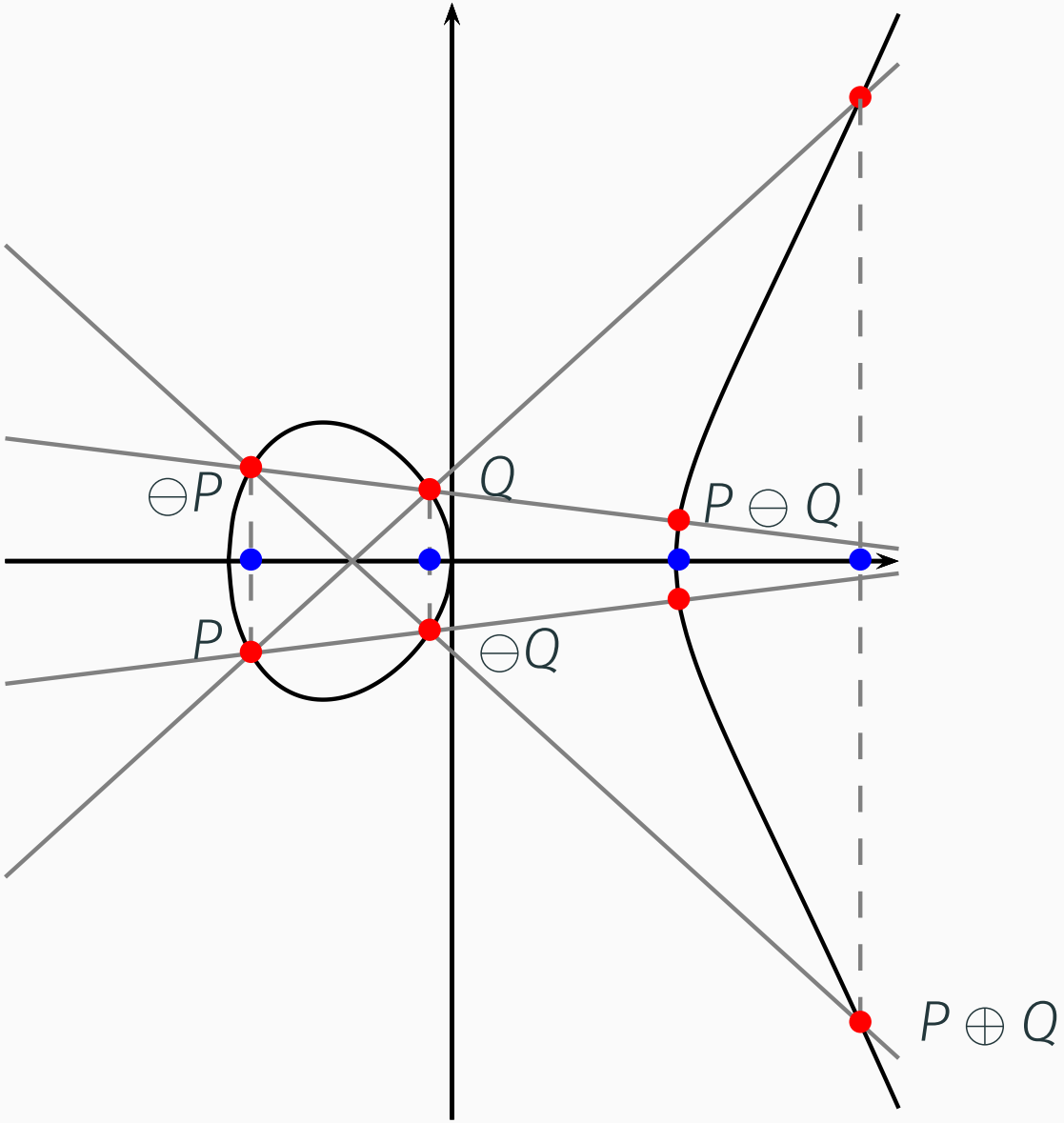
Diffie–Hellman modulo signs

The Diffie–Hellman protocol is now

$$\begin{aligned}
\textbf{Alice}\ &\text{computes } (a, \mathbf{x}(P)) \mapsto x_A = \mathbf{x}([a]P);\\
\textbf{Bob}\ &\text{computes } (b, \mathbf{x}(P)) \mapsto x_B = \mathbf{x}([b]P);\\
\textbf{Alice}\ &\text{computes } (a, x_B) \mapsto x_S = \mathbf{x}([a][b]P);\\
\textbf{Bob}\ &\text{computes } (b, x_A) \mapsto x_S = \mathbf{x}([b][a]P).
\end{aligned}$$

