Asymmetric protocols from discrete logarithms

Benjamin Smith

Team **GRACE**

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

Summer school on real-world crypto and privacy

Sibenik, Croatia, June 5 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 $\mathcal{G}$.

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).

1. $\mathcal{G} \subset \mathbb{F}_p^\times$ (multiplicative group), with p a 3072-bit prime ($\implies$ elements of $\mathcal{G}$ encode to 3072 bits)
2. $\mathcal{G} \subseteq \mathcal{E}(\mathbb{F}_p)$, with $\mathcal{E}/\mathbb{F}_p$ an elliptic curve, p a 256-bit prime ($\implies$ elements of $\mathcal{G}$ encode to 256 bits)
3. $\mathcal{G} \subseteq \mathcal{J}_\mathcal{C}(\mathbb{F}_p)$, with $\mathcal{C}/\mathbb{F}_p$ a genus-2 curve, p a 128-bit prime ($\implies$ elements of $\mathcal{G}$ encode to 256 bits)

Scalar multiplication

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

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 pairs: one public and one private key.

Every public key poses a mathematical problem; the private key is the solution.

Here, every keypair presents an instance 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 $\mathcal{G}$.

1. The security of keys is algorithmic.
2. Cryptanalysis begins at the moment a keypair is created, *not* when it is actually used!
3. It can be *much* easier to attack sets of keys than to attack individual keys.

Where we're going

1. Identification
2. Signatures
3. Key exchange
4. Encryption
5. More key exchange

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?

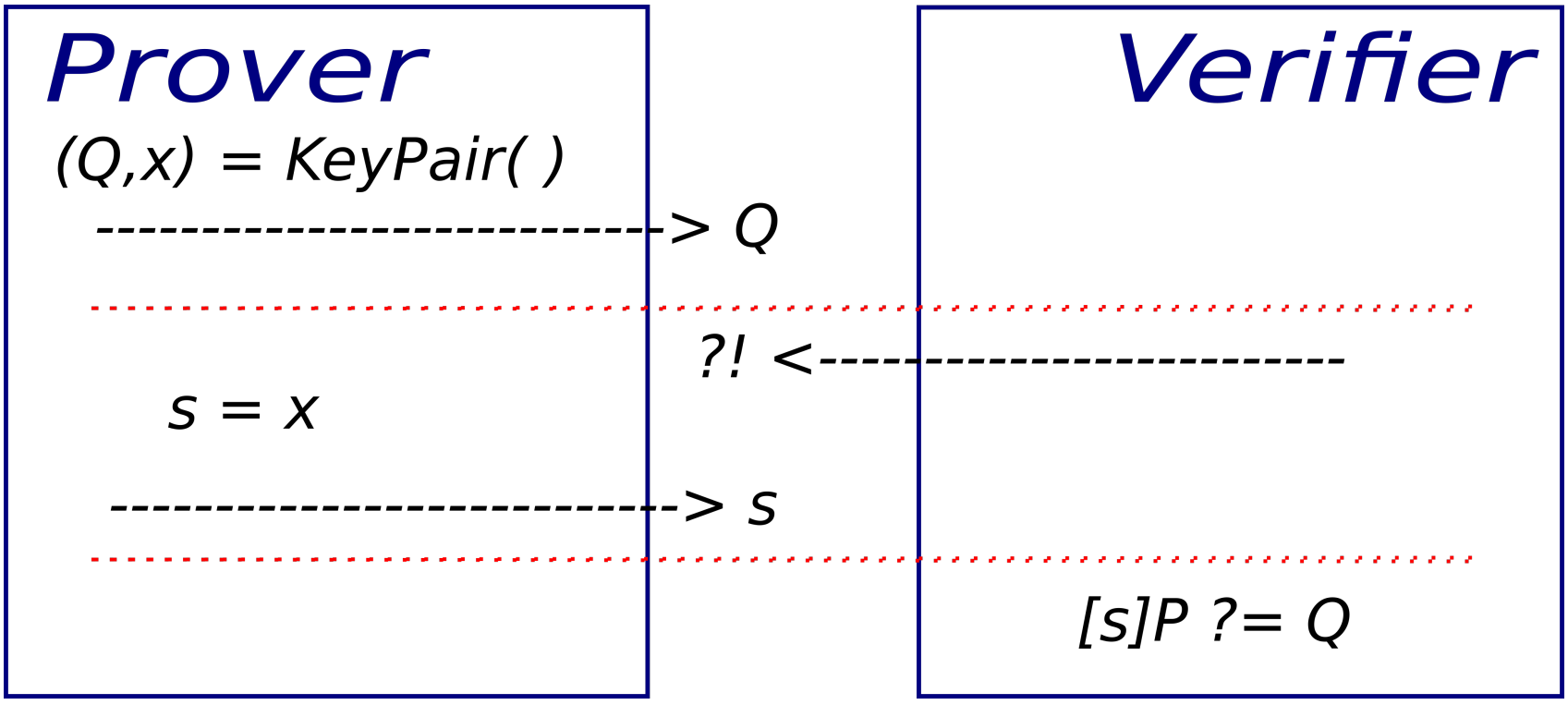
In our setting, you assert/claim an identity
by publishing ("committing") a public key Q
from a keypair $(Q = [x]P, x)$.

Now, how do you prove you know the discrete log 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

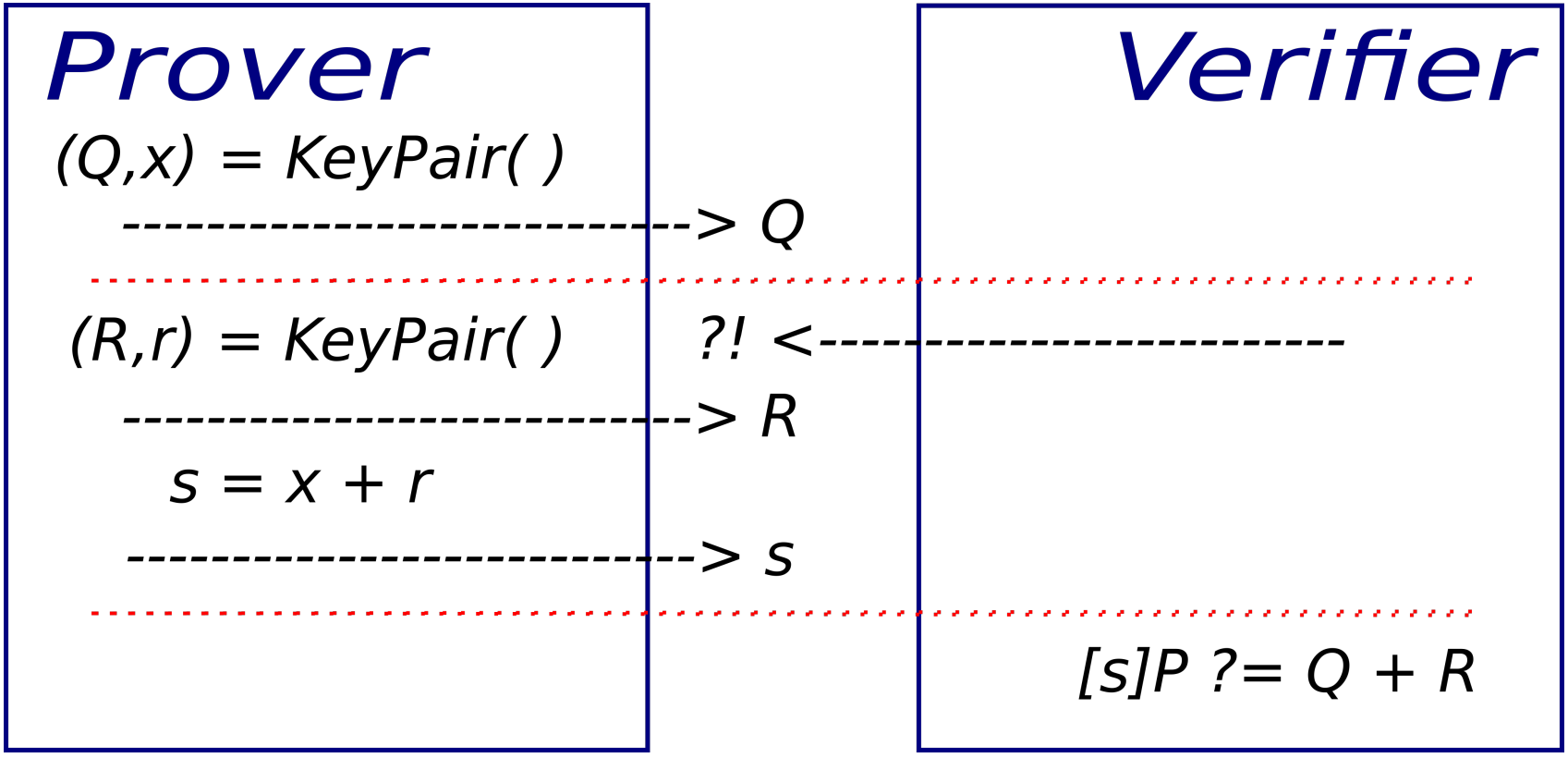


Verifier challenges; Prover returns x;
Verifier accepts iff $[s]P = Q$.

