Modern Elliptic Curve Cryptography

Benjamin Smith

SAC 2020 Summer School // October 19, 2020

Département d'Informatique, École polytechnique

The need for key agreement

Key agreement is a fundamental operation in cryptography.

It allows two principals to establish a shared secret key without prior contact.

The classic protocol for achieving 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.

For more generality and flexibility, we use **Key Encapsulation Mechanisms** (**KEM**s); though sometimes we really do need Diffie–Hellman as a component in more complicated cryptosystems.

Keypairs

Keys come in matching (Public, Private) pairs.

Keypairs may be long-term (**static**) or single-use (**ephemeral**).

Every **public key** poses an individual mathematical **problem**;
the matching **private key** gives the **solution**.

In our discrete-log setting, keypairs present **DLP instances**
in a prime-order group $\mathcal{G} = \langle P \rangle$:

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

Diffie–Hellman key exchange

Goal: Alice and Bob want to establish a **shared secret** with no prior contact.

In Diffie–Hellman key exchange, we **combine secret scalars** from both sides using **composition** of scalar multiplications, which becomes **multiplication** of scalars.

Warning: Diffie–Hellman has no built-in authentication!

Alice		Bob
$(Q_A, x_A) = KeyPair(\,)$		$(Q_B, x_B) = KeyPair(\,)$
--------------------------> Q_A		
		Q_B <------------------------
$S = [x_A]Q_B$		$S = [x_B]Q_A$

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.

The Diffie–Hellman problem

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

Given $(P, Q_A = [x_A]P, Q_B = [x_B]P)$, *compute* $S = [x_A x_B]P$.

Relating the DLP and CDHP:

- Clearly the CDHP reduces immediately to the DLP:
  you can solve a CDHP by solving a single DLP.
- There is a **conditional polynomial-time reduction** from the DLP to the CDHP (not obvious! Maurer–Wolf, ...)
- For the $\mathcal{G}$ we use in practice, there is an **unconditional subexponential time reduction** from the DLP to the CDHP (Muzerau–Smart–Vercauteren).

Modern Diffie–Hellman key exchange

Alice		**Bob**
$(Q_A, x_A) = KeyPair(\,)$		$(Q_B, x_B) = KeyPair(\,)$
-------------------------------> Q_A		
		Q_B <-------------------------------
$S = [x_A]Q_B$		$S = [x_B]Q_A$

DH never directly uses the group structure on $\mathcal{G}$. **All we need is a** ***set*** $\mathcal{G}$ and big randomly samplable sets A, B of efficiently computable functions $\mathcal{G} \to \mathcal{G}$ s.t.

- $[a][b] = [b][a]$ 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 $\mathrm{x}(P) = x_P$.
- Projectively: $\mathrm{x}((X : Y : Z)) = (X : Z) \in \mathbb{P}^1$ when $Z \neq 0$, and $\mathrm{x}(\mathcal{O})) = \mathrm{x}((0 : 1 : 0)) = (1 : 0)$.

Advantage: save time and space by ignoring y.

Diffie–Hellman modulo signs

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

The protocol is now

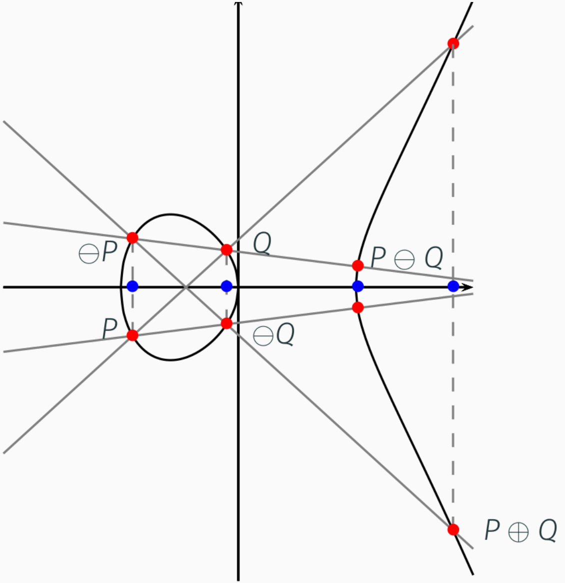
$$\begin{aligned} \textbf{Alice} \;\; & \text{computes } (a, \mathbf{x}(P)) \mapsto \mathbf{x}([a]P); \\ \textbf{Bob} \;\; & \text{computes } (b, \mathbf{x}(P)) \mapsto \mathbf{x}([b]P); \\ \textbf{Alice} \;\; & \text{computes } (a, \mathbf{x}([b]P)) \mapsto \mathbf{x}([a][b]P); \\ \textbf{Bob} \;\;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

