Introduction to Elliptic Curve Cryptography

Benjamin Smith

Team **GRACE**

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

ECC School, Nijmegen, November 9 2017

Problems

We want to solve some important everyday problems in asymmetric crypto: signatures and key exchange.

...Also, a less common problem: encryption.

Today we will look at basic constructions associated with *one* hard problem: the discrete logarithm problem in a group $\mathcal{G}$.

Naturally, $\mathcal{G}$ will be a subgroup of an elliptic curve.

Where we're going

1. Waffle
2. Identification
3. Signatures
4. Key exchange
5. Encryption

Concrete groups

For security against generic algorithms,
$\#\mathcal{G}$ is a prime $\sim 2^{256}$
(more generally, $2^{2\beta}$ where β is the security level).

Candidate groups for 128-bit security:

1. *Historical:* $\mathcal{G} \subset \mathbb{G}_m(\mathbb{F}_p)$, the multiplicative group, 3072-bit p ($\implies$ elements of $\mathcal{G}$ encode to 3072 bits)
2. *Contemporary:* $\mathcal{G} \subseteq \mathcal{E}(\mathbb{F}_p)$, with $\mathcal{E}/\mathbb{F}_p$ an elliptic curve, 256-bit p ($\implies$ elements of $\mathcal{G}$ encode to $256 + \varepsilon$ bits)
3. *Experimental:* $\mathcal{G} \subseteq \mathcal{J}_{\mathcal{C}}(\mathbb{F}_p)$, with $\mathcal{C}/\mathbb{F}_p$ a genus-2 curve, 128-bit p ($\implies$ elements of $\mathcal{G}$ encode to $256 + \varepsilon$ bits)

Scalar multiplication

Write $\mathcal{G}$ additively: eg. $P + Q = R$

(later, use $\oplus$ instead of $+$ to distinguish from addition in $\mathbb{F}_p$).

Scalar multiplication (exponentiation):

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

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

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

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 $\mathcal{G}$:

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

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

Keypairs

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

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

1. The security of keys is algorithmic.
2. It can be *much* easier to attack sets of keys than to attack individual keys.
3. Cryptanalysis can and does begin at the moment that a given keypair is created and "bound to" (ie, when the public key is published), *not* when the keys are actually used!

Identity

Identity means… holding a private key
—nothing more, nothing less.

Ultimately, we want **authentication**:
to know that we are talking to the holder of the
secret x corresponding to some public $Q = [x]P$.

In symmetric crypto, MACs and AEAD can
authenticate *data*, but *not communicating parties*.

The reason is simple: in symmetric crypto,
both sides hold the same secret
—and a shared identity is no identity.

Identification

How do you prove your identity?

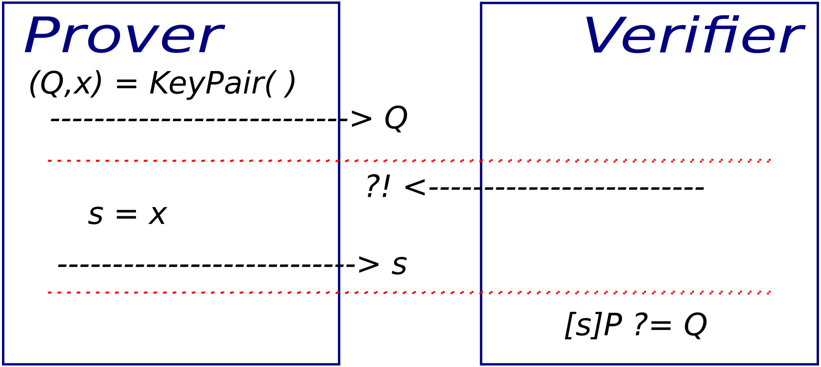
In our setting, you assert/claim an identity by publishing/binding/committing to a public key Q from a keypair $(Q = [x]P, x)$.

