Software Implementation of (Post-Quantum) Public-Key Cryptography

Curve-Based

Patrick Longa

Microsoft Research

SPACE 2020

Virtual Conference, December 2020

Outline

- Preliminaries: public-key cryptography
- (Post-quantum) public-key cryptography
  - Motivation: the quantum menace
- Implementation aspects of PKC
  - Preliminaries: CISC vs RISC, x64 ISA
  - Side-channel attacks: timing and cache attacks
  - Computer representation
  - Computer arithmetic (cases with ECC and SIKE)
  - Modular arithmetic
- The case of ECC
- The case of SIKE

Cryptography in use today

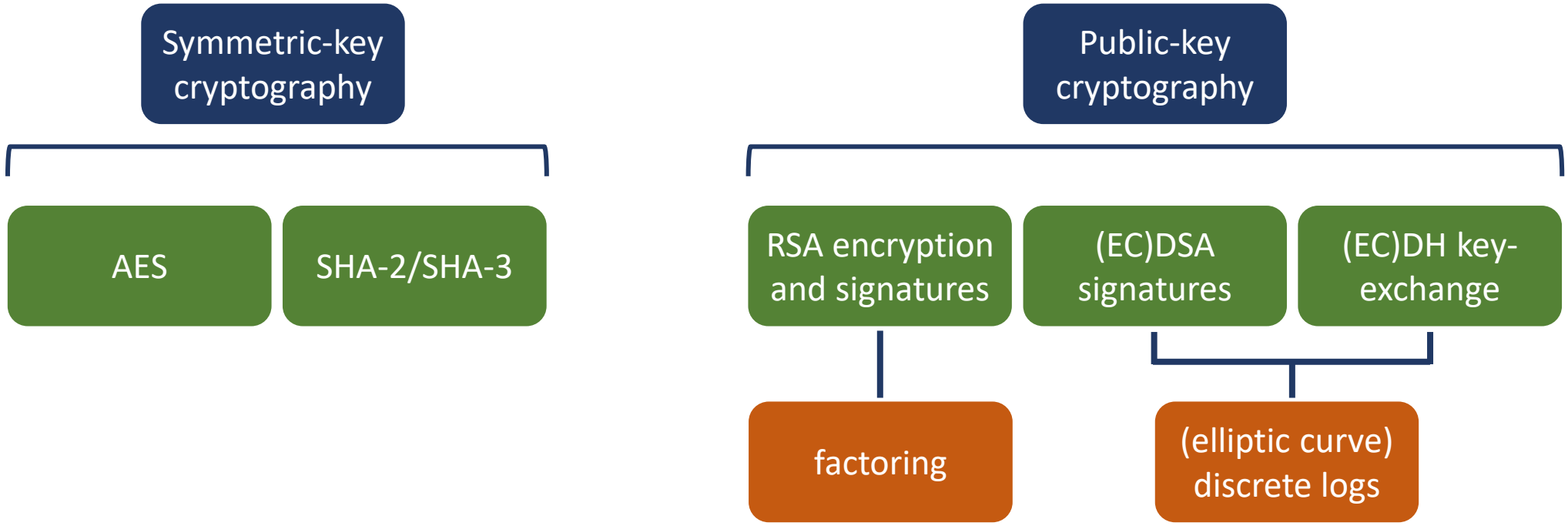
Symmetric-key cryptography

Public-key cryptography

Cryptography in use today

Cryptography in use today

Cryptography in use today



Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

- “Classical” public-key cryptography:

Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

- “Classical” public-key cryptography:
  - **Finite fields** (1976): discrete logarithm problem (DLP)

Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

- “Classical” public-key cryptography:
  - **Finite fields** (1976): discrete logarithm problem (DLP)
  - **RSA** (1977): integer factorization problem (IFP)

Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

- “Classical” public-key cryptography:
  - **Finite fields** (1976): discrete logarithm problem (DLP)
  - **RSA** (1977): integer factorization problem (IFP)
  - **ECC** (1985): elliptic curve discrete logarithm problem (ECDLP)



Public-key cryptography

- Public discovery by Whit Diffie and Martin Hellman (“New directions in cryptography”, 1976).
- Classified discovery at the GCHQ in the early 70’s.

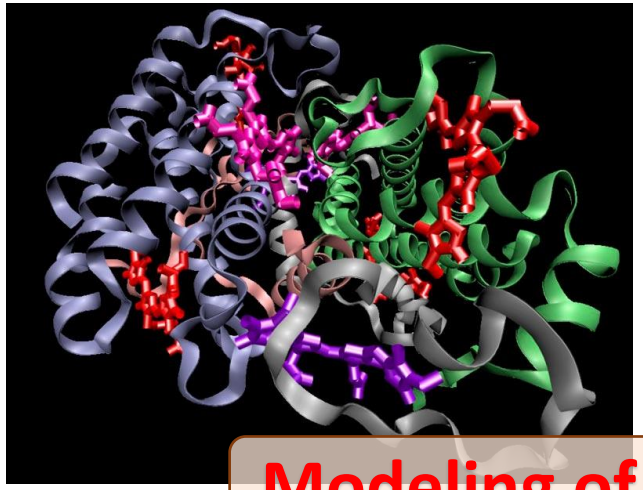
- “Classical” public-key cryptography:
  - **Finite fields** (1976): discrete logarithm problem (DLP)
  - **RSA** (1977): integer factorization problem (IFP)
  - **ECC** (1985): elliptic curve discrete logarithm problem (ECDLP)

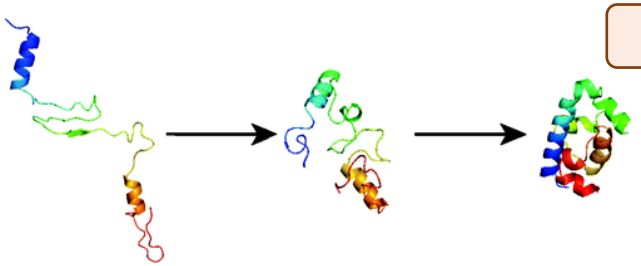
Widely dominated by number theoretic cryptography.