Question: *is this well-defined?*

Question: *How can we compute pseudo-scalar multiplication* $(m, \mathbf{x}(P)) \mapsto \mathbf{x}([m]P)$ *efficiently, without using* $\oplus$*?*

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

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



Pseudo-group operations

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

Pseudo-addition:

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

Pseudo-doubling:

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

Bonus: it is easier to identify, isolate, and avoid special cases for $\mathtt{xADD}$ than for $\oplus$.

Notation

In the following, we fix a **Montgomery curve**

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

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

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

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

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

xADD

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

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

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

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

We use

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

where

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

Differential addition chains

We evaluate $\mathbf{x}(P) \mapsto \mathbf{x}([m]P)$ by combining `xADD`s and `xDBL`s using **differential addition chains**.

This means that **every** $\oplus$ has summands with **known difference**.

Classic example: the **Montgomery ladder**.

The Montgomery ladder in a group

Algorithm 1: The Montgomery ladder in a group

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

Output: $[m]P$

```
(R_0, R_1) ← (0, P)                              // Invariant: R_1 = R_0 ⊕ P
for i in (β − 1, . . . , 0) do                   // invariant: R_0 = [⌊m/2^i⌋]P
    if m_i = 0 then
        (R_0, R_1) ← ([2]R_0, R_0 ⊕ R_1)
    else
        (R_0, R_1) ← (R_0 ⊕ R_1, [2]R_1)
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!)

$\Longrightarrow$ easy adaptation from $\mathcal{E}$ to $\mathbb{P}^1$.

The Montgomery ladder with pseudo-operations

Algorithm 2: 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: $\mathbf{x}([m]P)$

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

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

  if $m_i = 0$ **then**

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

  else

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

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

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

X25519

X25519 is the state-of-the-art Diffie–Hellman key-exchange algorithm, standardized for TLS 1.3 (and other applications).

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

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

Curve25519

Bernstein (PKC 2006) defined an elliptic curve

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

where $p = 2^{255} - 19$. This curve is known as `Curve25519`.

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

Curve25519

Bernstein (PKC 2006) defined an elliptic curve

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

where $p = 2^{255} - 19$. This curve is known as `Curve25519`.

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**

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

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: show that 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 $(u_1 : * : 1)$ has order r in $\mathcal{E}(\mathbb{F}_p)$ (remember: r is a 252-bit prime).

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

Key generation:

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