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

Identification



Prover

Verifier

(Q,x) = KeyPair( )

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

?! <------------------------

s = x

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

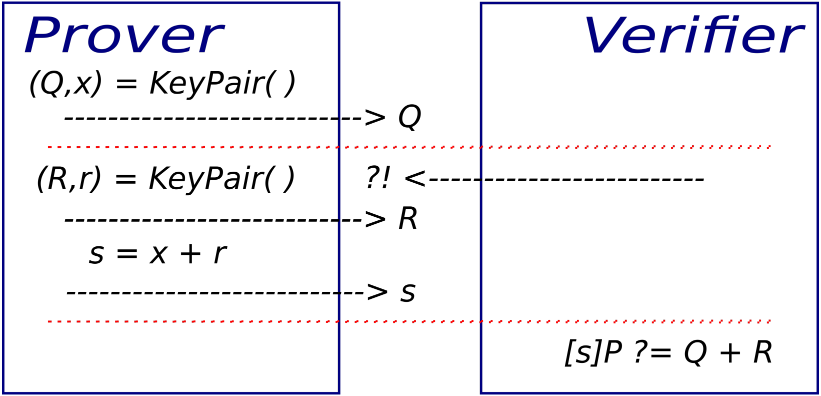
[s]P ?= Q

Verifier challenges; Prover returns x;
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.

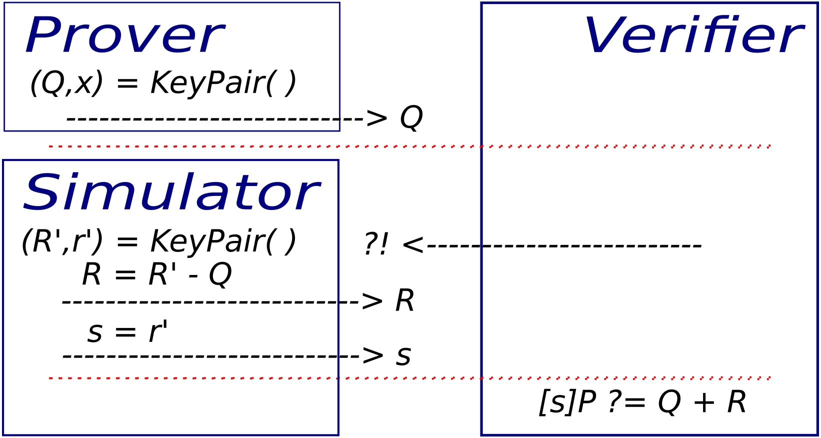


```
Prover                                              Verifier
 (Q,x) = KeyPair( )
   ---------------------------> Q
 (R,r) = KeyPair( )        ?! <------------------------
   ---------------------------> R
     s = x + r
   ---------------------------> s
                                          [s]P ?= Q + R
```

Prover generates an *ephemeral* keypair (R, r), commits R;
Prover sends R and $s = x + r$ to Verifier.
Note: s reveals nothing about x, because r is random
Verifier accepts because $[s]P = [x]P + [r]P = Q + R$.

Cheating

Problem: Simulator can easily impersonate Prover.



Prover

(Q,x) = KeyPair( )

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

Simulator

(R',r') = KeyPair( )

R = R' - Q

?! <-------------------------

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

s = r'

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

Verifier

[s]P ?= Q + R

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 distinguish between Prover and Simulator?

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

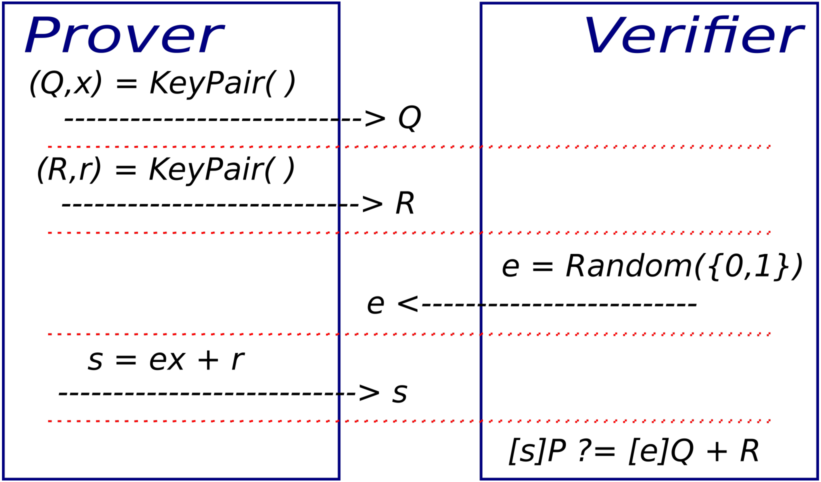
Simulator sends $s = \log(Q + R)$,
but knows *neither* $x = \log(Q)$ *nor* $r = \log(R)$.

Verifier can't ask for x.
If she asks for the ephemeral secret $r = \log(R)$ *as well as* s
then that would reveal x.