This is *mathematically* well-defined, but we sill need to compute $(m, \mathbf{x}(P)) \mapsto \mathbf{x}([m]P)$ efficiently, *without using* $\oplus$.

Key fact: $\{x(P), x(Q)\}$ determines $\{x(P \ominus Q), x(P \oplus Q)\}$



Pseudo-group operations

Any 3 of $\{\mathsf{x}(P), \mathsf{x}(Q), \mathsf{x}(P \ominus Q), \mathsf{x}(P \oplus Q)\}$ determines the 4th, so we can define

Pseudo-addition:

$$\mathsf{xADD} : (\mathsf{x}(P), \mathsf{x}(Q), \mathsf{x}(P \ominus Q)) \longmapsto \mathsf{x}(P \oplus Q)$$

Pseudo-doubling:

$$\mathsf{xDBL} : \mathsf{x}(P) \longmapsto \mathsf{x}([2]P)$$

We evaluate $\mathsf{x}(P) \mapsto \mathsf{x}([m]P)$ by combining xADDs and xDBLs using **differential addition chains**: scalar mult algorithms where *every* $\oplus$ has summands with **known difference**.

Classic example: the **Montgomery ladder**.

The Montgomery ladder in a group

Algorithm 2: The Montgomery ladder in a group

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and P

Output: $[m]P$

1 $(R_0, R_1) \leftarrow (0, P)$ // Invariant: $R_1 = R_0 \oplus P$

2 **for** i ***in*** $(\beta - 1, \ldots, 0)$ **do** invariant: $R_0 = [\lfloor m/2^i \rfloor]P$

3 if $m_i = 0$ **then**

4 $(R_0, R_1) \leftarrow ([2]R_0, R_0 \oplus R_1)$

5 **else**

6 $(R_0, R_1) \leftarrow (R_0 \oplus R_1, [2]R_1)$

7 **return** R_0 // $R_0 = [m]P$, $R_1 = [m+1]P$

For each addition $R_0 \oplus R_1$, the difference $R_0 \ominus R_1$ is *fixed* (& known in advance!) $\implies$ easy adaptation from $\mathcal{E}$ to $\mathbb{P}^1$.

The Montgomery ladder with pseudo-operations

Algorithm 3: The Montgomery ladder on the x-line $\mathbb{P}^1$

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $\mathbf{x}(P)$

Output: $\mathbf{x}([m]P)$

1 $(x_0, x_1) \leftarrow (\mathbf{x}(0), \mathbf{x}(P))$

2 **for** i *in* $(\beta - 1, \ldots, 0)$ **do**

3 $\quad$ **if** $m_i = 0$ **then**

4 $\quad\quad (x_0, x_1) \leftarrow (\mathbf{xDBL}(x_0), \mathbf{xADD}(x_0, x_1, \mathbf{x}(P))$

5 $\quad$ **else**

6 $\quad\quad (x_0, x_1) \leftarrow (\mathbf{xADD}(x_0, x_1, \mathbf{x}(P)), \mathbf{xDBL}(x_1))$

7 **return** x_0 // $x_0 = \mathbf{x}([m]P)$, $R_1 = \mathbf{x}([m+1]P)$

The **loop invariant** is $(x_0, x_1) = (\mathbf{x}([\lfloor m/2^i \rfloor]P), \mathbf{x}([\lfloor m/2^i \rfloor + 1]P))$.

Side-channel concerns

Cryptographic algorithms must anticipate basic side-channel attacks (especially timing attacks and power analysis).

Diffie–Hellman implementations must be **uniform** and **constant-time** with respect to the secret scalars:

- No branching on bits of secrets
  eg. No `if(m == 0): ...` with m_i secret
- No memory accesses indexed by (bits of) secrets
  (eg. No `x = T[m]` where m is secret)

What we want is to have *exactly the same sequence of computer instructions* for every possible secret input.