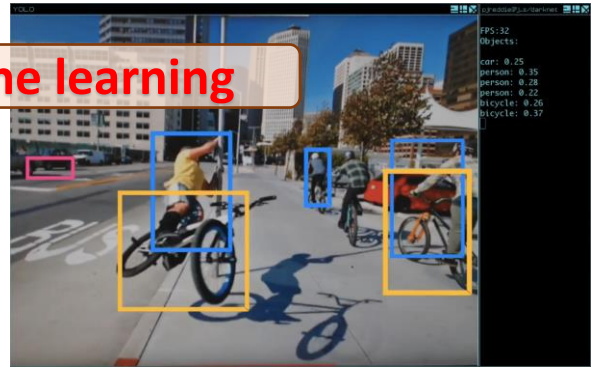
Motivation: the quantum menace

Quantum computing

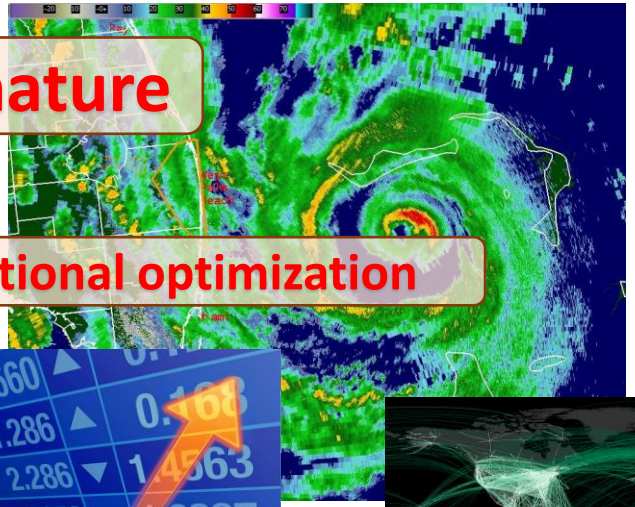




Machine learning

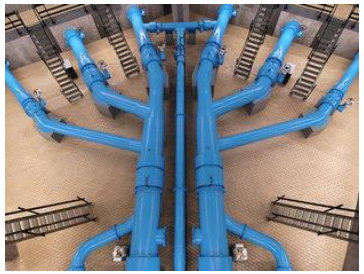


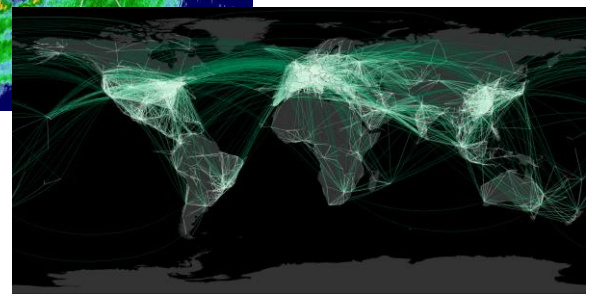
Modeling of nature



Computational optimization

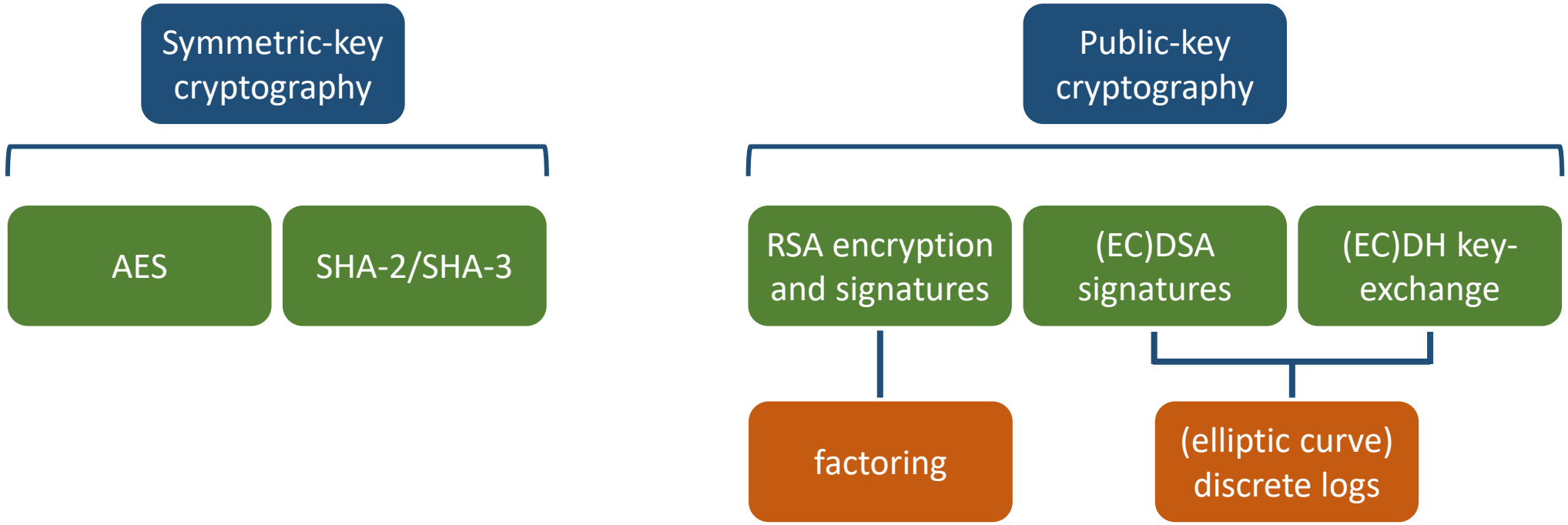