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

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

Chaum–Evertse–Graaf (1988)



Prover

Verifier

$(Q,x) = KeyPair(\,)$

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

$(R,r) = KeyPair(\,)$

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

$e = Random(\{0,1\})$

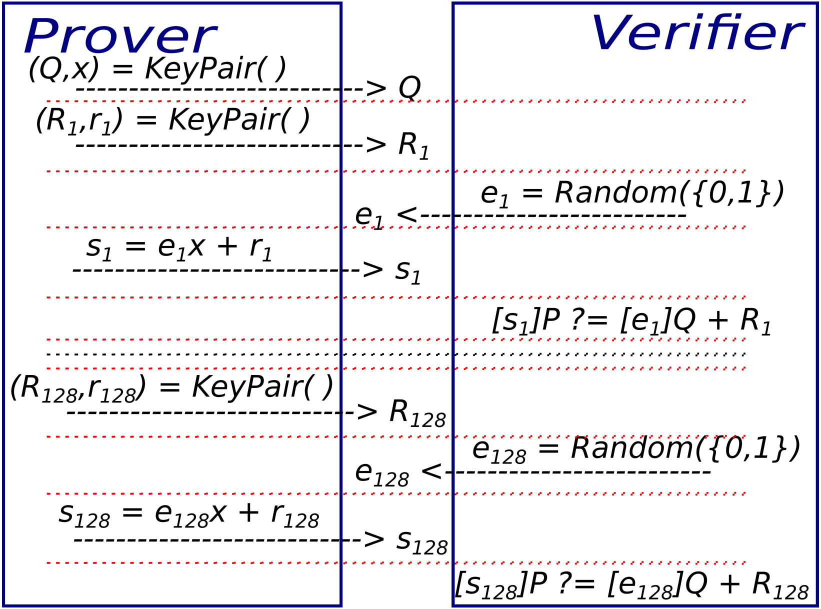
e <------------------------

$s = ex + r$

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

$[s]P \;?\!= [e]Q + R$

To cheat, Simulator must guess/anticipate *e*: 50% chance.
So repeat until Verifier is satisfied it's Prover (say 128 rounds).



Prover

Verifier

$(Q,x) = KeyPair(\,)$

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

$(R_1,r_1) = KeyPair(\,)$

----------------------------> R_1

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

e_1 <------------------------

$s_1 = e_1x + r_1$

----------------------------> s_1

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

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

----------------------------> R_{128}

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

e_{128} <------------------------

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

----------------------------> s_{128}

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

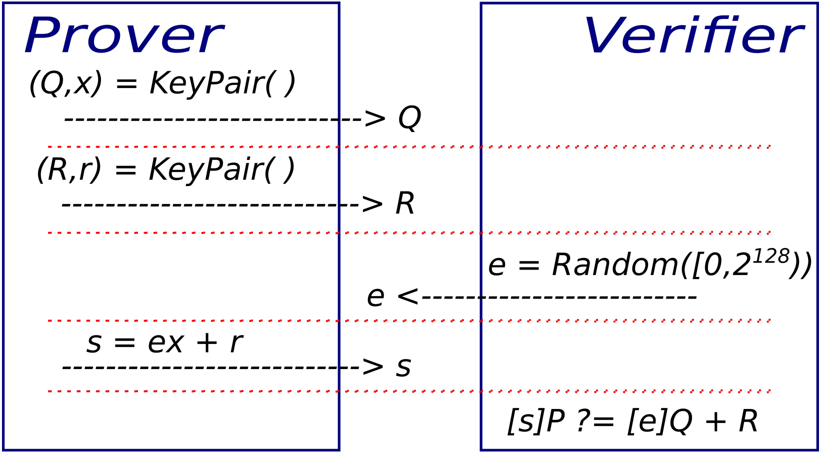
Schnorr ID (1991)

It's annoying to have to run 128 rounds of the Chaum–Evertse–Graaf ID protocol:

1. too much communication,
2. too much computation (128× 256-bit scalar multiplications for both Prover and Verifier!)

Schnorr (1991): we "parallelise" the 128 rounds, replacing 128 single bits with a single 128 bits.