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

Ephemera

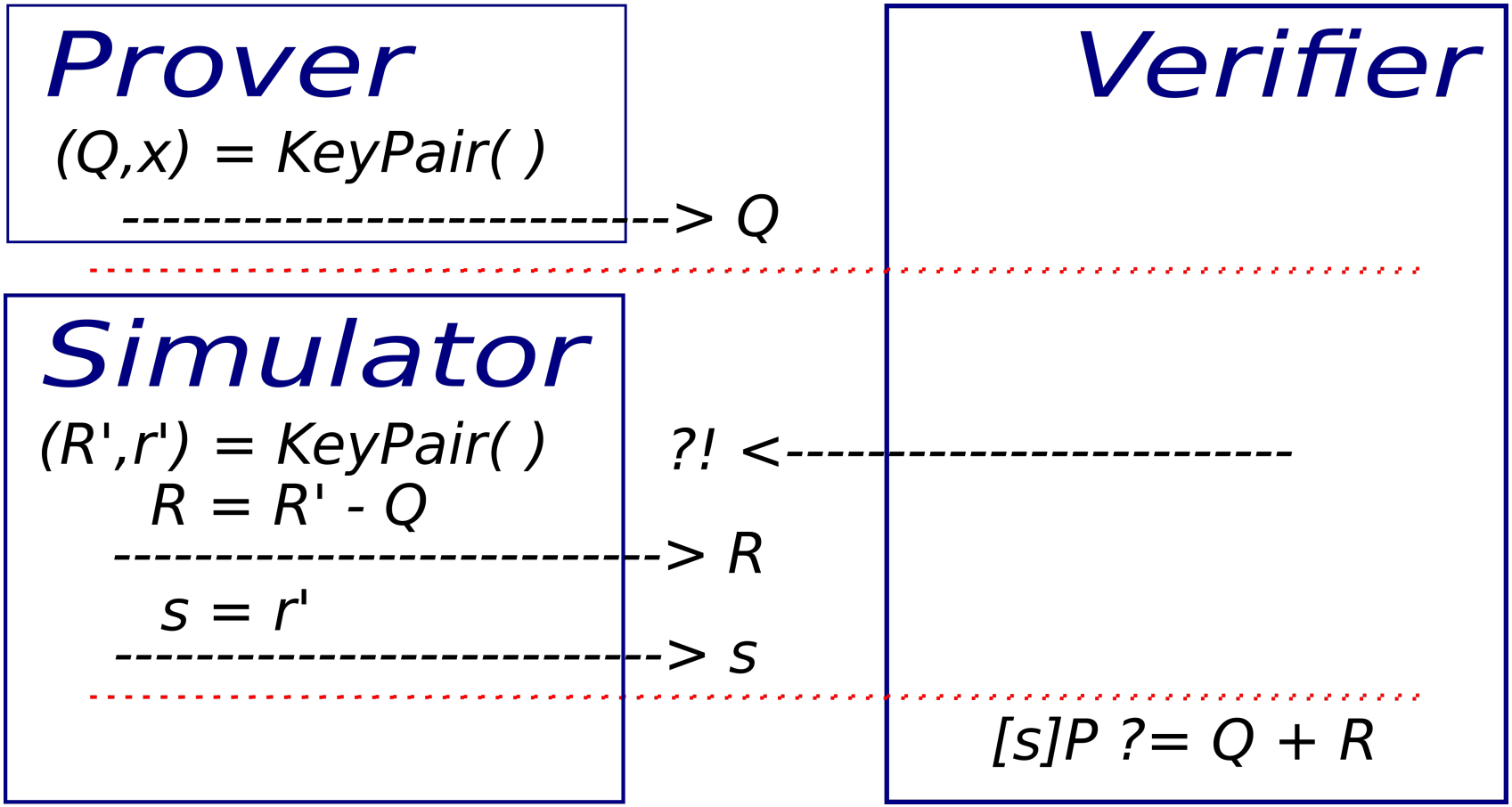
Trick: hide long-term secrets with disposable one-shot secrets.



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

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 distinguish between Prover and Simulator?

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

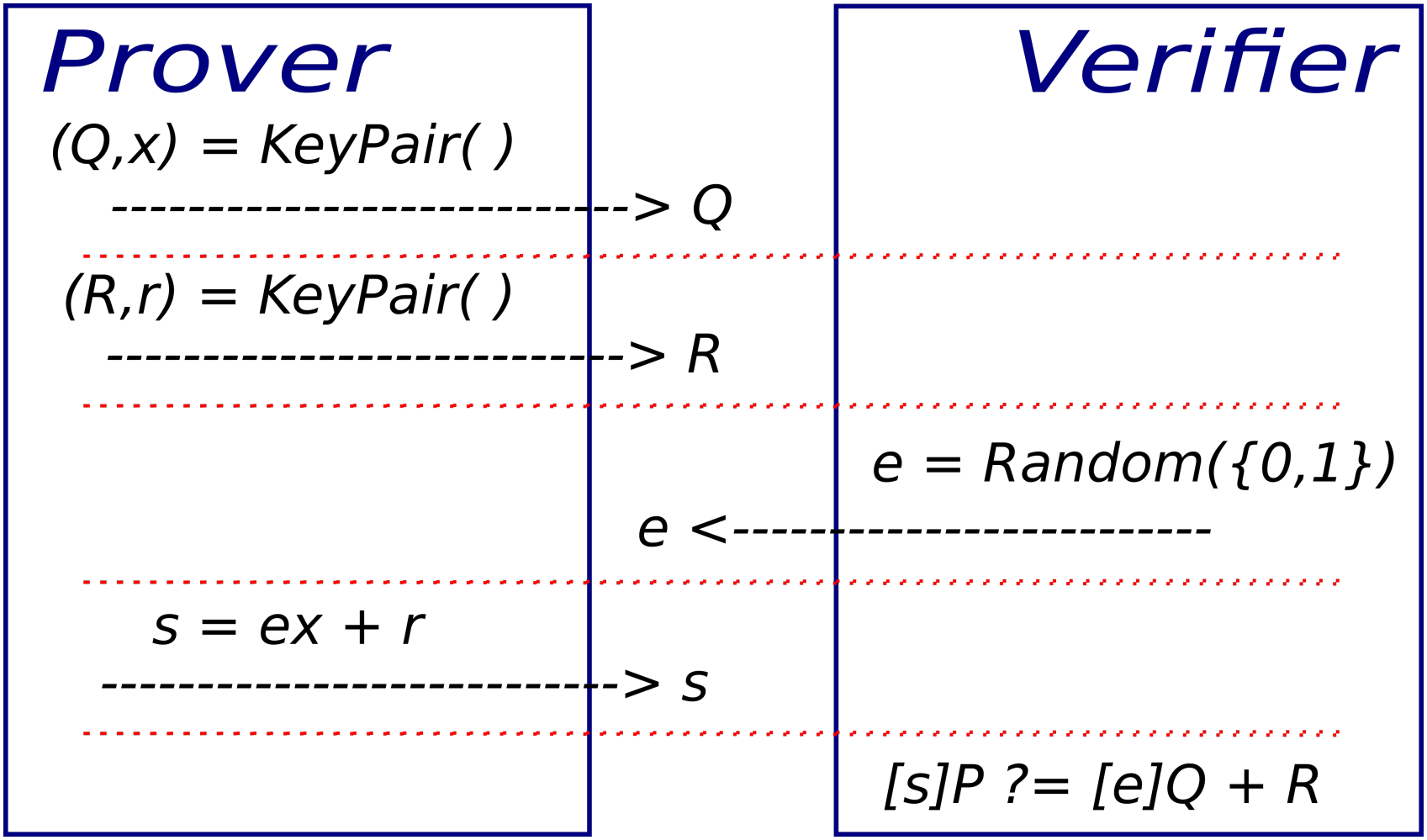
Simulator sends $s = \log(Q + R)$,
but knows *neither* $x = \log(Q)$ *nor* $r = \log(R)$.

Verifier can't ask for x. She could ask for the ephemeral secret
$r = \log(R)$ *as well as* s,
but that would reveal x.