Towards a uniform/constant-time Montgomery ladder

Algorithm 4: The Montgomery ladder for X25519

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $x = \mathsf{x}(P)$ with P in $\mathcal{E}(\mathbb{F}_p)$
Output: $\mathsf{x}([m]P)$

$\mathsf{u} \leftarrow (x, 1)$
$(\mathsf{x}_0, \mathsf{x}_1) \leftarrow ((1, 0), \mathsf{u})$
for i *in* $(\beta - 1, \ldots, 0)$ **do**
    if $m_i = 0$ **then**
        $(\mathsf{x}_0, \mathsf{x}_1) \leftarrow (\mathsf{xDBL}(\mathsf{x}_0), \mathsf{xADD}(\mathsf{x}_0, \mathsf{x}_1, \mathsf{u}))$
    else
        $(\mathsf{x}_0, \mathsf{x}_1) \leftarrow (\mathsf{xADD}(\mathsf{x}_0, \mathsf{x}_1, \mathsf{u}), \mathsf{xDBL}(\mathsf{x}_1))$
return x_0

We must ensure **xDBL** & **xADD** are uniform, and convert the **if** to a constant-time **conditional swap** (see appendix).

Towards a uniform/constant-time Montgomery ladder

Algorithm 5: The Montgomery ladder for X25519

Input: $m = \sum_{i=0}^{\beta-1} m_i 2^i$ and $x = \mathsf{x}(P)$ with P in $\mathcal{E}(\mathbb{F}_p)$

Output: $\mathsf{x}([m]P)$

$\mathsf{u} \leftarrow (x, 1)$

$(\mathsf{x}_0, \mathsf{x}_1) \leftarrow ((1, 0), \mathsf{u})$

for i *in* $(\beta - 1, \ldots, 0)$ **do**

  $(\mathsf{x}_0, \mathsf{x}_1) \leftarrow \mathsf{SWAP}(m_i, (\mathsf{x}_0, \mathsf{x}_1))$

  $(\mathsf{x}_0, \mathsf{x}_1) \leftarrow (\mathsf{xDBL}(\mathsf{x}_0), \mathsf{xADD}(\mathsf{x}_0, \mathsf{x}_1, \mathsf{u}))$

  $(\mathsf{x}_0, \mathsf{x}_1) \leftarrow \mathsf{SWAP}(m_i, (\mathsf{x}_0, \mathsf{x}_1))$

return x_0

$\mathsf{SWAP}(b, (v_0, v_1))$ returns (v_b, b_{1-b}) (see appendix).

Easy exercise: reduce the number of **SWAP**s from 2β to $\beta + 1$.

X25519

x-only Diffie–Hellman is a cute mathematical/algorithmic trick.

It's also the way we do Diffie–Hellman ***in the real world*** *today.*

X25519 is a Diffie–Hellman key-exchange algorithm in TLS 1.3, OpenSSH, Signal/Whatsapp, and other applications...

- Based on Bernstein's **Curve25519** software (2006)
- Formalized in **RFC7748**, *Elliptic curves for security* (2016)

A massive upgrade on traditional ECDH (used e.g. in TLS ≤ 1.2), which was based on NIST's standard prime-order curves.

More detail: see the appendix.

From groups to group actions

The quantum menace

Shor's quantum algorithm solves DLP in **polynomial time**.

Attacking real-world DH instances with Shor requires **large, general-purpose quantum computers**.

Q: Will sufficiently large quantum computers ever be built?

*Say **yes** if you want to get funded.*

Global research effort: replacing classic group-based public-key cryptosystems with **postquantum** alternatives.

Key exchange from group actions

Funnily enough, the closest thing we have to postquantum DH is based on a group *and* elliptic curve**s**!