Database search



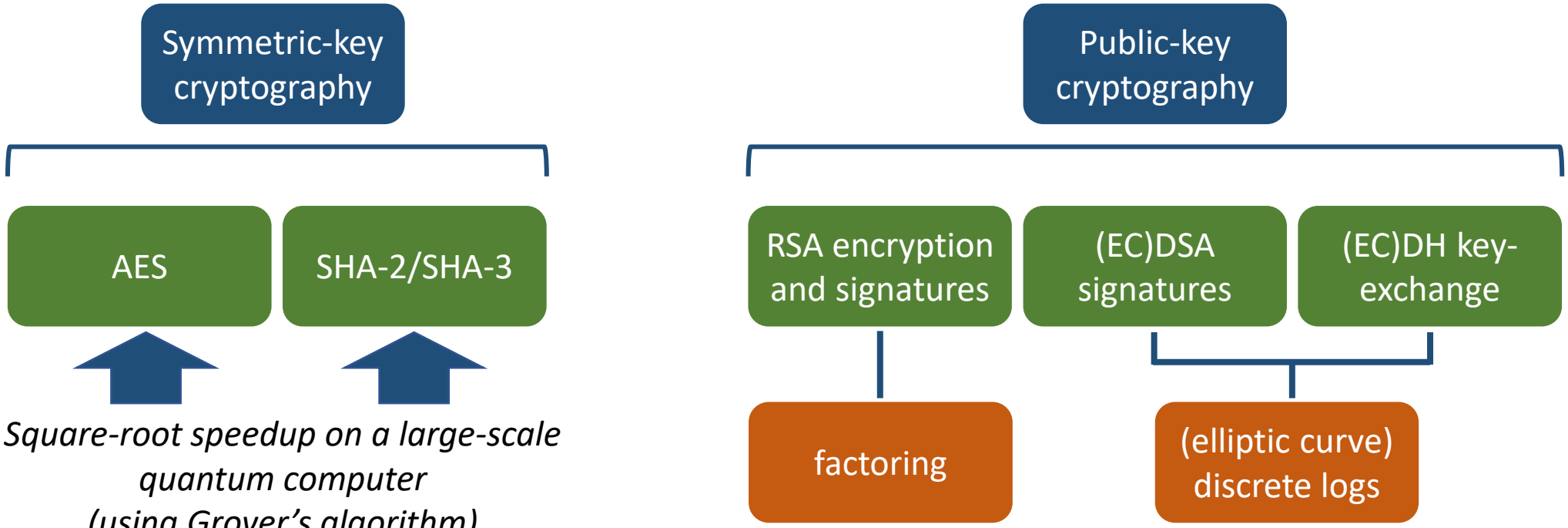


Quantum computing

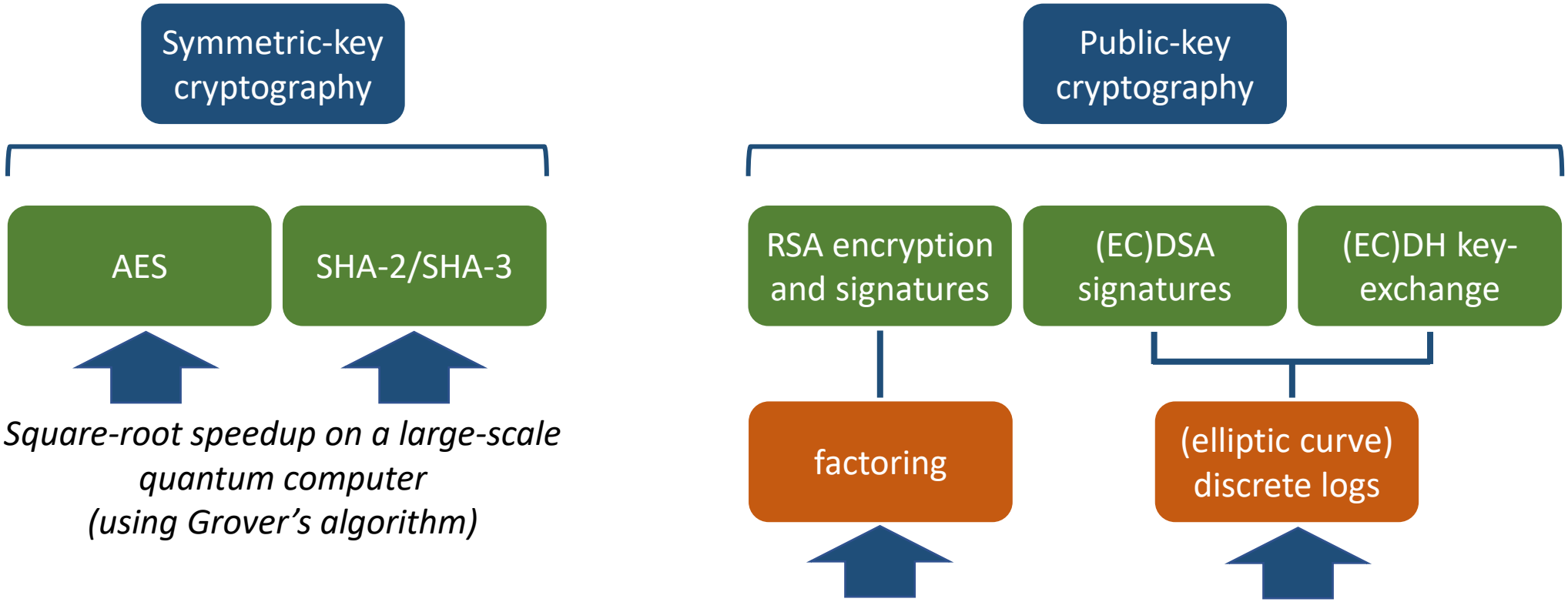
Cryptography in use today



Cryptography in use today

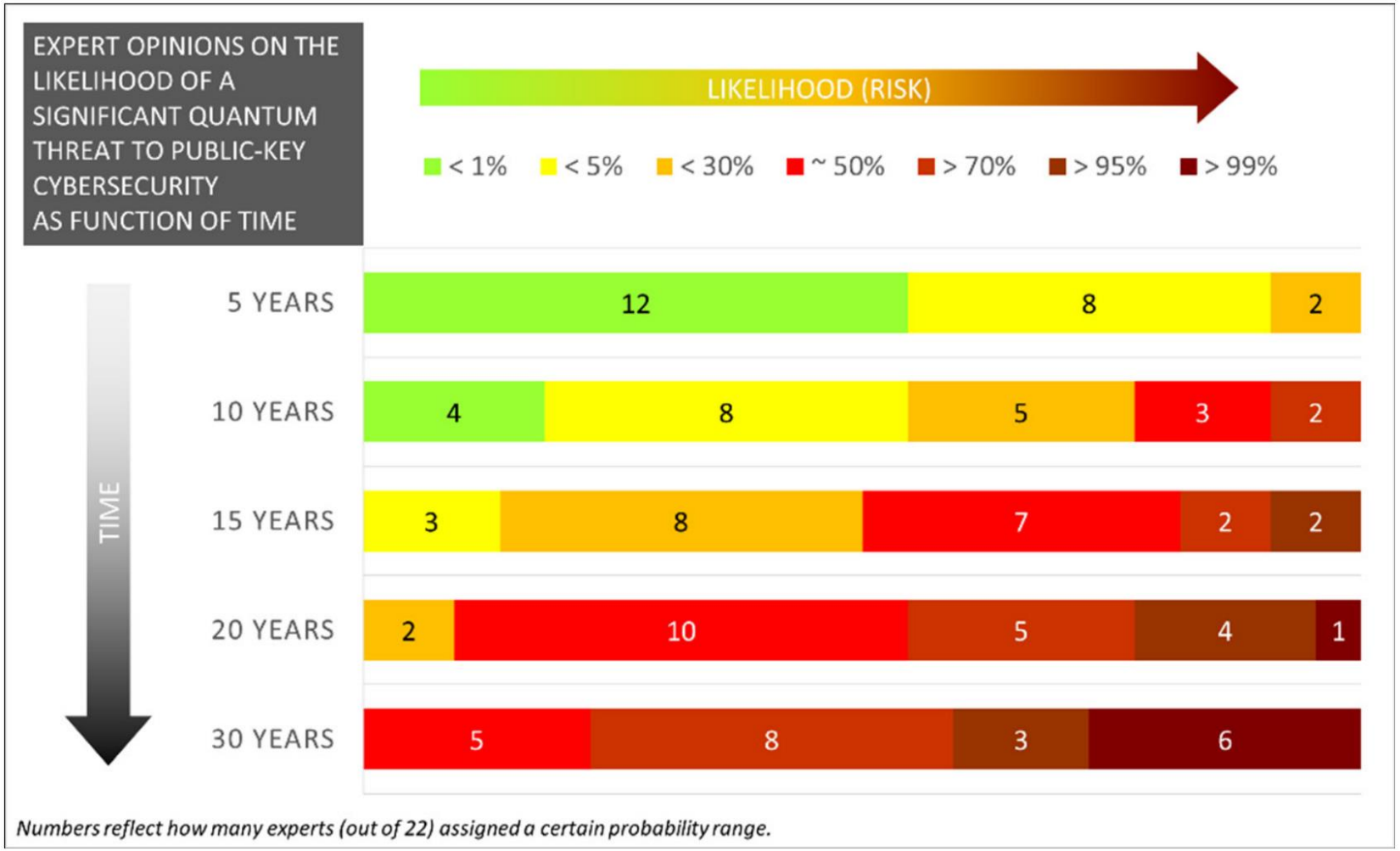


Cryptography in use today



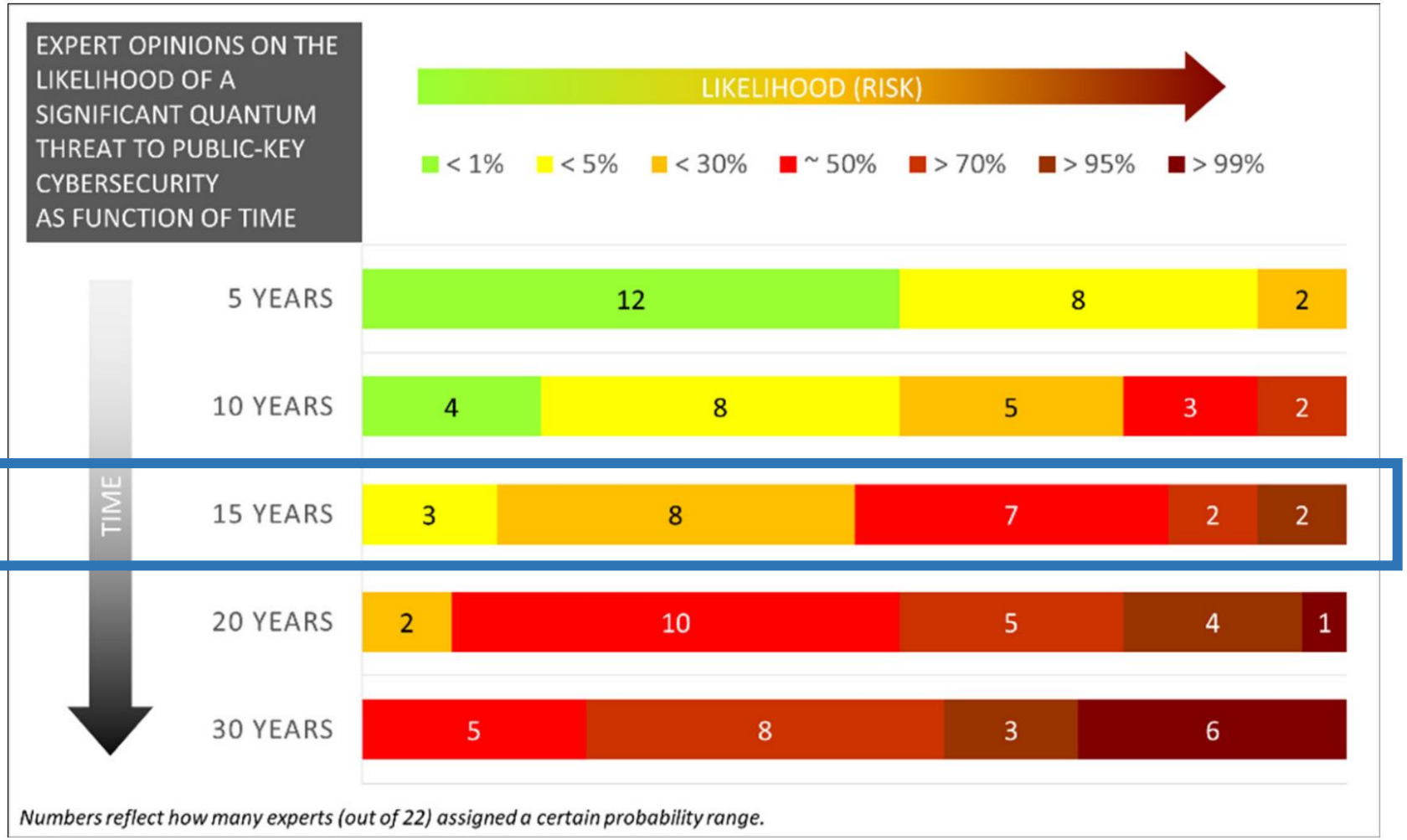
When will a large-scale, fault-tolerant quantum computer be built?

Expert opinions on the likelihood of the quantum threat to current public-key cryptosystems



(*) M. Mosca, M. Piani, “Quantum Threat Timeline”, Global Risk Institute (2019)

Expert opinions on the likelihood of the quantum threat to current public-key cryptosystems