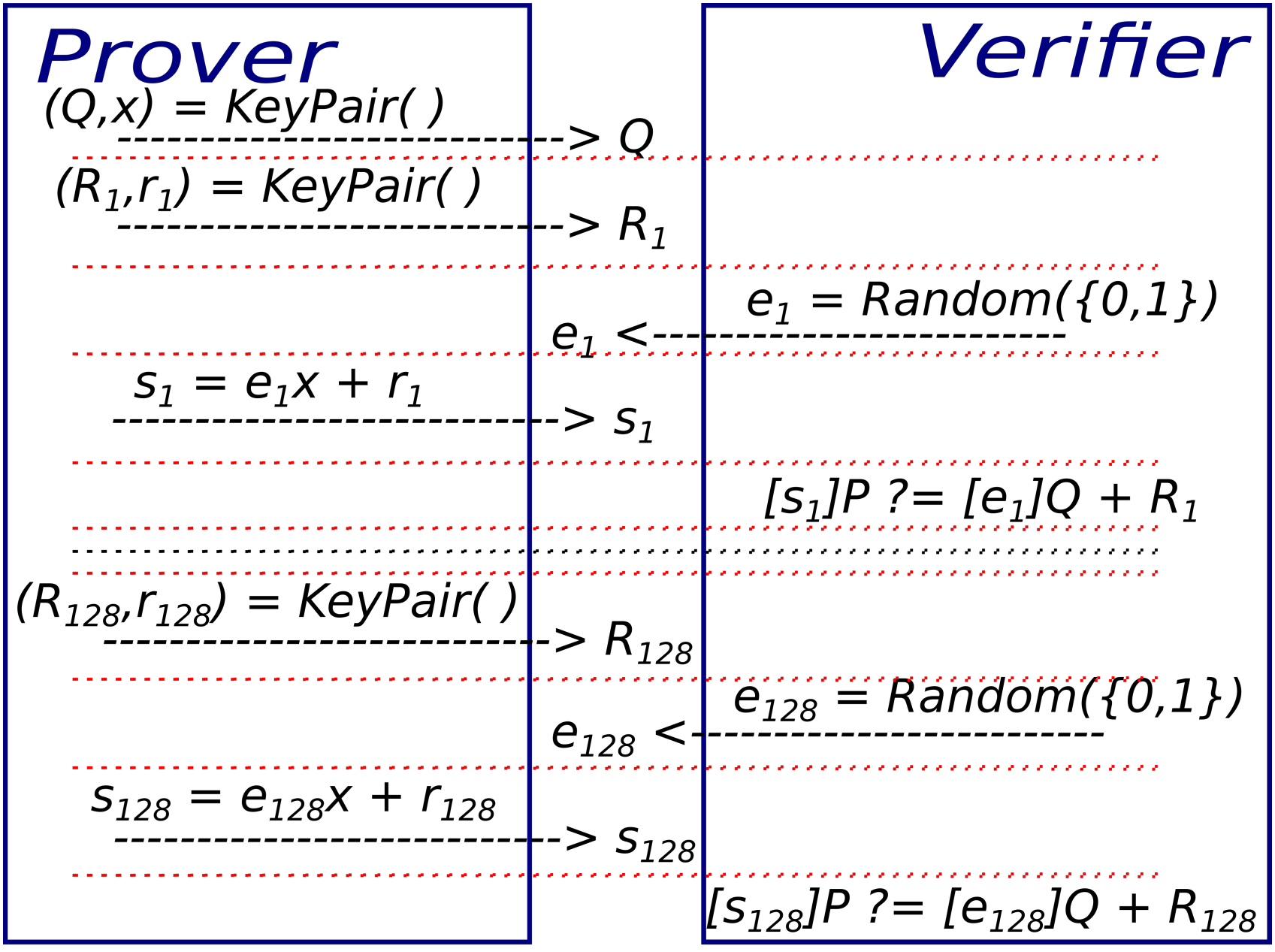
Alternatively: let Verifier ask for **either** s **or** r
(and check $[s]P? = Q + R$ or $[r]P = R$).

- 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)



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
(R1,r1) = KeyPair( )
R1
e1 = Random({0,1})
e1
s1 = e1x + r1
s1
[s1]P ?= [e1]Q + R1
(R128,r128) = KeyPair( )
R128
e128 = Random({0,1})
e128
s128 = e128x + r128
s128
[s128]P ?= [e128]Q + R128

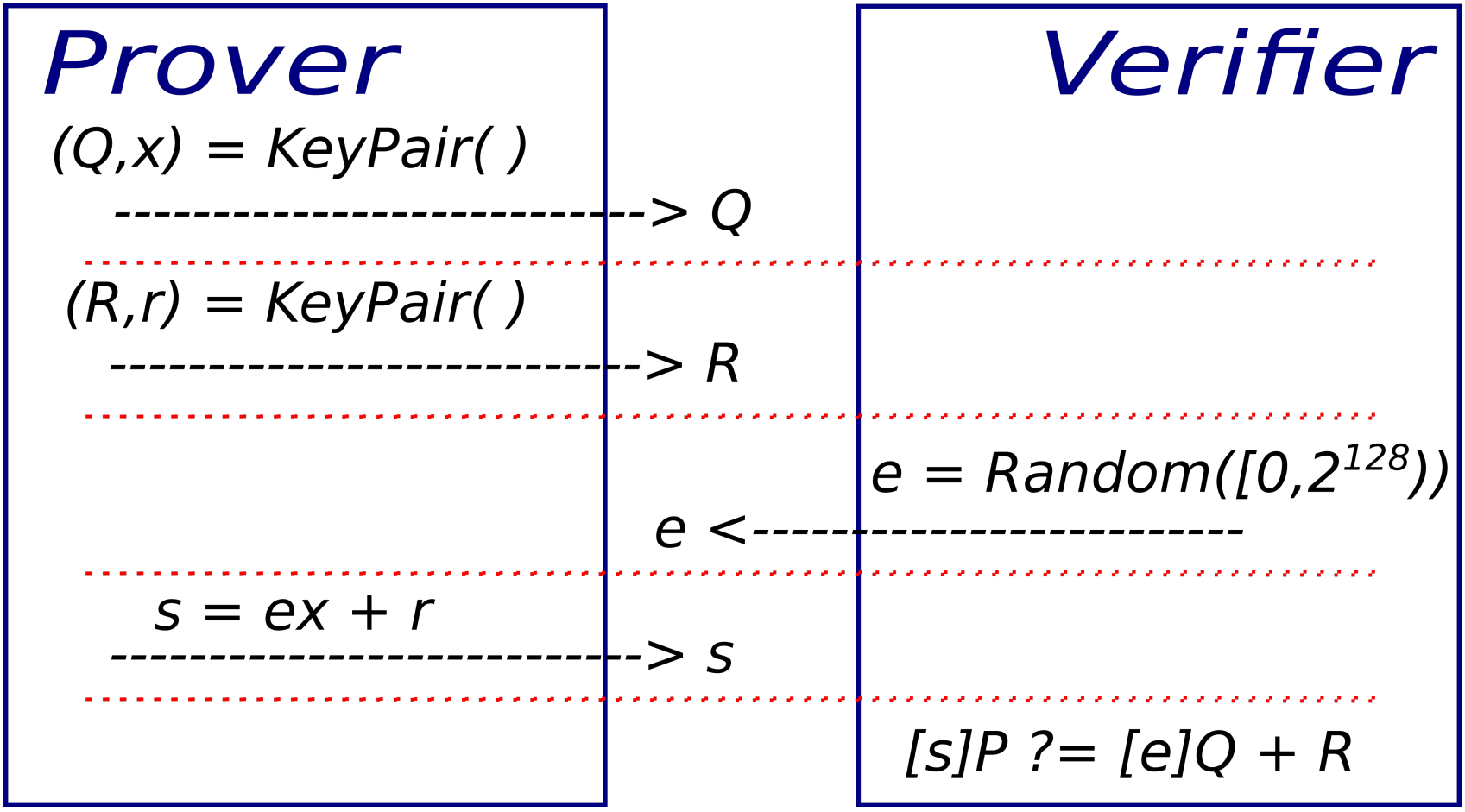
Schnorr ID (1991)