Classic DH: $\mathbb{Z}/N\mathbb{Z}$ acts on a group $\mathcal{G} \subset \mathcal{E}(\mathbb{F}_p)$.

$$A = [a]P \qquad B = [b]P \qquad S = [a]B = [b]A = [ab]P$$

Modern DH: $\mathbb{Z}/N\mathbb{Z}$ acts on a quotient set

$$A = \pm[a]P \qquad B = \pm[b]P \qquad S = \pm[a]B = \pm[b]A = \pm[ab]P$$

Group-action DH: a (multiplicative) group $\mathfrak{G}$ acts on a set

$$A = \mathfrak{a} \cdot P \qquad B = \mathfrak{b} \cdot P \qquad S = \mathfrak{a} \cdot B = \mathfrak{b} \cdot A = \mathfrak{ab} \cdot P$$

Group-action Diffie–Hellman

Group-action DH: a (multiplicative) group $\mathfrak{G}$ acts on a set $\mathcal{S}$.

$$A = \mathfrak{a} \cdot P \qquad B = \mathfrak{b} \cdot P \qquad S = \mathfrak{a} \cdot B = \mathfrak{b} \cdot A = \mathfrak{ab} \cdot P$$

This is a logical continuation of modern Diffie–Hellman:

- **composition** in DH is all in the scalars, so we replace the ring $\mathbb{Z}/N\mathbb{Z}$ with a group $\mathfrak{G}$ (a simpler algebraic structure with composition)
- **vulnerability** to Shor's algorithm comes from the group structure on the public keys, so we remove this entirely and work with an unstructured set $\mathcal{S}$ instead.

Problem: finding $(\mathfrak{G}, \mathcal{S})$ such that the action $(\mathfrak{a}, P) \mapsto \mathfrak{a} \cdot P$ is efficient and the DLP and CDHP analogues are hard.

CSIDH: candidate postquantum group action

CSIDH (Castryck, Lange, Martindale, Panny, Renes 2018): a candidate postquantum group action for key exchange.

Based on ideas and techniques from Couveignes, Rostovtsev–Stolbunov, and De Feo–Kieffer–Smith.

Based on **CM theory** for a quadratic imaginary field K:

Group: $\mathfrak{G} = \mathrm{Cl}(O_K)$, the group of ideal classes of the maximal order of K

Space: $\mathcal{S} = \{\mathcal{E}/\mathbb{F}_q \mid \mathrm{End}(\mathcal{E}) \cong O_K\}/(\mathbb{F}_q\text{-isomorphism})$

Action: Ideals $\mathfrak{a}$ in O_K correspond to **isogenies** $\phi_\mathfrak{a} : \mathcal{E} \to \mathcal{E}/\mathcal{E}[\mathfrak{a}] =: \mathfrak{a} \cdot \mathcal{E}$. *This action extends to fractional ideals and factors through* $\mathrm{Cl}(O_K)$.

Details: see Joost Renes' talk, or ask us any time this week!

Appendices

Conditional swaps

Conditional swap

Remove **if**s using classic constant-time **conditional swaps**.

This can be done in several ways.

Here's a conditional swap for a pair of binary values, viewed as integers, using only arithmetic operations:

Algorithm 6: Conditional swap using arithmetic operations

1 **Function** *SWAP*
  Input: $b \in \{0, 1\}$ and $(\mathbf{x}_0, \mathbf{x}_1)$
  Output: $(\mathbf{x}_0, \mathbf{x}_1)$ if $b = 0$, $(\mathbf{x}_1, \mathbf{x}_0)$ if $b = 1$
2 **return** $((1 - b)\mathbf{x}_0 + b\mathbf{x}_1\,, b\mathbf{x}_0 + (1 - b)\mathbf{x}_1)$

Projective coordinates

Projective coordinates

In practice, we almost always use **projective coordinates** for $\mathcal{E}$, putting $x = X/Z$ and $y = Y/Z$. The curve equation becomes

$$\mathcal{E} : Y^2Z = X^3 + aXZ^2 + bZ^3 \,.$$

The points become

$$\mathcal{E}(\mathbb{F}_p) = \left\{(\alpha : \beta : \gamma) : \alpha, \beta, \gamma \in \mathbb{F}_p, \beta^2\gamma = \alpha^3 + a\alpha\gamma^2 + b\gamma^3\right\}$$

modulo **projective equivalence**, which is

$$(X : Y : Z) = (\lambda X : \lambda Y : \lambda Z) \quad \text{for all} \quad \lambda \in \mathbb{F}_p^\times \,.$$

We **exclude** $(0 : 0 : 0)$, which is not a projective point.