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

Shared secret:

Alice computes $S = u_{ab} = \mathrm{X25519}(a, B)$

Bob computes $S = u_{ba} = \mathrm{X25519}(b, A)$

Side-channel concerns

Our 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.

Algorithm 3: The Montgomery ladder for X25519

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

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

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

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

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

    if $m_i = 0$ **then** `// Remove branch using conditional swaps`

        $(\mathtt{x}_0, \mathtt{x}_1) \leftarrow (\mathtt{xDBL}(\mathtt{x}_0), \mathtt{xADD}(\mathtt{x}_0, \mathtt{x}_1, \mathtt{u}))$ `// Uniform xDBL, xADD`

    else

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

return $\mathtt{x}_0$

Conditional swap algorithms

Algorithm 4: Conditional swap: parallel bit operations

Function *SWAP*

Input: $b \in \{0,1\}$ and $(\mathsf{x}_0, \mathsf{x}_1)$

Output: $(\mathsf{x}_0, \mathsf{x}_1)$ if $b = 0$, $(\mathsf{x}_1, \mathsf{x}_0)$ if $b = 1$

$v \leftarrow b$ **and** $(\mathsf{x}_0$ **xor** $\mathsf{x}_1)$

return $(\mathsf{x}_0$ ***xor*** v, x_1 ***xor*** $v)$

Algorithm 5: Conditional swap: arithmetic operations

Function *SWAP*

Input: $b \in \{0,1\}$ and $(\mathsf{x}_0, \mathsf{x}_1)$

Output: $(\mathsf{x}_0, \mathsf{x}_1)$ if $b = 0$, $(\mathsf{x}_1, \mathsf{x}_0)$ if $b = 1$

return $((1-b)\mathsf{x}_0 + b\mathsf{x}_1, b\mathsf{x}_0 + (1-b)\mathsf{x}_1)$

Towards a uniform/constant-time Montgomery ladder

Algorithm 6: The Montgomery ladder for X25519

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

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

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

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

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

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

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

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

return $\mathtt{x}_0$

Clearly we can halve the number of conditional swaps (try doing this!).

Elliptic curve signature schemes

Keypairs

Keys come in matching (Public,Private) pairs.

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

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

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

where P is some fixed generator of $\mathcal{G}$.

Identity

Having an **identity** means being distinguishable, from everyone else.

Question: *what is identity, in a cryptographic sense?*

Identity

Having an **identity** means being distinguishable, from everyone else.

Question: what is identity, in a cryptographic sense?

Identity means **holding the private key** corresponding to a bound public key.

Identity and authentication

We want to cryptographically **authenticate communicating parties** in a protocol.

That is: we want to know that we are communicating with someone holding the secret x corresponding to some public $Q = [x]P$.

Recall that in **symmetric** crypto, MACs and AEAD can authenticate **data**, but **not communicating parties**. The reason for this is simple: in symmetric crypto, *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.

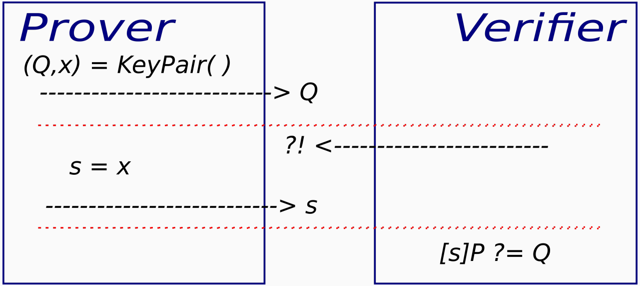
To formalize this, we introduce three characters:

Prover wants to *prove* their identity

Verifier wants to *verify* the identity of Prover

Simulator wants to impersonate Prover

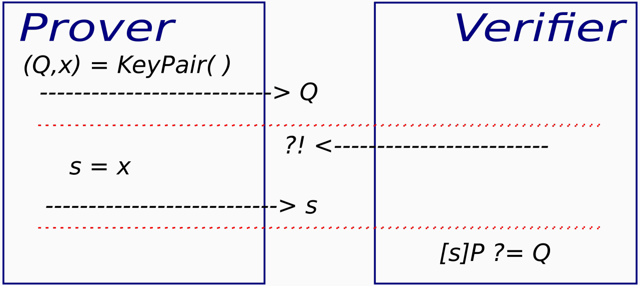
Ineffective identification



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

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

Ineffective identification