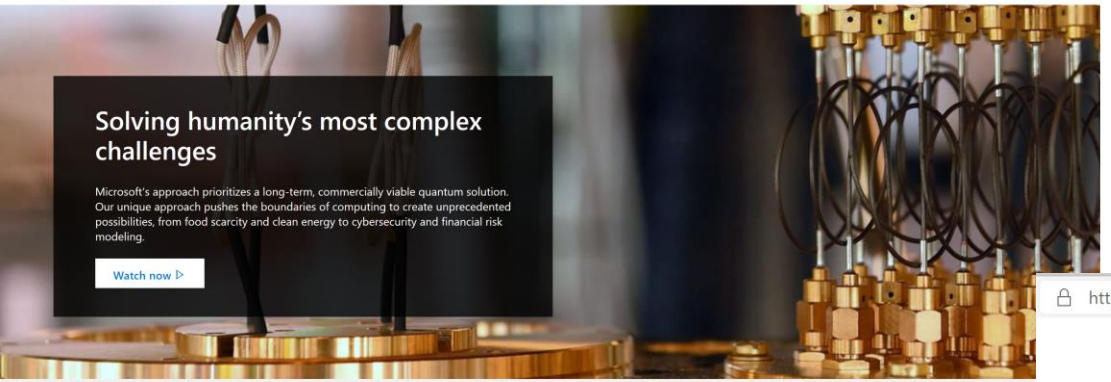


50% respondents say chances are $\geq 50\%$

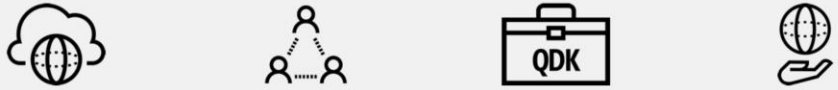
(*) M. Mosca, M. Piani, “Quantum Threat Timeline”, Global Risk Institute (2019)

https://www.microsoft.com/en-us/quantum

Microsoft | Quantum Vision Development kit Quantum network Resources All Microsoft Search

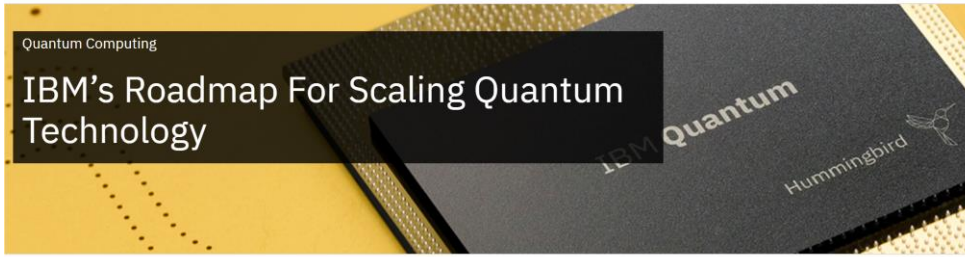


A scalable, open approach to quantum computing



https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/

IBM Research Blog Topics Labs About



September 15, 2020 | Written by: Jay Gambetta

Categorized: Quantum Computing

Back in 1969, humans overcame unprecedented technological hurdles to make history: we put two of our own on the Moon and returned them safely. Today's computers are capable, but assuredly earthbound when it comes to accurately capturing the finest details of our universe. Building a device that truly captures the behavior of atoms—and can harness these behaviors to solve some of the most challenging problems of our time—might seem impossible if you limit your thinking to the computational world you know. But like the Moon landing, we have an ultimate objective to access a realm beyond what's possible on classical computers: we want to build a large-scale quantum computer. The future's quantum computer will pick up the slack where classical computers falter, controlling the behavior of atoms in order to run revolutionary applications across industries, generating world-changing materials or transforming the way we do business.

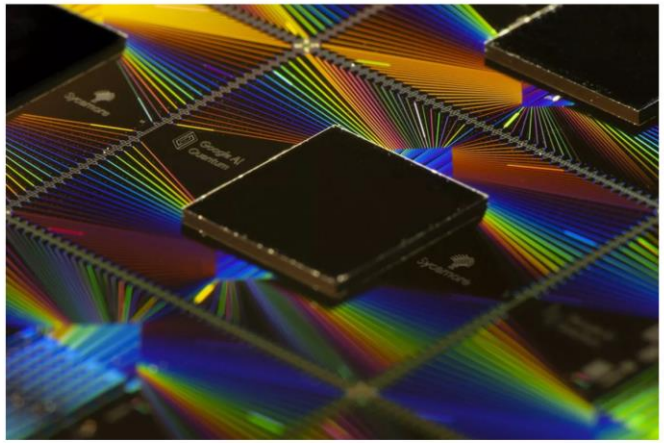
Today, we are releasing the roadmap that we think will take us from the noisy, small-scale devices of today to the million-plus qubit

https://www.theverge.com/2019/10/23/20928294/google-quantum-supremacy-sycamore-computer-c

Google confirms 'quantum supremacy' breakthrough

Its research paper is now available to read in its entirety

By Jon Porter | @JonPorty | Oct 23, 2019, 6:31am EDT



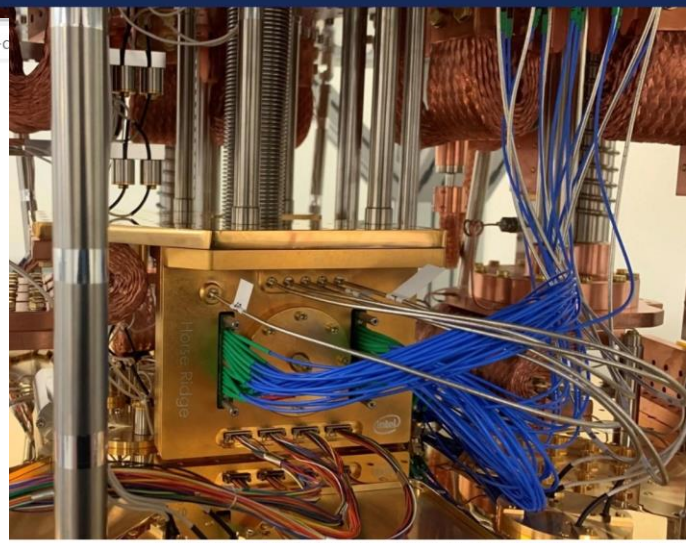
Google's Sycamore quantum processor, which was behind the breakthrough. | Credit: Google

Google has officially announced that it's achieved quantum supremacy in a new article published in the scientific journal *Nature*. The announcement comes exactly one month after it initially leaked, when Google's paper was accidentally published early. Now, however, it's official, meaning the full details of the research are public, and the broader scientific community can fully scrutinize what Google says it's achieved.

https://venturebeat.com/2020/12/03/intel-unveils-its-second-generation-quantum-computing-control-chip/