The **point at infinity** $\mathcal{O}_\mathcal{E}$ is $(0 : 1 : 0)$ in projective coordinates. It is **the unique** point where $Z = 0$.

Compressing points

We use projective points $(X : Y : Z)$ throughout our algorithms, but these require $3 \log_2 p$ bits each.

To store and transmit points as **public keys**, we **compress** them to $\log_2 p + 1$ **bits** as follows:

1. **Normalize** $(X : Y : Z)$ to $(x : y : 1) = (X/Z : Y/Z : 1)$.
2. Compute[6] the **"sign"** σ of y in $\mathbb{F}_p$.
3. Store (x, σ).

To **recover** y from (x, σ), compute the square root of $x^3 + ax + b$ with sign σ.

[6] There is no canonical definition, but you could use e.g. $\text{sign}(y) = \text{LSB}(y)$.

Coordinate sytems

Why use projective coordinates?

Mathematically, projective coordinates give a unified form for all points on the curve: $\mathcal{O}_{\mathcal{E}} = (0 : 1 : 0)$ is a point like any other, not a special symbol.

Algorithmically, projective coordinates let us **avoid expensive divisions** in $\mathbb{F}_p$. The Z-coordinate "accumulates denominators".

In practice, we use not only projective coordinates, but also **alternative models** for the curve equation and group law to **gain efficiency** and facilitate **implementation safety**.

We will see an example of this when we cover **modern ECDH**, which uses **Montgomery curve arithmetic**.

Montgomery arithmetic and X25519

Montgomery models for elliptic curves

In the following, we fix a **Montgomery curve**[7]

$$\mathcal{E} : BY^2Z = X(X^2 + AXZ + Z^2)$$

with $A \neq \pm 2$ and $B \neq 0$ in $\mathbb{F}_p$.

Notation: given points P and Q in $\mathcal{E}(\mathbb{F}_p)$, we write

$$P = (X_P : Y_P : Z_P)\,, \qquad P \oplus Q = (X_\oplus : Y_\oplus : Z_\oplus)\,,$$
$$Q = (X_Q : Y_Q : Z_Q)\,, \qquad P \ominus Q = (X_\ominus : Y_\ominus : Z_\ominus)\,.$$

[7] ***Observe:*** *we can convert to and from a short Weierstrass model for $\mathcal{E}$ via $(X : Y : Z) \mapsto (X - AZ/3 : Y : Z)$, so all the elliptic curve theory we have already described transfers to this curve.*

xADD

Pseudo-addition on $\mathcal{E} : BY^2Z = X(X^2 + AXZ + Z^2)$:

$$\text{xADD} : (\mathbf{x}(P), \mathbf{x}(Q), \mathbf{x}(P \ominus Q)) \longmapsto \mathbf{x}(P \oplus Q)$$

We use

$$(X_\oplus : Z_\oplus) = \left(Z_\ominus \cdot [U + V]^2 : X_\ominus \cdot [U - V]^2\right)$$

where

$$\begin{cases} U = (X_P - Z_P)(X_Q + Z_Q) \\ V = (X_P + Z_P)(X_Q - Z_Q) \end{cases}$$

xDBL

Pseudo-doubling on $\mathcal{E} : BY^2Z = X(X^2 + AXZ + Z^2)$:

$$\mathsf{xDBL} : \mathbf{x}(P) \longmapsto \mathbf{x}([2]P)$$

We use

$$(X_{[2]P} : Z_{[2]P}) = \left(Q \cdot R : S \cdot \left(R + \tfrac{A+2}{4} S\right)\right)$$

where

$$\begin{cases} Q = (X_P + Z_P)^2 \,, \\ R = (X_P - Z_P)^2 \,, \\ S = 4X_P \cdot Z_P = Q - R \,. \end{cases}$$

Curve25519

Bernstein (PKC 2006) defined the elliptic curve

$$\mathcal{E} : Y^2Z = X(X^2 + 486662 \cdot XZ + Z^2) \quad \text{over } \mathbb{F}_p$$

where $p = 2^{255} - 19$.