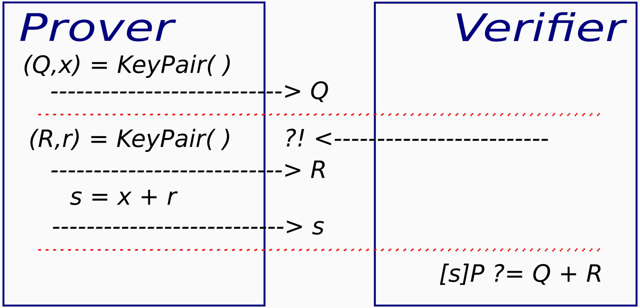


1. Verifier challenges;
2. Prover returns x;
3. Verifier accepts iff $[s]P = Q$.

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

Solution: hide long-term secrets with disposable one-shot (ephemeral) secrets.

Using ephemeral keys

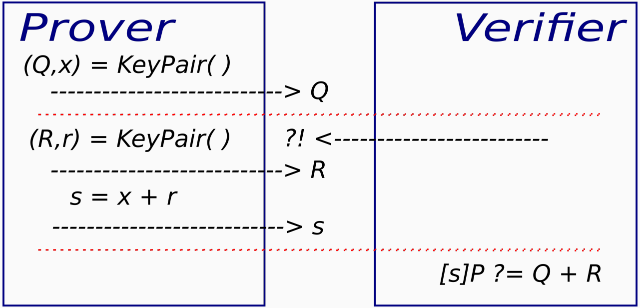


1. Prover generates an *ephemeral* keypair (R, r), commits to R;
2. Verifier challenges;
3. Prover sends R and $s = x + r$ to Verifier.

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

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

Using ephemeral keys

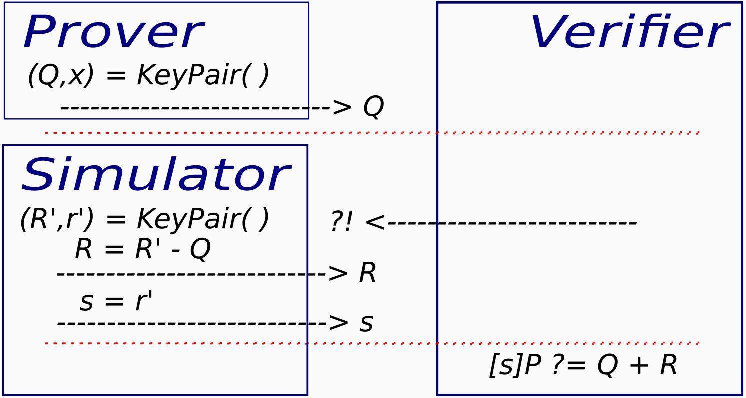


1. Prover generates an *ephemeral* keypair (R, r), commits to R;
2. Verifier challenges;
3. Prover sends R and $s = x + r$ to Verifier.

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

   Verifier accepts because $[s]P = [x]P + [r]P = Q + R$. **Problem: easy to cheat!**

Cheating: Simulator can easily impersonate the Prover



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

Note: *Simulator never knows x—nor the log of R, because then 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.

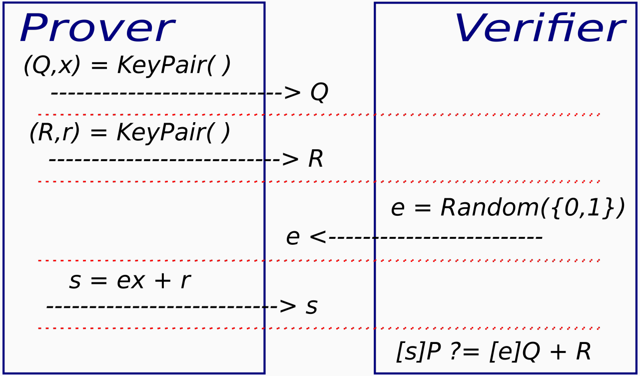
If she asks for the ephemeral secret $r = \log(R)$ while knowing s, then that would also reveal x.

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...

Prover | **Verifier**

$(Q,x) = \text{KeyPair}(\,)$

$\text{-------------------------->}\ Q$

$(R_1,r_1) = \text{KeyPair}(\,)$

$\text{-------------------------->}\ R_1$

$e_1 = \text{Random}(\{0,1\})$

$e_1\ \text{<------------------------}$

$s_1 = e_1 x + r_1$

$\text{-------------------------->}\ s_1$

$[s_1]P \mathrel{?=} [e_1]Q + R_1$

$(R_{128},r_{128}) = \text{KeyPair}(\,)$

$\text{-------------------------->}\ R_{128}$

$e_{128} = \text{Random}(\{0,1\})$

$e_{128}\ \text{<------------------------}$

$s_{128} = e_{128} x + r_{128}$

$\text{-------------------------->}\ s_{128}$

$[s_{128}]P \mathrel{?=} [e_{128}]Q + R_{128}$

Zero knowledge

The Chaum–Evertse–Graaf ID protocol is **zero knowledge**:

- Verifier is convinced that Prover holds the secret x, but
- Verifier learns absolutely nothing about the value of x.

This holds independent of the number of rounds carried out.

Schnorr ID (1991)

Running 128 rounds of Chaum–Evertse–Graaf ID is extremely inconvenient:

1. too many **interactive rounds** of communication
   (128 challenges and responses),
2. too much **bandwidth**
   ($128\times$ 256-bit group elements and $128 \times$ 256-bit scalars)
3. too much **computation** on each side
   ($128\times$ 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

Prover		**Verifier**
$(Q,x) = KeyPair(\,)$		
$\longrightarrow Q$		
$(R,r) = KeyPair(\,)$		
$\longrightarrow R$		
		$e = Random([0,2^{128}))$
	$e \longleftarrow$	
$s = ex + r$		
$\longrightarrow s$		
		$[s]P \mathrel{?=} [e]Q + R$

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

One round. Scalar mults: Prover = 1×256-bit, Verifier = $1 \times 256 + 1 \times 128$-bit.

Not zero knowledge

The Schnorr ID protocol is **not zero knowledge**.

Not zero knowledge

The Schnorr ID protocol is **not zero knowledge**.

- The triple (Q, s, e) is a particular solution of the equation $R = [s]P \ominus [e]Q$,
- so Verifier can choose e as a function of Q.

Still, it is (reasonably) clear that Verifier does not learn anything **useful** about x from the protocol.

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.

Step one: 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.

Prover

(Q,x) = KeyPair( )

-------------------------------> Q

(R,r) = KeyPair( )

-------------------------------> R

e = Hash(R)

s = ex + r

-------------------------------> s

Verifier

e = Hash(R)

[s]P ?= [e]Q + R

Step two: “identify to the data”