Schnorr ID



Prover

Verifier

(Q,x) = KeyPair( )

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

(R,r) = KeyPair( )

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

$e = \mathrm{Random}([0,2^{128}))$

e <-------------------------

$s = ex + r$

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

$[s]P \mathrel{?=} [e]Q + R$

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

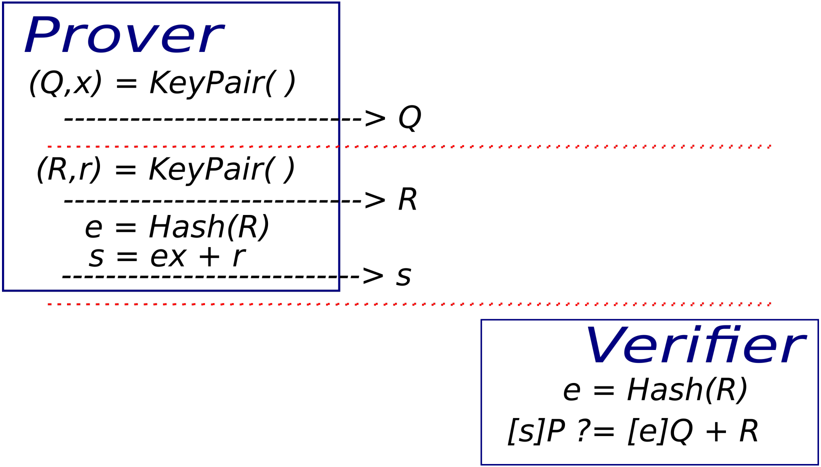
A signature is a sort of non-interactive proof that the Signer witnessed (created, saw) some data.

Authenticity, *message integrity*, *non-repudiability*: only the Signer could have created it, and only the Signer's public key is needed to *verify* it.

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

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

“Non-interactive Schnorr”



Prover

(Q,x) = KeyPair( )

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

(R,r) = KeyPair( )

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

e = Hash(R)

s = ex + r

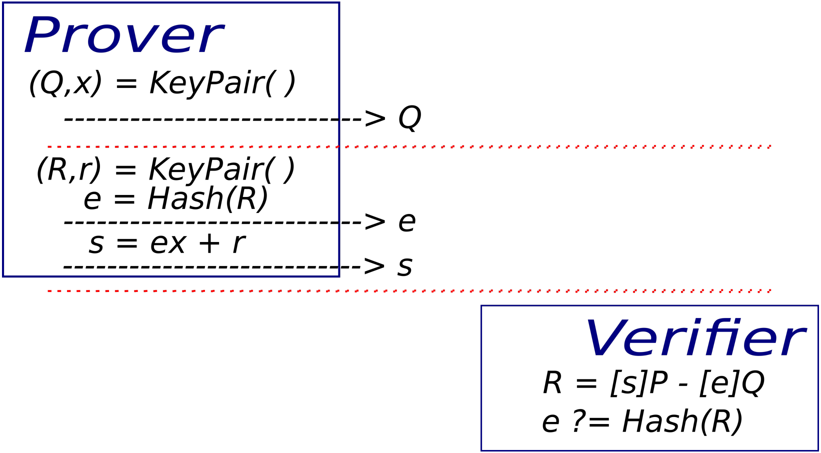
----------------------------> s

Verifier

e = Hash(R)

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

“Compact non-inter Schnorr”



Prover

(Q,x) = KeyPair( )

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

(R,r) = KeyPair( )

e = Hash(R)

---------------------------> e

s = ex + r

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

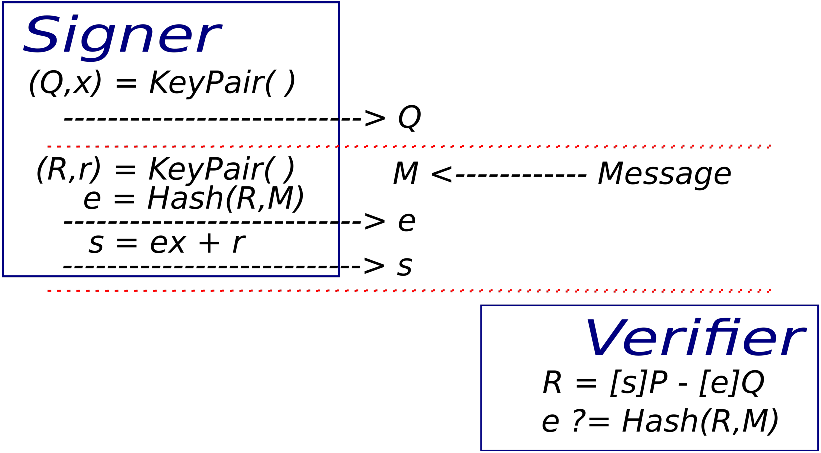
Verifier

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 it instead!