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



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

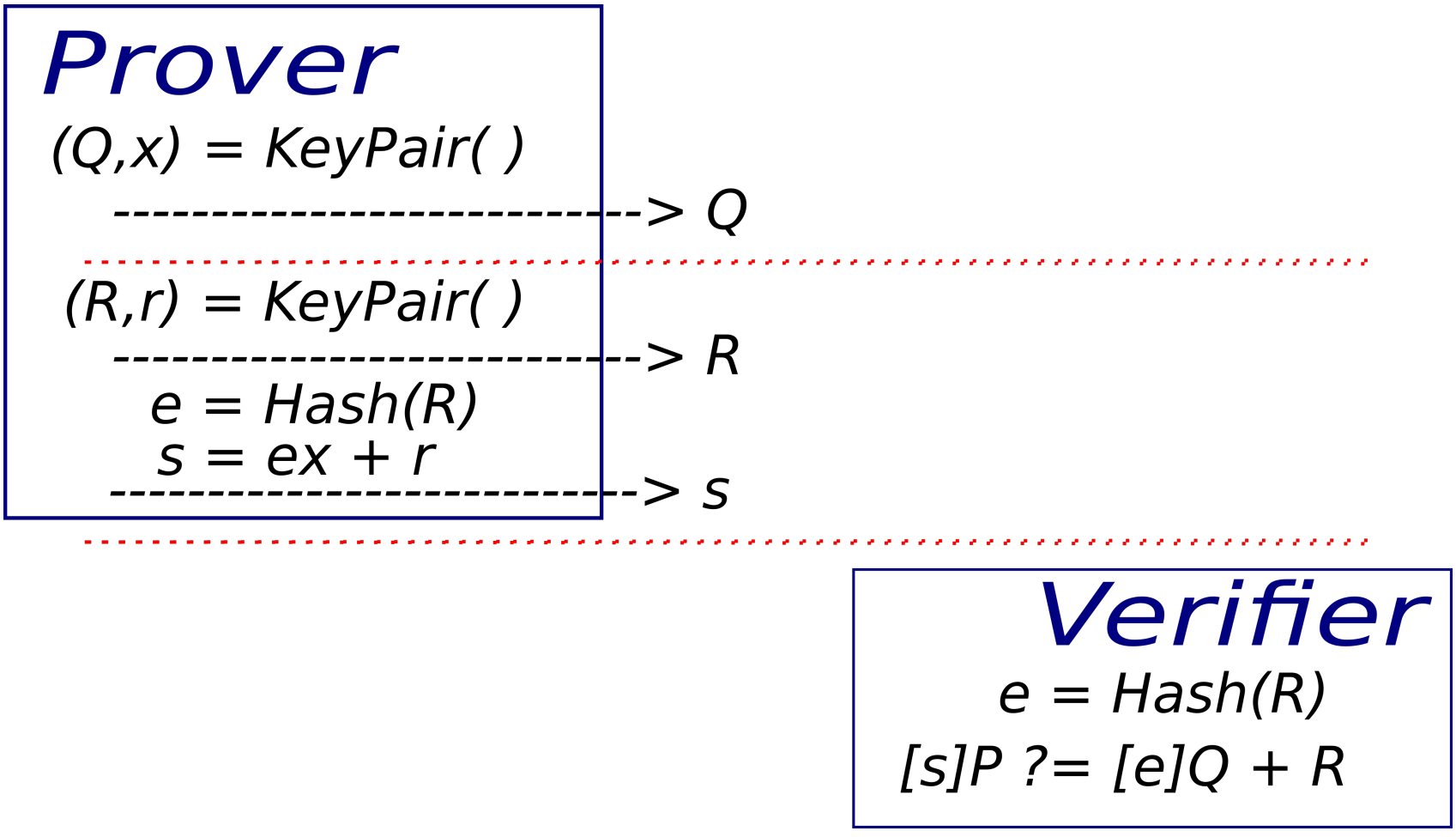
Schnorr identification is efficient, but (1) it's interactive, which is inconvenient; and (2) we're more interested in signatures.

A signature is a sort of non-interactive proof that
the Signer witnessed (created, saw) some data.
Provides *authenticity*, *message integrity*, and *non-repudiability*
because only the Signer could have created it,
and only the Signer's public key is needed to *verify* it.

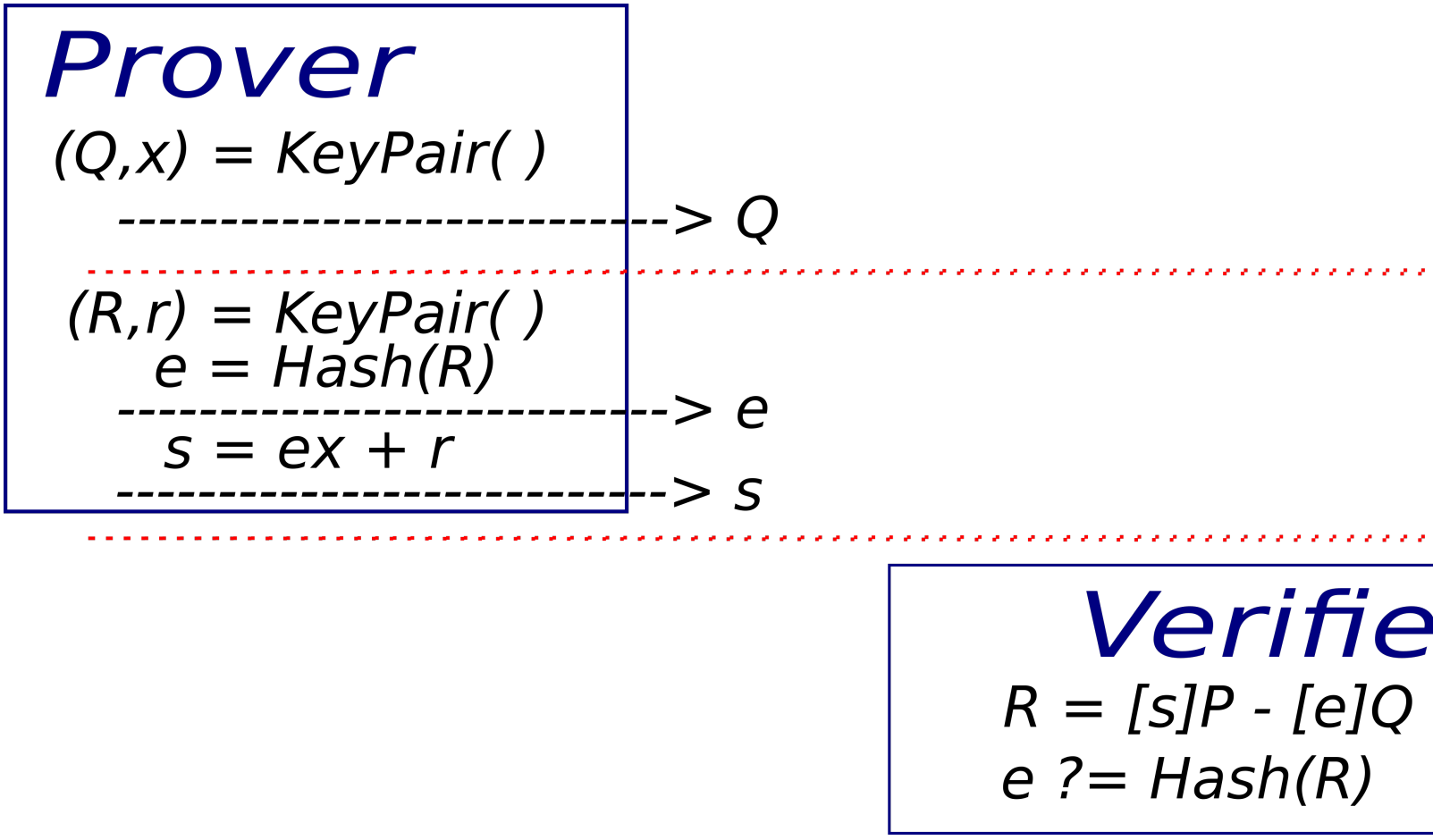
We build *Schnorr signatures* by applying
the *Fiat–Shamir transform* to the Schnorr ID scheme:

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

“Non-interactive Schnorr”

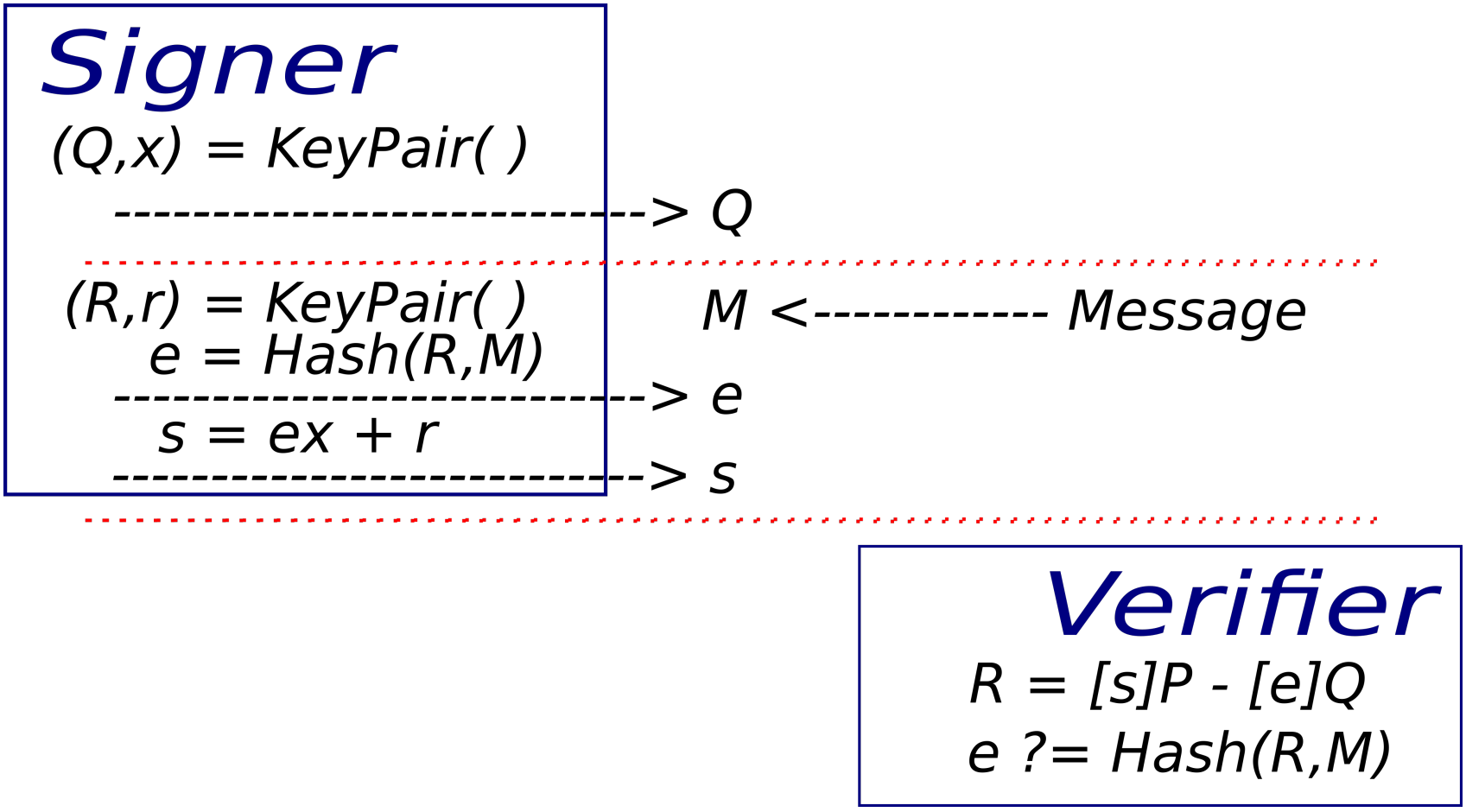


"Compact non-inter Schnorr"