The curve has order $\#\mathcal{E}(\mathbb{F}_p) = 8r$, where r is prime.

If we let B be any nonsquare in $\mathbb{F}_p$, then the *quadratic twist*

$$\mathcal{E}' : B \cdot Y^2Z = X(X^2 + 486662 \cdot XZ + Z^2)$$

has order $\#\mathcal{E}'(\mathbb{F}_p) = 4r'$, where r' is prime.

The X25519 function

The **X25519 function** maps $\mathbb{Z}_{\geq 0} \times \mathbb{F}_p$ into $\mathbb{F}_p$, via

$$(m, u) \longmapsto u_m := x_m \cdot z_m^{(p-2)}$$

where $(x_m : * : z_m) = [m](u : * : 1) \in \mathcal{E}(\mathbb{F}_p) \cup \mathcal{E}'(\mathbb{F}_p)$.

Note: generally $z_m \neq 0$, in which case $(u_m : * : 1) = [m](u : * : 1)$ in $\mathcal{E}(\mathbb{F}_p)$ or $\mathcal{E}'(\mathbb{F}_p)$.

Exercise: for any given u, inverting $(m, u) \mapsto u_m$ amounts to solving a discrete logarithm in either $\mathcal{E}(\mathbb{F}_p)$ or $\mathcal{E}'(\mathbb{F}_p)$.

Diffie–Hellman with X25519

The global public "base point" is $u_1 = 9 \in \mathbb{F}_p$.

The point $(u_1 : * : 1) \in \mathcal{E}(\mathbb{F}_p)$ has 252-bit prime order r.

The "scalars" are integers in $S = \{2^{254} + 8i : 0 \leq i < 2^{251}\}$.

Alice samples a secret $a \in S$, computes $A := u_a = \text{X25519}(a, u_1)$, publishes A.

Bob samples a secret $b \in S$, computes $B := u_b = \text{X25519}(b, u_1)$, publishes B.

Alice computes the shared secret u_{ab} as $\text{X25519}(a, B)$

Bob computes the shared secret u_{ab} as $\text{X25519}(b, A)$.

Discrete logarithms in generic groups

Discrete logarithms in generic groups

Shoup's notion of a probabilistic **generic algorithm**: operating on $\mathbb{Z}/N\mathbb{Z}$, elements encoded in a set of bitstrings $S \subset \{0,1\}^*$. Compute $\oplus$, etc., on elements of S using **oracles**.

Idea: generic algorithms work independently of the encoding $\sigma : \mathbb{Z}/N\mathbb{Z} \to S$, so cannot use any information about the representation of elements of $\mathbb{Z}/N\mathbb{Z}$ (or, more generally, any $\mathcal{G}$).

Theorem (Shoup)
If $\mathcal{A}$ is a generic algorithm making at most m oracle queries, and $x \in \mathbb{Z}/N\mathbb{Z}$ and the encoding σ is chosen at random, then the probability that $\mathcal{A}$ computes x from $\sigma(1)$ and $\sigma(x)$ is $O(m^2/p)$ (with the probability taken over the choice of x and the coin flips of $\mathcal{A}$), where p is the largest prime divisor of N.

Square-root DLP algorithms

Algorithm 7: Baby-step giant-step algorithm

1 **Function** *BSGS*

Input: P and Q in $\mathcal{G} = \langle P \rangle$ of order N

Output: x such that $Q = [x]P$

2 $B \leftarrow \lceil \sqrt{N} \rceil$; $R \leftarrow P$; Initialize a hash table $\mathcal{T}$

3 **for** i *in* $(1, \ldots, B)$ **do** invariant: $R = [i]P$

4 Hash R and store $\mathcal{T}[R] \leftarrow i$

5 $R \leftarrow R \oplus P$

6 **for** j *in* $(0, \ldots, B)$ **do** invariant: $S = [x - jB]P$

7 $S \leftarrow Q \ominus [j]R$

8 **if** $S \in \mathcal{T}$ **then** $S = [i]P$, so $x = jB + i$

9 **return** $(j \cdot B + \mathcal{T}[S])$

Pohlig–Hellman I: Discrete logs in prime-power groups

Algorithm 8: Discrete logarithm in a prime-power group.