Schnorr signatures (1991)



Signer

(Q,x) = KeyPair( )

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

(R,r) = KeyPair( )

e = Hash(R,M)

M <------------ Message

--------------------------> e

s = ex + r

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

Verifier

R = [s]P - [e]Q

e ?= Hash(R,M)

Hash should 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).

Diffie–Hellman key exchange

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

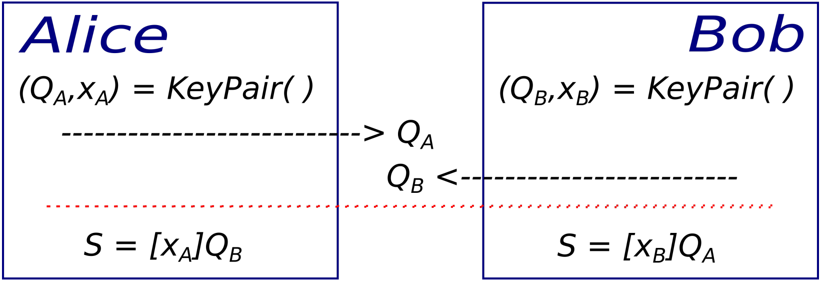
In Schnorr signatures, we *mask* secret scalars using
addition in $\mathcal{G}$, which becomes *addition* of scalars.

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

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

Alice and Bob want to establish a shared secret
with no prior contact (eg. for subsequent symmetric crypto).

They use the fact that $[a][b] = [b][a] = [ab]$ for all $a, b \in \mathbb{Z}$.



Alice

$(Q_A, x_A) = \mathit{KeyPair}(\,)$

$\xrightarrow{\hspace{4em}} Q_A$

Bob

$(Q_B, x_B) = \mathit{KeyPair}(\,)$

$Q_B \xleftarrow{\hspace{4em}}$

$S = [x_A]Q_B$

$S = [x_B]Q_A$

Alice & Bob now use a KDF (Key Derivation Function) to
compute a shared cryptographic key from the shared secret S.

Keypairs can be long-term ("static DH") or ephemeral.

Warning: no authentication! Trivial/universal MITM.

The Diffie–Hellman problem

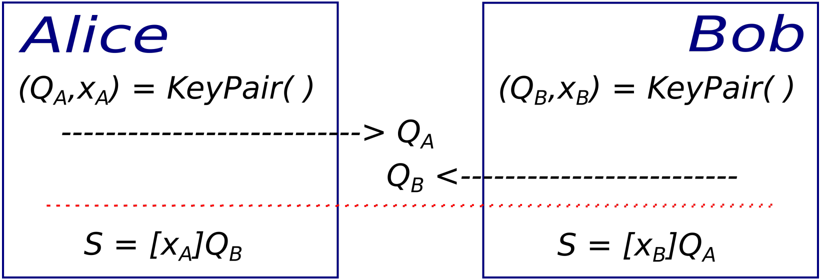
Diffie–Hellman security depends
not (directly) on the DLP, but rather on the
Computational Diffie–Hellman Problem:

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

If you can solve DLPs, then you can solve CDHPs.
The converse is not at all obvious,
but we have conditional results (Maurer–Wolf, ...)

For the $\mathcal{G}$ we use in practice, there is a
subexponential time equivalence with the DLP
(Muzerau–Smart–Vercauteren).

Modern Diffie–Hellman key exchange



Alice

$(Q_A, x_A) = KeyPair(\,)$

---------------------------> Q_A

Bob

$(Q_B, x_B) = KeyPair(\,)$

Q_B <------------------------

$S = [x_A]Q_B$

$S = [x_B]Q_A$

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 randomly sampleable and efficiently computable functions
$[a] : \mathcal{G} \to \mathcal{G}$, $[b] : \mathcal{G} \to \mathcal{G}$ such that $[a][b] = [b][a]$
such that the corresponding CDHP is believed hard.

Today we will see this in Curve25519, where $\mathcal{G} = \mathcal{E}/\pm 1$;
tomorrow you will see it in SIDH (Craig's lecture).

Modern Diffie–Hellman

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

Alice computes $(a, \pm P) \mapsto \pm[a]P$;
Bob computes $(b, \pm[a]P) \mapsto \pm[ab]P$...