Intel unveils its second-generation quantum computing control chip

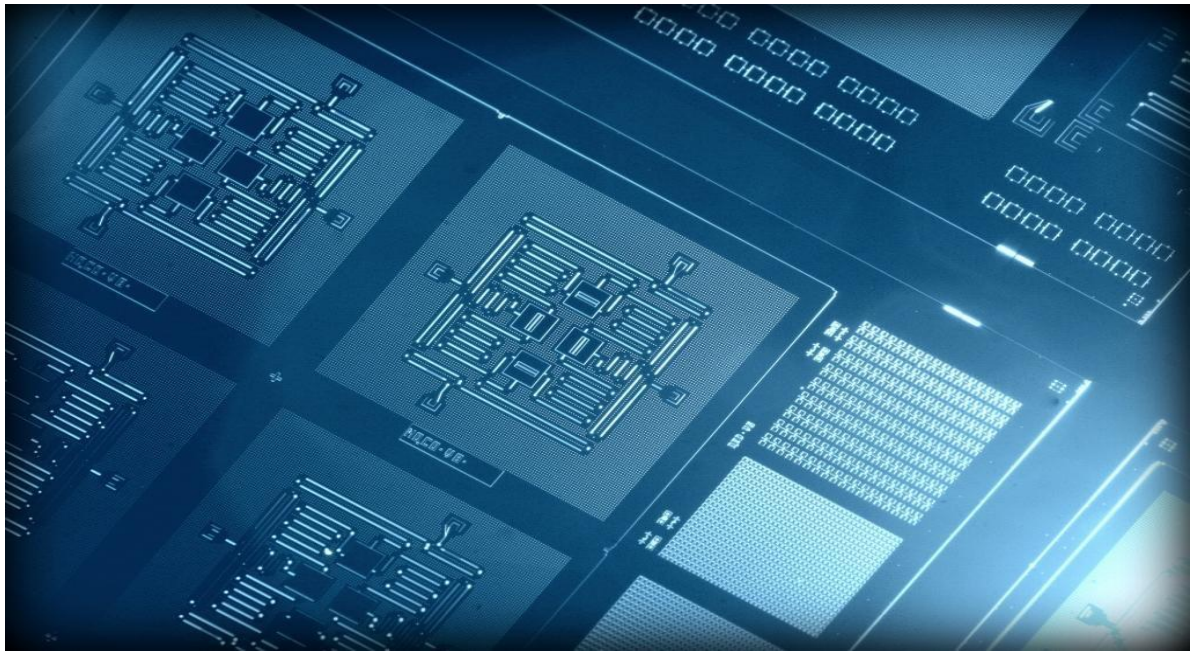
Dean Takahashi @deantak December 3, 2020 12:00 PM **Entrepreneur**



Intel's Horse Ridge II quantum computing chip is hiding under all that refrigeration, packaging, and wiring. Image Credit: Int

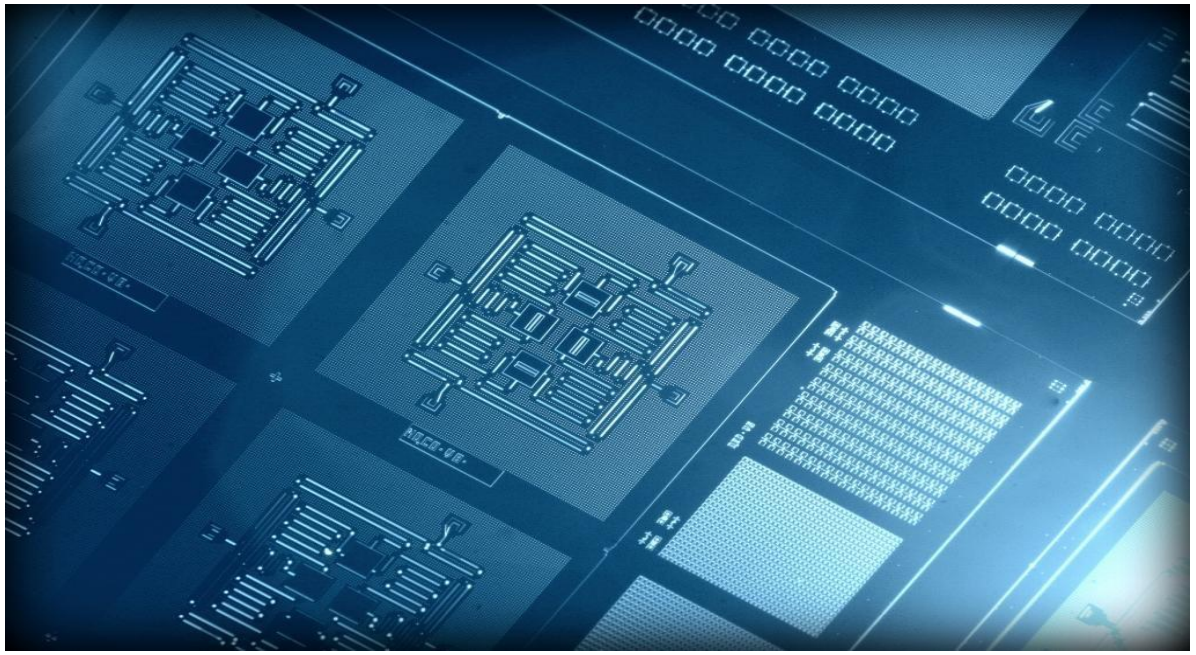
Intel unveiled its second-generation quantum computing control chip during its Intel Labs virtual event today. The chip, codenamed Horse Ridge II, is another milestone toward making quantum computing — one of the holy grails of computing — more practical. The new prototype builds on the first-generation Horse Ridge controller introduced in 2019. Horse Ridge II has more capability and higher levels of integration to control a quantum computer, which remains a long-term goal for the company.

Cryptopocalypse and the NIST response



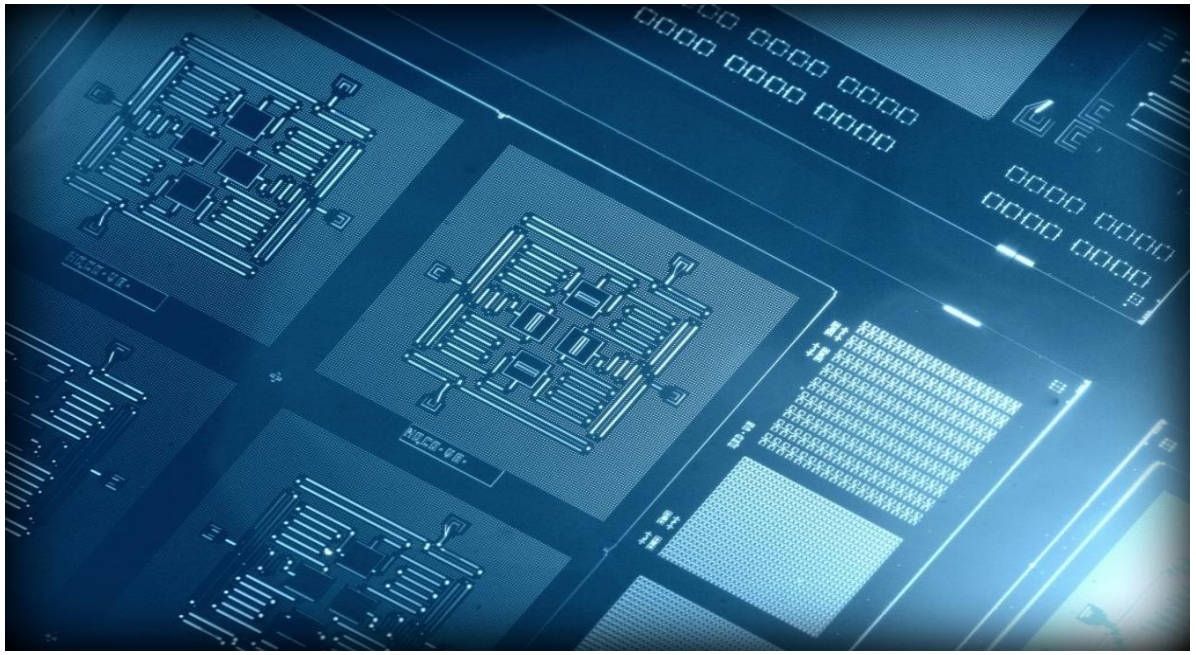
- **Feb 2016:** NIST announces initial plan for standardization of post-quantum algorithms