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

Schnorr signatures (1991)



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

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

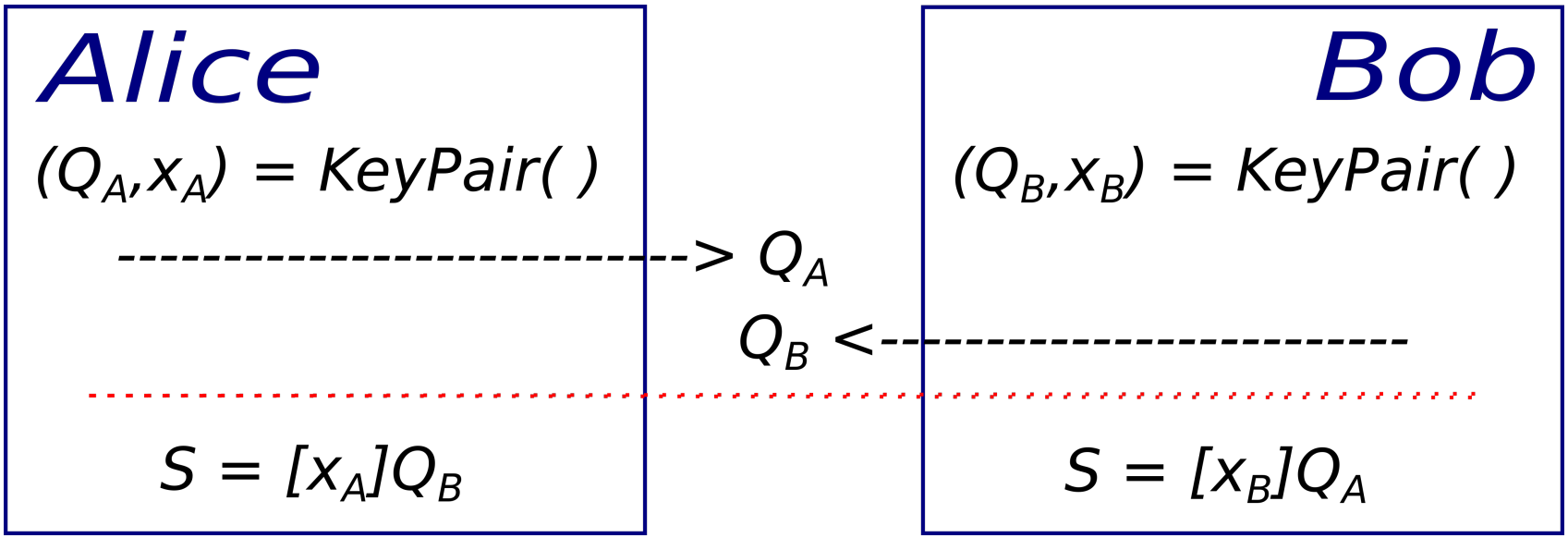
In our signatures, we *mask* secret scalars using addition in $\mathcal{G}$,
which becomes *addition* of scalars.

For 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 & 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

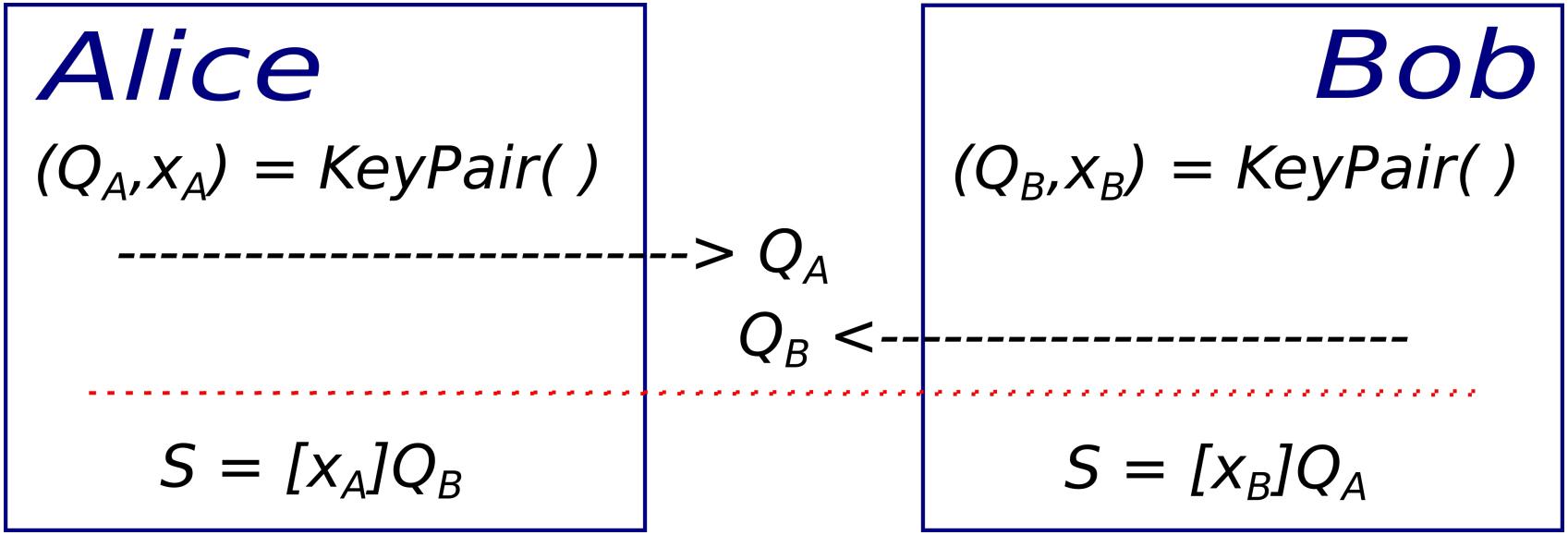
D–H key exchange does not depend directly on the DLP,
but rather on the Computational Diffie–Hellman Problem:

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

Clearly if you can solve the DLP, then you can solve CDHPs.
The opposite direction 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



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.

You've already seen this in Curve25519, where $\mathcal{G} = \mathcal{E}/\pm 1$,
On Thursday you can see it in SIDH (Craig II).

Public-key encryption

A classic textbook problem
that almost nobody has in practice.