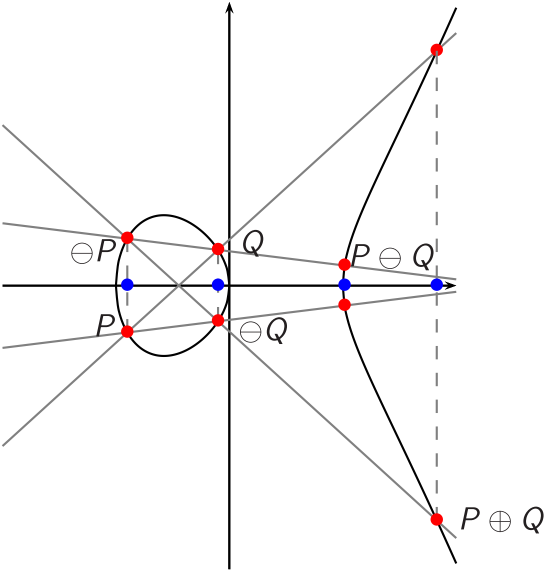
Elliptic curves: work on x-line $\mathbb{P}^1 = \mathcal{E}/\langle \pm 1 \rangle$.

Advantage: save time and space by ignoring y.

Problem: How do we compute $\pm[m]$ efficiently,
without using $\oplus$?

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

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



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

pseudo-addition

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

pseudo-doubling

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

Bonus: easier to identify, isolate, and avoid special cases for `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 $\mathbb{F}_p$.

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

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

xADD

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

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

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

We use

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

where

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

We evaluate $[m]$ by combining `xADD`s and `xDBL`s using differential addition chains

(ie. every $\oplus$ has summands with known difference).

Classic example: the Montgomery ladder.

Algorithm 1 The Montgomery ladder in a group

```
function Ladder($m = \sum_{i=0}^{\beta-1} m_i 2^i,\ P$)
    $(R_0, R_1) \leftarrow (0, P)$
    for $i := \beta - 1$ down to $0$ do
        if $m_i = 0$ then
            $(R_0, R_1) \leftarrow ([2]R_0, R_0 \oplus R_1)$
        else
            $(R_0, R_1) \leftarrow (R_0 \oplus R_1, [2]R_1)$
        end if
    end for ▷ invariant: $(R_0, R_1) = ([\lfloor m/2^i \rfloor]P, [\lfloor m/2^i \rfloor + 1]P)$
    return $R_0$                    ▷ $R_0 = [m]P,\ R_1 = [m+1]P$
end function
```

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 $\mathbb{P}^1$.

Algorithm 2 The Montgomery ladder on the x-line $\mathbb{P}^1$

```
function LADDER($m = \sum_{i=0}^{\beta-1} m_i 2^i$, $\mathbf{x}(P)$)
    $(x_0, x_1) \leftarrow (\mathbf{x}(0), \mathbf{x}(P))$
    for $i := \beta - 1$ down to 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
            $(x_0, x_1) \leftarrow (\mathtt{xADD}(x_0, x_1, \mathbf{x}(P)), \mathtt{xDBL}(x_1))$
        end if
    end for ▷ inv.: $(x_0, x_1) = (\mathbf{x}([\lfloor m/2^i \rfloor]P, \mathbf{x}([\lfloor m/2^i \rfloor + 1]P))$
    return $x_0$                              ▷ $x_0 = \mathbf{x}([m]P)$, $R_1 = \mathbf{x}([m+1]P)$
end function
```

X25519

X25519 is a Diffie–Hellman key-exchange algorithm
for TLS (and other applications),
based on Bernstein's *Curve25519* software (2006).

It is formalized in RFC7748,
Elliptic curves for security (2016).

It is an upgrade on the old ECDH in TLS,
which was based on NIST prime-order curves.

Curve25519

Bernstein (PKC 2006) defined the elliptic curve

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

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

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

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

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

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

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

Alice and Bob compute the shared secret u_{ab}
as $\mathrm{X25519}(a, B)$ and $\mathrm{X25519}(b, A)$, respectively.

Side-channel concerns

We 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.

We're using the Montgomery ladder, which is almost uniform:

Algorithm 3 The Montgomery ladder for X25519

```
function LADDER($m = \sum_{i=0}^{\beta-1} m_i 2^i$, $x$)
    $\mathbf{u} \leftarrow (x, 1)$
    $(\mathbf{x}_0, \mathbf{x}_1) \leftarrow ((1, 0), \mathbf{u})$
    for $i := \beta - 1$ down to 0 do
        if $m_i = 0$ then
            $(\mathbf{x}_0, \mathbf{x}_1) \leftarrow (\mathtt{xDBL}(\mathbf{x}_0), \mathtt{xADD}(\mathbf{x}_0, \mathbf{x}_1, \mathbf{u}))$
        else
            $(\mathbf{x}_0, \mathbf{x}_1) \leftarrow (\mathtt{xADD}(\mathbf{x}_0, \mathbf{x}_1, \mathbf{u}), \mathtt{xDBL}(\mathbf{x}_1))$
        end if
    end for
    return $\mathbf{x}_0$
end function
```

We need to ensure that `xDBL` and `xADD` are uniform,
and we need to remove the **if** statement.