1 **Function** *DiscreteLogPrimePower*

  Input: P and Q in $\mathcal{G} = \langle P \rangle$ where $\#\mathcal{G} = p^e$ for some p, e

  Output: x such that $Q = [x]P$

2 $y \leftarrow 0$

3 $S \leftarrow [p^{e-1}]P$ `// in order-`p `subgroup`

4 **for** *i in (0,...,e-1)* **do** invariant: $y = x \bmod p^i$

5 $T \leftarrow [p^{e-1-i}](Q \ominus [y]P)$ `// in order-`p `subgroup`

6 $d \leftarrow \mathrm{BSGS}(T, S)$ `// or use Pollard` ρ

7 $y \leftarrow y + p^i \cdot d$

8 **return** y

For DiscreteLogPrime we can use (e.g.) BSGS, in time $O(\sqrt{p})$.

Pohlig–Hellman II: Reduction to prime power order

Algorithm 9: Discrete logarithm in a group where the prime factorization of the order is known.

1 **Function** *DiscreteLogComposite*

  Input: P and Q in $\mathcal{G} = \langle P \rangle$ where $\#\mathcal{G} = N = \prod_{i=1}^{n} p_i^{e_i}$

  Output: x such that $Q = [x]P$

2 **for** i ***in*** $(1, \ldots, n)$ **do**

3 $P_i \leftarrow [N/p_i^{e_i}]P$ `// in order-` $p_i^{e_i}$ `subgroup`

4 $Q_i \leftarrow [N/p_i^{e_i}]Q$ `// in order-` $p_i^{e_i}$ `subgroup`

5 $x_i \leftarrow$ DiscreteLogPrimePower(Q_i, P_i).

6 **return** *ChineseRemainderTheorem*$((x_1, p_1^{e_1}), \ldots, (x_n, p_n^{e_n}))$

DLP with a CDHP oracle: the Maurer reduction

Conditional CDHP $\Longrightarrow$ DLP: the Maurer reduction

We want to **solve** DLP instances in a group $\mathcal{G}$ of prime order p, **given a DH oracle** for $\mathcal{G}$.

First step: find/precompute an $\mathcal{E}/\mathbb{F}_p : Y^2 = X^3 + aX + b$ such that $\mathcal{E}(\mathbb{F}_p)$ is cyclic and has **polynomially smooth** order.

The idea: $\mathcal{E}(\mathbb{F}_p)$ must have a polynomial-time DLP algorithm using only basic group operations (e.g. Pohlig–Hellman).

Caveat: **constructing such an $\mathcal{E}$** *in polynomial time* is hard!

- Maurer supposes this is feasible (conditional reduction)
- For cryptographic p we are often lucky
- In general, this seems an impossibly strong hypothesis

The Maurer reduction CDHP $\implies$ DLP continued

We want to solve a DLP instance $Q = [x]P$ in $\mathcal{G}$.

Given: a smooth-order auxiliary curve $\mathcal{E}/\mathbb{F}_p : Y^2 = X^3 + aX + b$ and a generator (x_0, y_0) for $\mathcal{E}(\mathbb{F}_p)$.

The CDHP oracle lets us compute $[F(x)]P$ for all polynomials F.

1. Use Tonelli–Shanks to compute[8] an $R = [y]P$ such that $[y^2]P = [x^3 + ax + b]P$.
   Now $(Q, R) = ([x]P, [y]P) \in \mathcal{E}(\mathcal{G})$; we still don't know x or y.
2. Compute $Q_0 = [x_0]P$ and $R_0 = [y_0]P$)
3. Solve the DLP instance $(Q, R) = [e](Q_0, R_0)$ in $\mathcal{E}(\mathcal{G})$ for e.
4. Compute $(x, y) = [e](x_0, y_0)$ in $\mathcal{E}(\mathbb{F}_p)$ and return x.

[8] If this fails (i.e. $x^3 + ax + b$ is not square in $\mathbb{F}_p$): replace $Q = [x]P$ with $Q + [\delta]P = [x + \delta]P$ for some δ and try again...