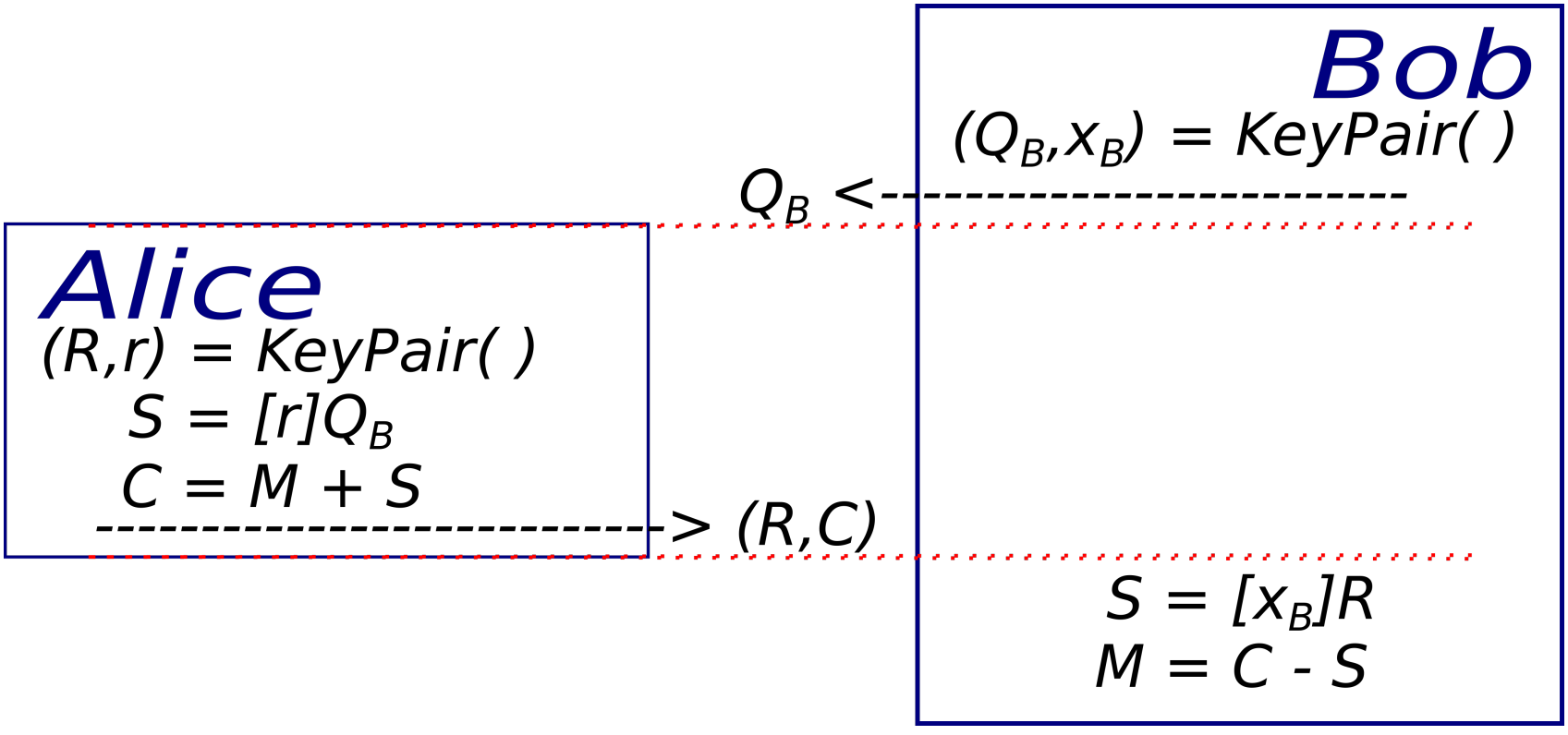
Alice wants to encrypt a message M for Bob.
Bob has a long-term keypair (Q_B, x_B).

Simple approach (ElGamal):

Alice views Q_B as Bob's half of a DH key exchange,
completes the Diffie–Hellman on her side,
uses the shared secret to encrypt M,
and sends her half of the DH with M.

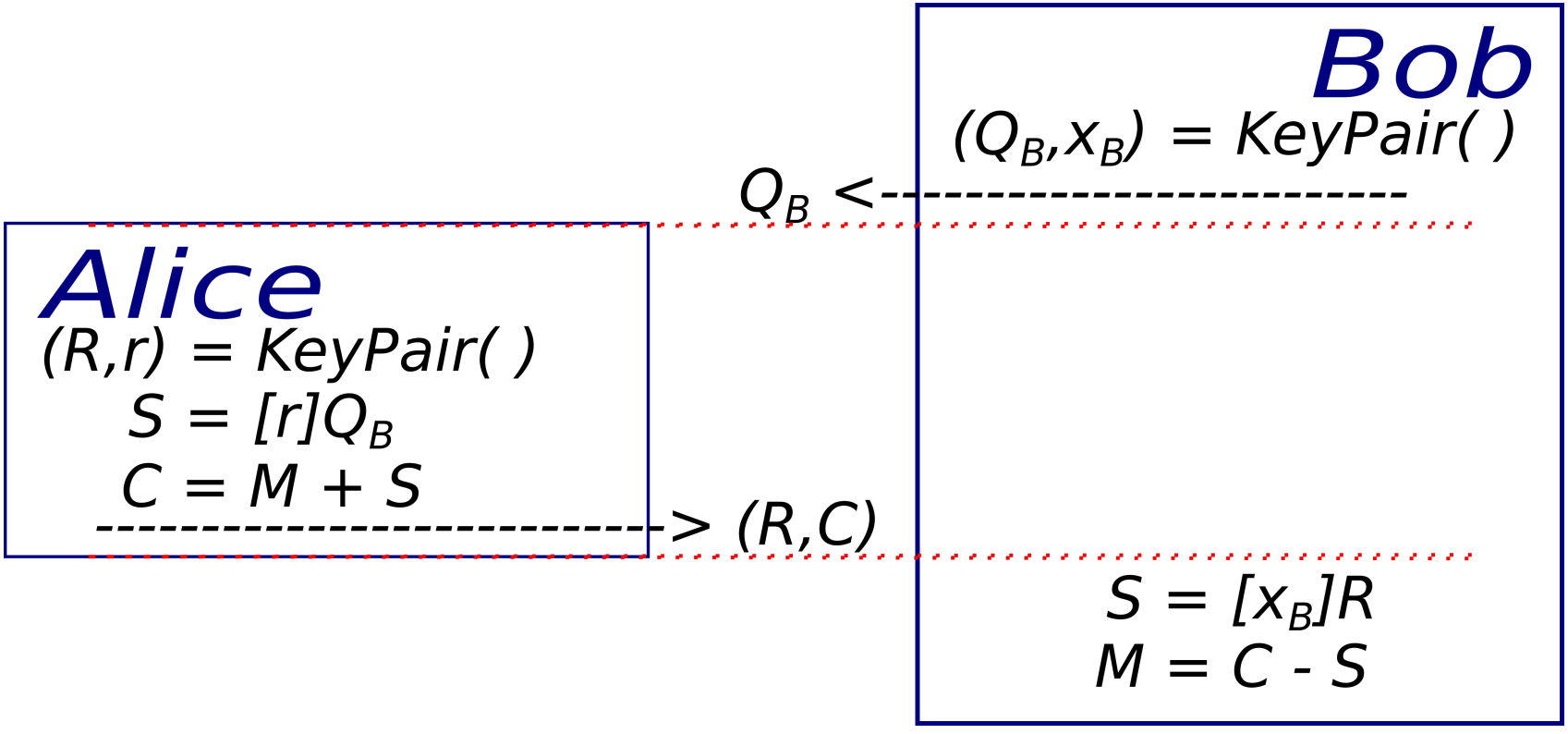
To decrypt, Bob completes the DH on his side.

Classic ElGamal encryption (1984)



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

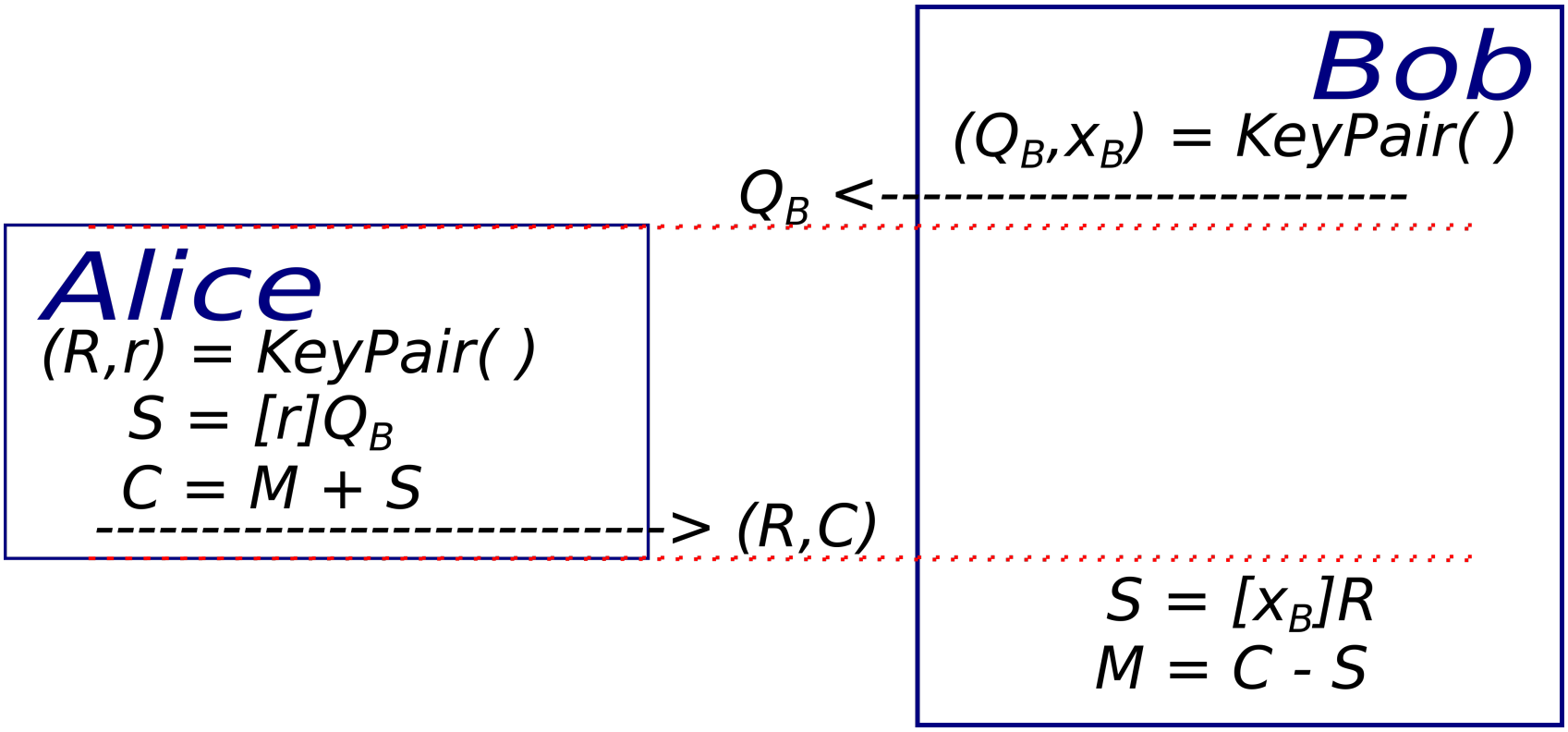


Problem: ElGamal is homomorphic!