Conditional swap

We can get rid of the if statement
using a classic constant-time *conditional swap*.

Algorithm 4 Conditional swap

```
function SWAP($b$,($\mathbf{x}_0, \mathbf{x}_1$))
    $v \leftarrow b$ and ($\mathbf{x}_0$ xor $\mathbf{x}_1$)
    return ($\mathbf{x}_0$ xor $v, \mathbf{x}_1$ xor $v$)
end function
```

Algorithm 5 Conditional swap

```
function SWAP($b$,($\mathbf{x}_0, \mathbf{x}_1$))
    return $((1-b)\mathbf{x}_0 + b\mathbf{x}_1 , b\mathbf{x}_0 + (1-b)\mathbf{x}_1)$
end function
```

Public-key encryption

Classic textbook problem, rarely appears in practice.

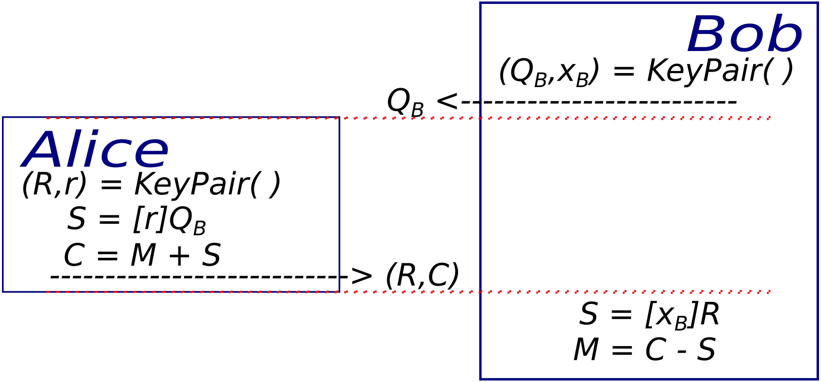
Alice wants to encrypt a message M for Bob.
Bob has a long-term keypair (Q_B, x_B).

Simple approach (ElGamal):

Alice views Q_B as Bob's half of a DH key exchange.
She can complete the Diffie–Hellman on her side,
use the shared secret to encrypt M,
and send her half of the DH with M.

To decrypt, Bob completes the DH on his side,
and uses the shared secret to decrypt.

Classic ElGamal encryption (1984)



Bob

$(Q_B, x_B) = KeyPair(\,)$

Q_B <-------------------------

Alice

$(R,r) = KeyPair(\,)$

$S = [r]Q_B$

$C = M + S$

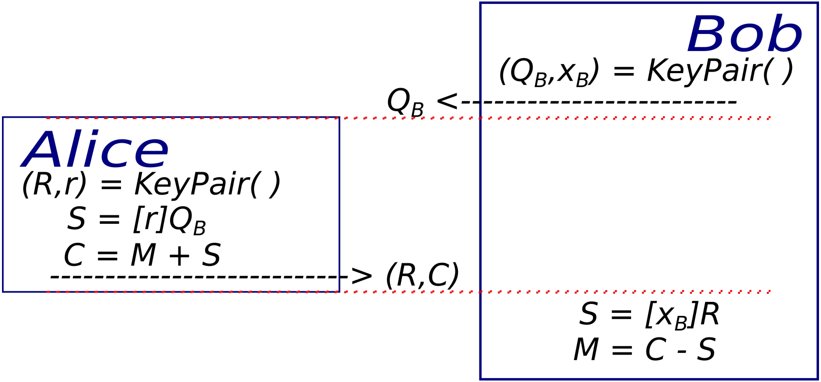
---------------------------> (R,C)

$S = [x_B]R$

$M = C - S$

Notice that this includes a half-static, half-ephemeral DH.
Alice's keypair *must* be ephemeral: never repeat r!
Otherwise, given ciphertexts (R, C_1) and (R, C_2),
you can compute $M_1 - M_2 = C_1 - C_2$.