```
Prover
(Q,x) = KeyPair( )
      ----------------------------> Q
.....................................................................
(R,r) = KeyPair( )
   e = Hash(R)
      ----------------------------> e
    s = ex + r
      ----------------------------> s
.....................................................................
                                              Verifier
                                         R = [s]P - [e]Q
                                         e ?= Hash(R)
```

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

Schnorr signatures (1991)

```
Signer
 (Q,x) = KeyPair( )
    --------------------------> Q
 ............................................................................
 (R,r) = KeyPair( )                 M <------------ Message
     e = Hash(R,M)
    --------------------------> e
      s = ex + r
    --------------------------> s
 ............................................................................
                                                   Verifier
                                          R = [s]P - [e]Q
                                          e ?= Hash(R,M)
```

The hash function must provide 128 bits of **prefix-second-preimage** resistance.

Traditionally **no need for collision resistance**, though you might want it to protect against attacks on multiple keys.

Schnorr signatures

Schnorr signatures are proven secure in the **random oracle model**, though not in the **standard model**.

Schnorr **patented** his signature scheme; as a result, few people actually used it. *(Instead, we ended up with the inferior DSA and ECDSA protocols...)*

The patent **expired in 2008**.

This has led to the rise of contemporary Schnorr variants like **EdDSA**.

Bonus Track 1: EdDSA signatures

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

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

Key generation: 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)*.

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

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

Question: *what are the important differences between EdDSA and plain Schnorr?*

EdDSA Key Generation

Algorithm 7: EdDSA Key Generation

Output: $(Q, k) \in \mathcal{E}(\mathbb{F}_p) \times \{0,1\}^\beta$

$k \leftarrow \mathtt{Random}(\{0,1\}^\beta)$

$x \parallel y \leftarrow H(k)$ `// Both` x `and` y `are in` $\{0,1\}^\beta$

$Q \leftarrow [x]P$

return (Q, k)

EdDSA Signing

Algorithm 8: EdDSA signing

Input: Message $m \in \{0,1\}^*$, secret key k

Output: Signature $\sigma = (R, s) \in \mathcal{G} \times \mathbb{Z}/N\mathbb{Z}$

$x \parallel y \leftarrow H(k)$

$r \leftarrow H(y \parallel M)$

$R \leftarrow [r]P$

$s \leftarrow r + H(R \parallel Q \parallel M)x$

return $\sigma = (R, s)$

EdDSA Verification

Algorithm 9: EdDSA verification

Input: Message $m \in \{0,1\}^*$, signature $\sigma = (R, s) \in \mathcal{G} \times \mathbb{Z}/N\mathbb{Z}$, public key $Q = [x]P$

Output: **True** or **False**

$h \leftarrow H(R \parallel Q \parallel M)$

$R' = [s]P \oplus [-h]Q$ `// Multiexponentiation`

if $R' = R$ **then**

  return ***True***

else