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



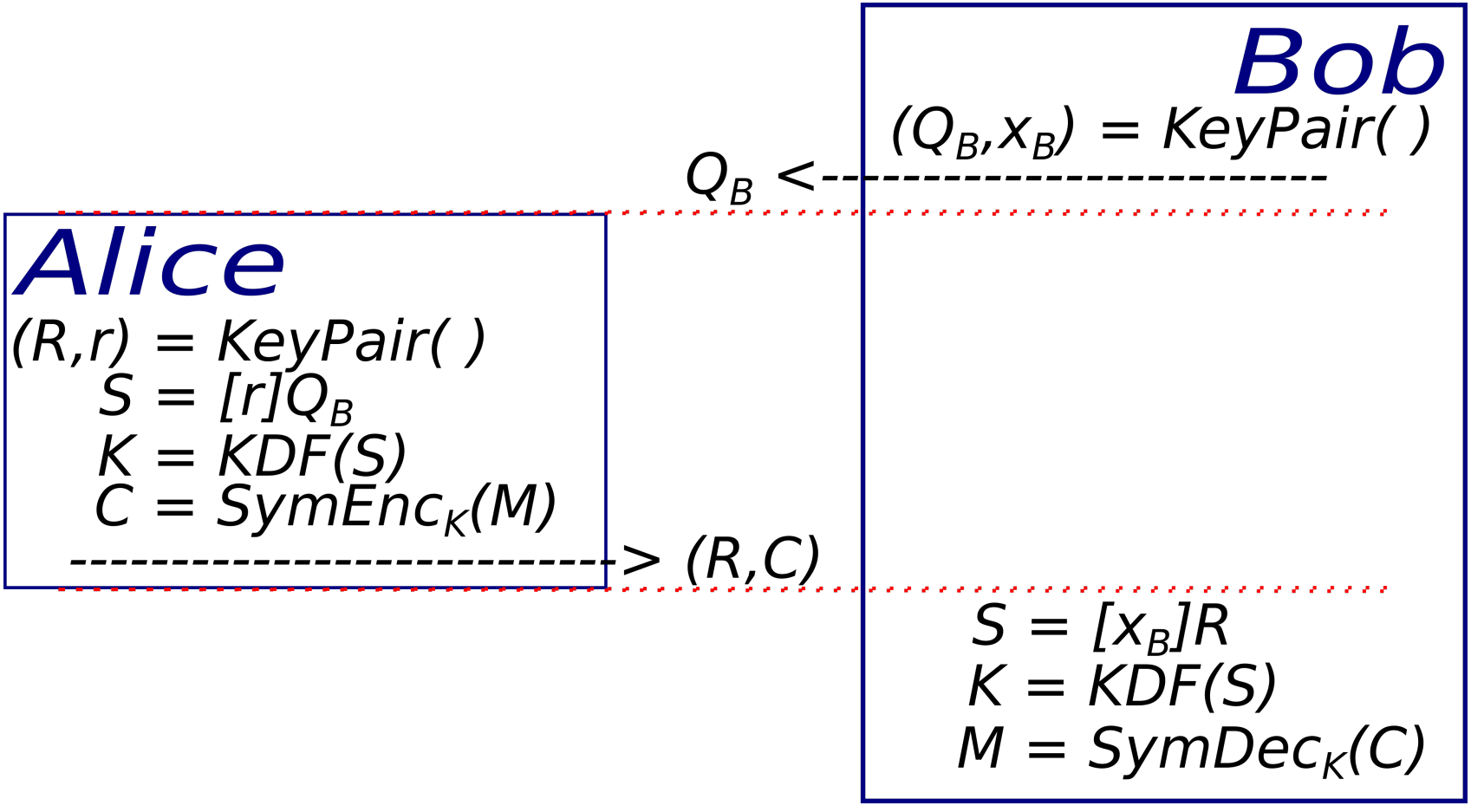
We have a deeper categorical/typing/casting problem:
Real messages are blobs of bits *(or strange domain elts!)*,
not elements of $\mathcal{G}$.
Real ciphertexts should be random-looking bitstrings
(or strange codomain elts), not elements of $\mathcal{G}$.

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



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.

Chasing your tail

At this point we could define ElGamal signatures, then derive DSA, then define ECDSA...

But we've already defined decent signatures (and I'd recommend EdDSA, which is Schnorr-like).

So let's get back to the authentication problem: in particular, *authenticated* key exchange.

X3DH: Extended triple DH

X3DH (Marlinspike–Perrin, 2016)
is an authenticated key agreement scheme
suitable for *asynchronous* communication.

Notably used to kick off communications in
the Signal protocol (used by Signal and WhatsApp).

X3DH requires a server, that is *not trusted*.

The standard version of X3DH is built around
X25519 (a Modern Diffie–Hellman key exchange)
and XEdDSA (a Schnorr-like signature scheme).

X3DH: the aim

From the spec:

Alice wants to send Bob some initial data using encryption,
and also establish a shared secret key
which may be used for bidirectional communication.

Bob wants to allow parties like Alice
to establish a shared key with him and send encrypted data.

However, *Bob might be offline* when Alice attempts to do this.
To enable this, Bob has a relationship with some **server**.

The server can store messages from Alice to Bob
which Bob can later retrieve.
The server also lets Bob publish some data,
which the server will provide to parties like Alice.

X3DH: three phases

The X3DH protocol is carried out in 3 phases, each typically happening some time apart.

1: Bob publishes his *identity key* and *prekeys* to some (untrusted) server.

2: Alice fetches a *prekey bundle* from the server, and uses it to send an initial message to Bob *e.g. starting an encrypted chat session*.

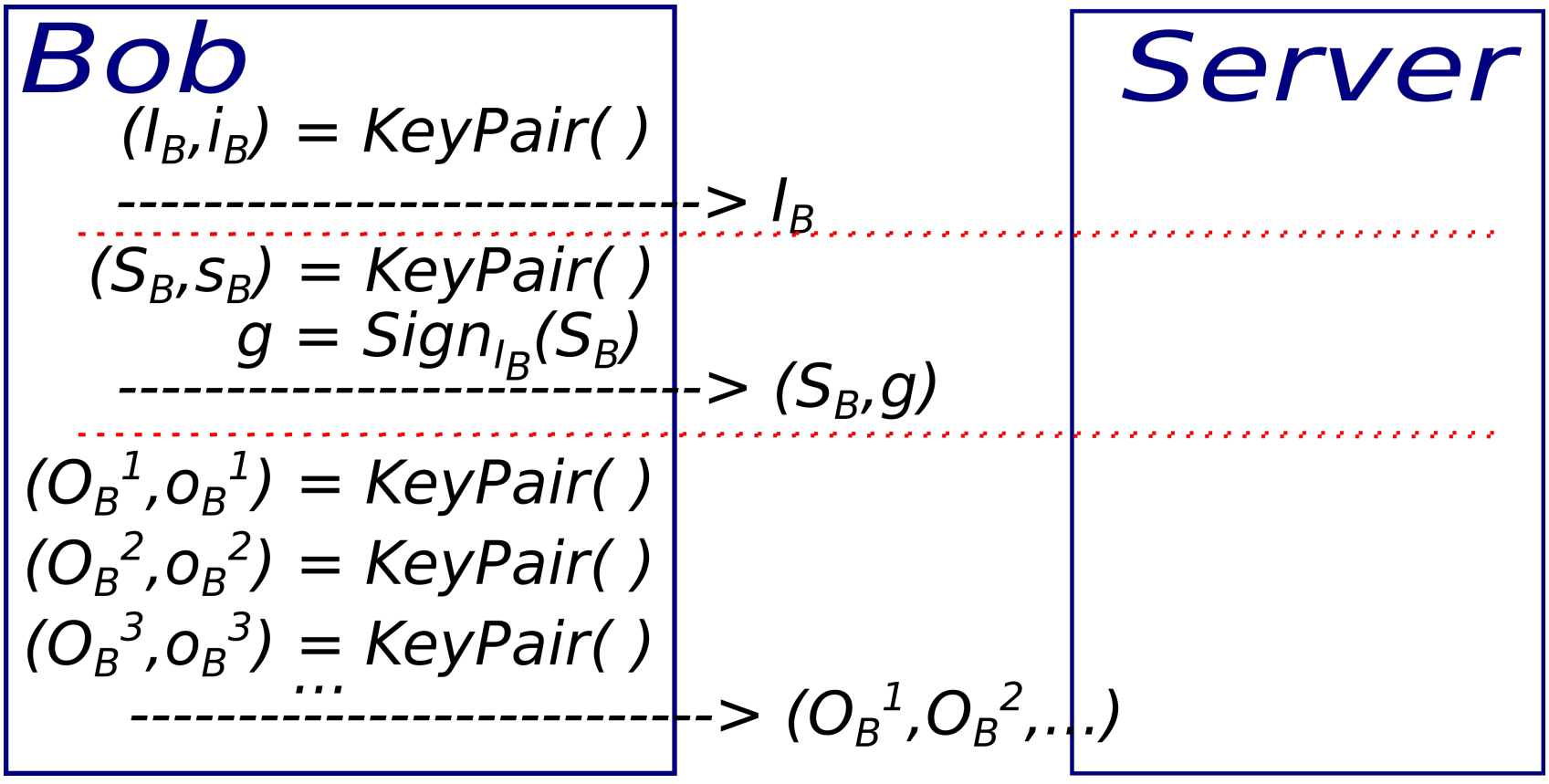
3: Bob receives and processes Alice's message.