Classic ElGamal is homomorphic



Bob

(Q_B, x_B) = KeyPair( )

Q_B <-------------------------

Alice

(R,r) = KeyPair( )

$S = [r]Q_B$

C = M + S

---------------------------> (R,C)

$S = [x_B]R$

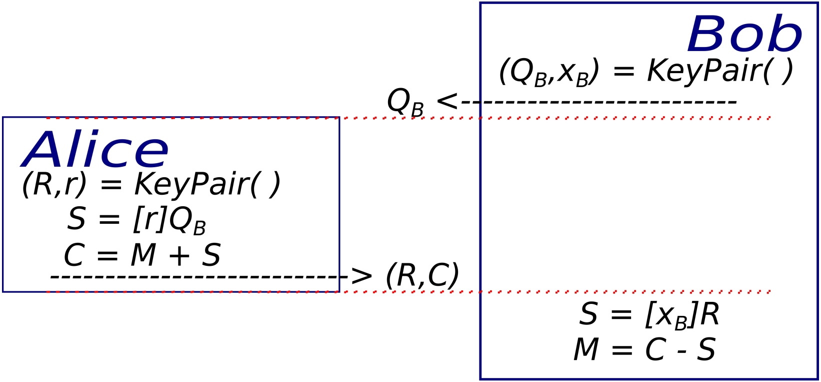
M = C - S

Problem: ElGamal is homomorphic!

Eg. $(R_1 + R_2, C_1 + C_2)$ is a legitimate encryption of $M_1 + M_2$.

This violates semantic security.

Towards modern ElGamal encryption



Bob

$(Q_B, x_B) = KeyPair(\ )$

Q_B <------------------------

Alice

$(R,r) = KeyPair(\ )$

$S = [r]Q_B$

$C = M + S$

--------------------------> (R,C)

$S = [x_B]R$

$M = C - S$

We have a deeper categorical/typing/casting problem:

Real messages are blobs of bits, **not elements of $\mathcal{G}$**.

Real ciphertexts should be random-looking bitstrings

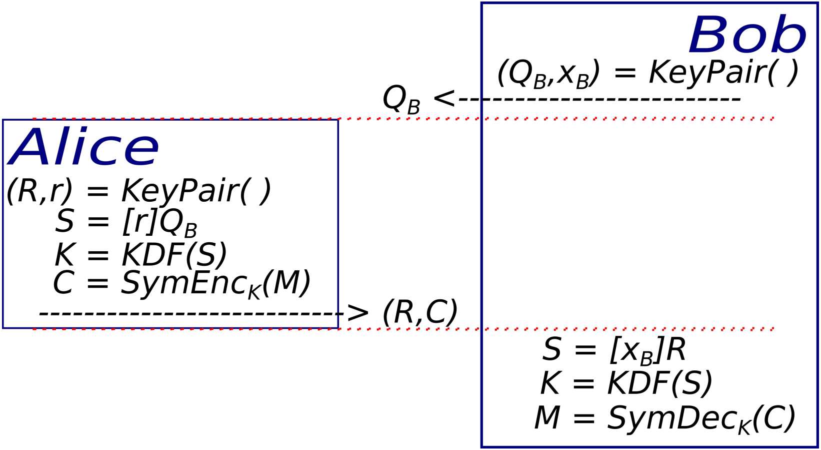
(or strange codomain elts), not elements of $\mathcal{G}$.

Don't do algebra in public

Discrete logarithms, groups, and algebraic structures are components of *cryptographic algorithms*, *not* the data these algorithms operate on.

If at any time your mathematics unconsciously bleeds through into your keys or data, *then you are doing something wrong*.

What you really want to do: DHIES



Bob

$(Q_B,x_B) = \mathit{KeyPair}(\ )$

Q_B <--------------------------

Alice

$(R,r) = \mathit{KeyPair}(\ )$

$S = [r]Q_B$

$K = \mathit{KDF}(S)$

$C = \mathit{SymEnc}_K(M)$

---------------------------> (R,C)

$S = [x_B]R$

$K = \mathit{KDF}(S)$

$M = \mathit{SymDec}_K(C)$

More details: Abdalla–Bellare–Rogaway ($\leq$ 2001)

Deliberate weirdness

If you're a research cryptographer, or
if you want to do something exotic like e-voting,
then you might *want* something homomorphic!

Problem I: encoding messages into $\mathcal{G}$.
Easy for $\mathbb{F}_p^\times$, *trickier for* $\mathcal{E}(\mathbb{F}_p)$.

Problem II: even once you have defined an encoding
of some messages into $\mathcal{G}$, you are stuck with an
intrinsically limited message space.