Cryptopocalypse and the NIST response



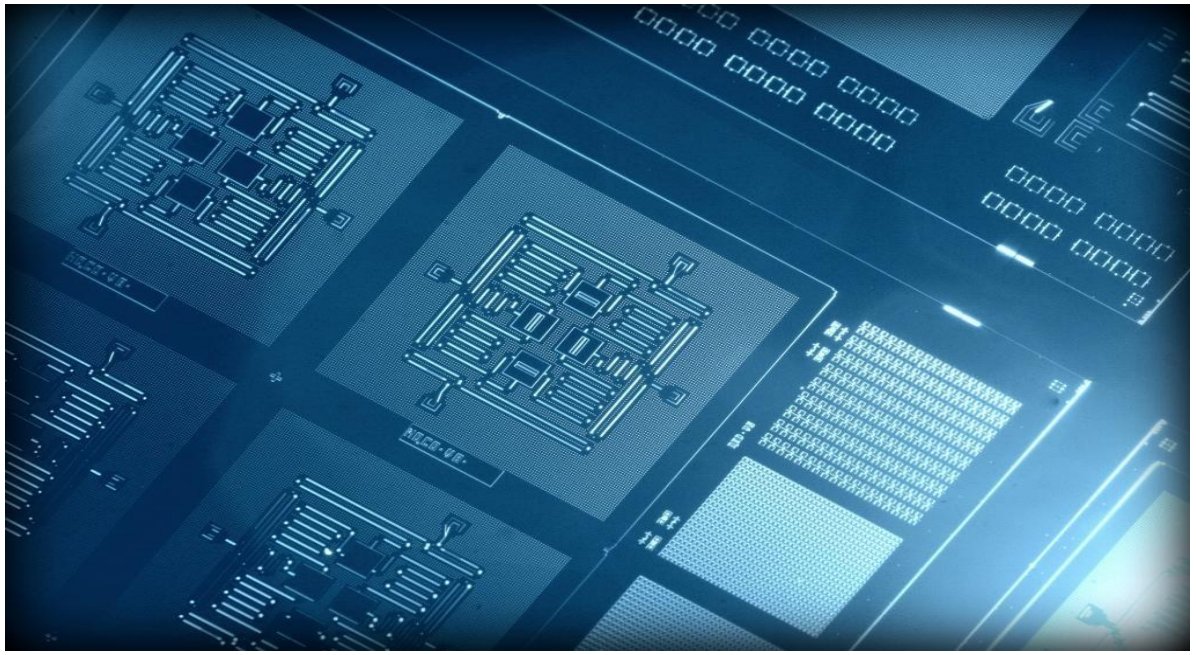
- **Feb 2016:** NIST announces initial plan for standardization of post-quantum algorithms
- **Nov 2017:** 1st round of the NIST "competition" officially starts

Cryptopocalypse and the NIST response



- **Feb 2016:** NIST announces initial plan for standardization of post-quantum algorithms
- **Nov 2017:** 1st round of the NIST "competition" officially starts
- **Jul 2020:** NIST PQC process enters its 3rd round

Cryptopocalypse and the NIST response



- **Feb 2016:** NIST announces initial plan for standardization of post-quantum algorithms
- **Nov 2017:** 1st round of the NIST "competition" officially starts
- **Jul 2020:** NIST PQC process enters its 3rd round

NIST: "The goal of this process is to select a number of acceptable candidate cryptosystems for standardization."
(This includes key encapsulation, encryption and digital signatures).

What do we need to protect *now*?

Assuming a *large-scale, fault tolerant* quantum computer will be developed in, say, 15 years:

- Attacker records encrypted data *today*…

  … uses quantum computer to access secret data *in 15 years* from now.

What do we need to protect *now*?

Assuming a *large-scale, fault tolerant* quantum computer will be developed in, say, 15 years:

- Attacker records encrypted data *today*…

  … uses quantum computer to access secret data *in 15 years* from now.

- Integrity of authentication only matters at the time of connection
  - Keep using classical digital signature schemes for now

What do we need to protect *now*?

Assuming a *large-scale, fault tolerant* quantum computer will be developed in, say, 15 years:

- Attacker records encrypted data *today*…

  … uses quantum computer to access secret data *in 15 years* from now.

- Integrity of authentication only matters at the time of connection
  - Keep using classical digital signature schemes for now