X3DH: lots of keys

There are a lot of keypairs involved in X3DH...

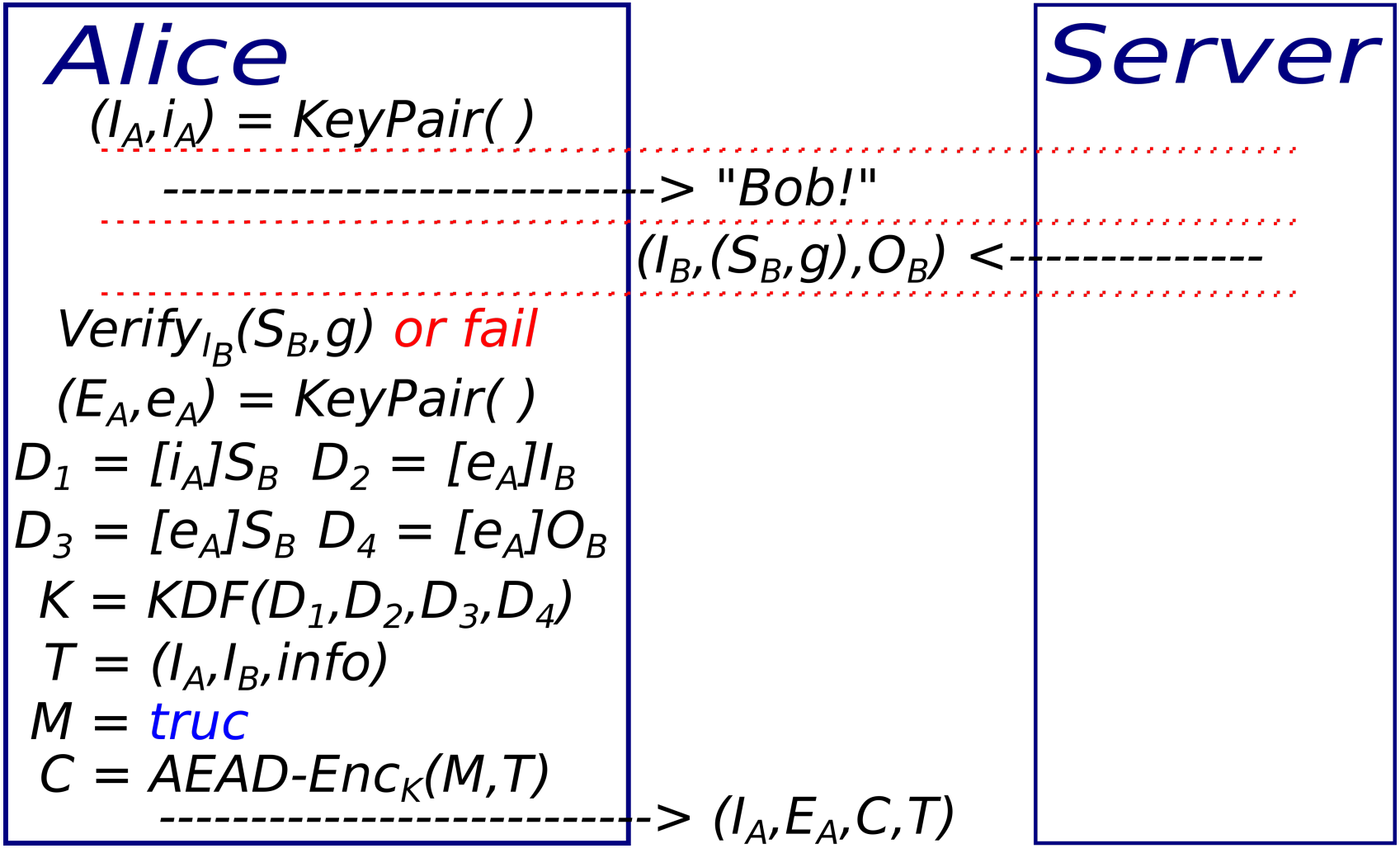
- (I_B, i_B) = Bob's long-term **identity** keypair
  (used for signatures)
- (S_B, s_B) = Bob's **signed prekey** pair
  (to be changed periodically)
- (E_B, e_B) = Bob's **ephemeral** keypair
- (O_B, o_B) = Bob's **one-time prekey** pair
  (one of many, to be used one per X3DH)

X3DH: Phase 1



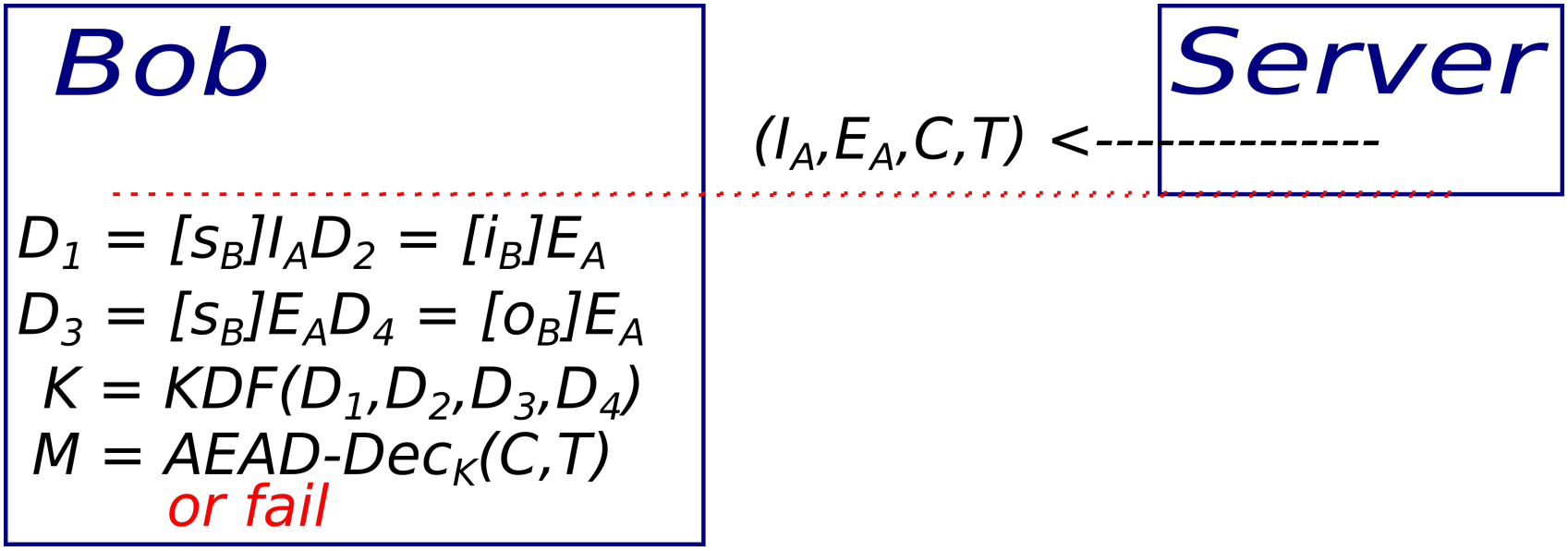
Bob uploads his ID key I_B, Signed prekey (S_B, g), and a list of one-time prekeys O_B^i.

X3DH: Phase 2



Here *truc* is whatever you like: the start of a conversation, or just an empty message.

X3DH: Phase 3



If the X3DH protocol finally completes
(i.e. Alice's Verify and Bob's AEAD-Dec don't fail), then

- Alice and Bob are mutually authenticated;
- they have a shared secret key K; and
- Alice has also sent an encrypted message M to Bob
  (eg. the start of an encrypted chat).

Ref: `https://whispersystems.org/docs/specifications/x3dh`