  return ***False***

Bonus Track 2:
Multiscalar multiplication

Multiscalar multiplication

For k bits of security ($2k$-bit curve), intensive computations:

Signing 1x scalar multiplication by a $2k$-bit scalar

Schnorr Verif. 1x scalar mult. by a k-bit scalar,
plus 1x scalar mult. by a $2k$-bit scalar

EdDSA Verif. 2x scalar mults by 2k-bit scalars

The **cost of verification** is the chief **drawback** of EC signatures.

Multiexponentiation: a standard trick to **merge** "parallel" scalar multiplications into one, reducing verification cost...

Exercise: *compare the expected cost of key generation, signing, and verification for RSA and elliptic-curve signatures.*

Multiexponentiation with Straus' algorithm

Algorithm 10: Generic binary multiexponentiation

Function *Multiexponentiation*

  Input: $(a = \sum_{i=0}^{\beta-1} a_i 2^i, b = \sum_{i=0}^{\beta-1} b_i 2^i)$ in $[0..2^\beta)^2$, P, Q in $\mathcal{G}$

  Output: $[a]P \oplus [b]Q$

  $(T_{0,0}, T_{1,0}, T_{0,1}, T_{1,1}) \leftarrow (0_\mathcal{G}, P, Q, P \oplus Q)$

  $R \leftarrow 0_\mathcal{G}$

  for $i = \beta - 1$ ***down to*** 0 **do**

    $R \leftarrow [2]R \oplus T_{a_i, b_i}$

  return R

There exist **uniform** and also **differential** (x-only) variants.

#doubles same as plain exponentiation by $\max(|a|, |b|)$. *What about #adds?*

Bonus Track 3:
ECDSA

ECDSA

ECDSA is the standard Elliptic Curve Digital Signature Algorithm.

Targeting k bits of security:

Fix a $2k$-bit hash function H and a secure elliptic curve $\mathcal{E}/\mathbb{F}_p$ such that $\mathcal{E}(\mathbb{F}_p)$ contains a $\approx 2k$-bit prime-order subgroup $\mathcal{G} = \langle P \rangle \cong \mathbb{Z}/N\mathbb{Z}$.

Public-private keypairs are standard DLP instances $(Q = [x]P, x)$.

ECDSA Signing

Algorithm 11: ECDSA signing

Input: Message $m \in \{0,1\}^*$, secret key x

Output: Signature $\sigma = (s, t) \in [1, N-1]^2$

1 $z \leftarrow H(m)$

2 $(R = [r]P, r) \leftarrow \mathrm{KeyGen}()$ `// Ephemeral keypair`

3 $s \leftarrow x(R) \bmod N$ `// Viewing x-coord` $x(R) \in \mathbb{F}_p$ `as an integer!`

4 **if** $s = 0$ **then go to** Line 2

5 $t \leftarrow r^{-1}(z + sx) \pmod N$

6 **if** $t = 0$ **then go to** Line 2

7 **return** $\sigma = (s, t)$

ECDSA Verification

Algorithm 12: ECDSA signing

Input: Message $m \in \{0,1\}^*$, signature $\sigma = (s,t) \in [1, N-1]^2$, public key $Q = [x]P$

Output: **True** or **False**

$z \leftarrow H(m)$

$u \leftarrow zt^{-1} \pmod N$

$v \leftarrow st^{-1} \pmod N$

$S \leftarrow [u]P \oplus [v]Q$ `// Multiexponentiation`

if $s \equiv x(S) \pmod N$ **then** `// Viewing` $x(S)$ `as an integer!`

  return *True*

else

  return *False*

Disadvantages of ECDSA

ECDSA is **harder to implement correctly** than plain Schnorr or EdDSA.

It is also **harder to prove** things about ECDSA than it is for Schnorr or EdDSA, because it involves weird operations like changing moduli (from p to N, etc.)

For ECDSA, Schnorr, and EdDSA, the keys and **signatures are exceptionally small** (this is the USP of EC signatures).

Main drawback: **verification is relatively slow** (even with multiexponentiation).