Need quantum-resistant key agreement and encryption for long-term security

Post-quantum (KEM) families

Code-based

Lattice-based

Isogeny-based

Post-quantum (KEM) families

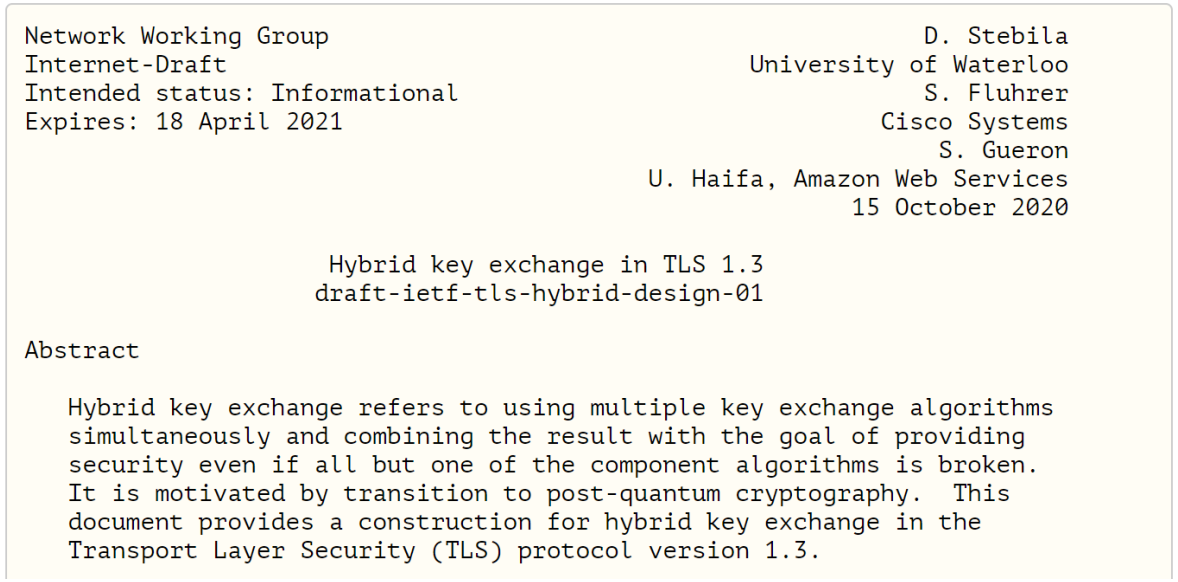
Code-based	**Classic McEliece, BIKE, HQC**
Lattice-based	**NTRU, FrodoKEM (LWE), Kyber (M-LWE), Saber (M-LWR)**
Isogeny-based	**SIKE**

Hybrid protocols

- Post-quantum crypto is still maturing
  - Our confidence on its security and efficiency is growing rapidly but there is still a gap

Hybrid protocols

- Post-quantum crypto is still maturing
  - Our confidence on its security and efficiency is growing rapidly but there is still a gap
- The current recommendation is to use a hybrid approach
  - Combine classical schemes (e.g., ECC) with post-quantum (e.g., SIKE)



Network Working Group — D. Stebila
Internet-Draft — University of Waterloo
Intended status: Informational — S. Fluhrer
Expires: 18 April 2021 — Cisco Systems
S. Gueron
U. Haifa, Amazon Web Services
15 October 2020

Hybrid key exchange in TLS 1.3
draft-ietf-tls-hybrid-design-01

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken. It is motivated by transition to post-quantum cryptography. This document provides a construction for hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3.

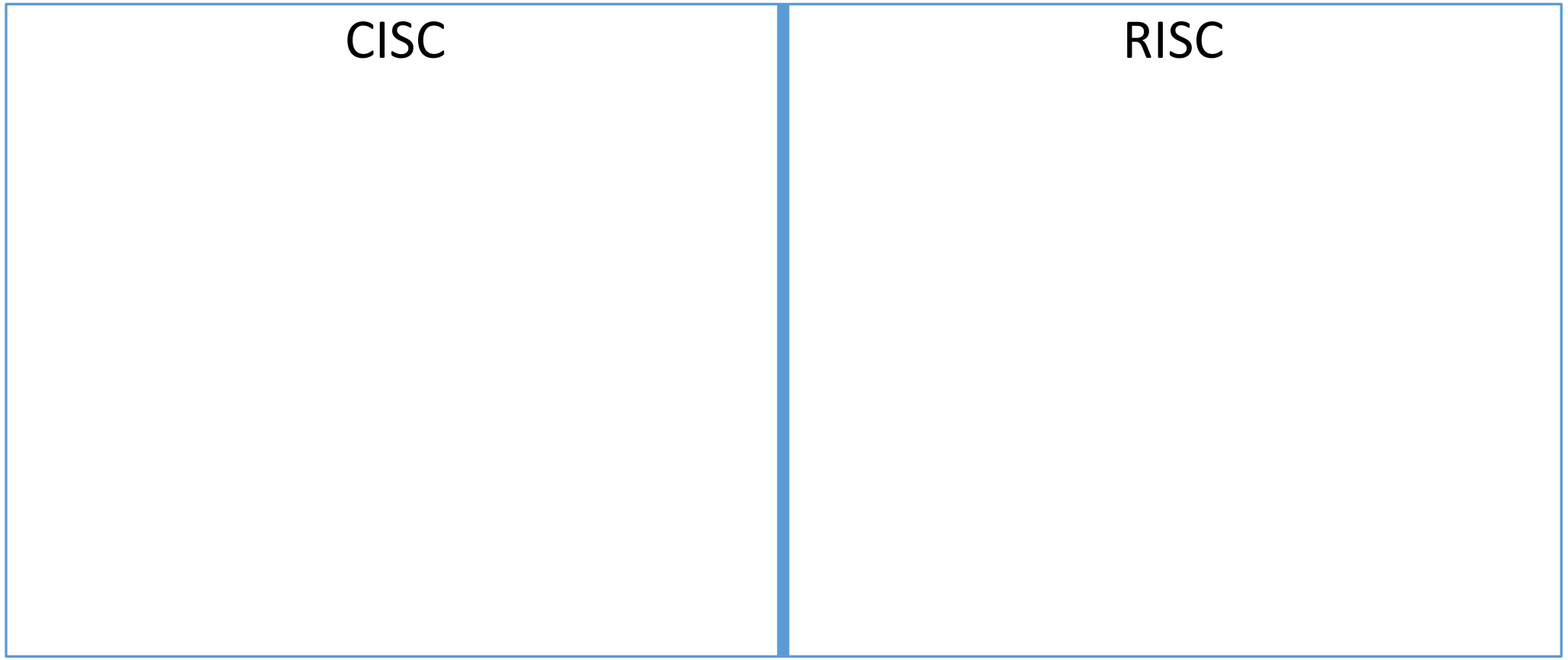
PKC: implementation aspects

Preliminaries

Computer architectures: CISC versus RISC

Computer architectures: CISC versus RISC

Before …



Computer architectures: CISC versus RISC

Before …

CISC	RISC
• Based on complex, multi-clock instructions that do multiple operations • Memory instructions are incorporated in instructions • Smaller code size • Relies on more complex hardware • Less compiler work to translate high-level language to assembly • **Example:** x64 processors	

Computer architectures: CISC versus RISC

Before …

CISC	RISC
• Based on complex, multi-clock instructions that do multiple operations	• Based on simple, single-clock instructions
• Memory instructions are incorporated in instructions	• Memory instructions are independent from instructions
• Smaller code size	• Larger code size
• Relies on more complex hardware	• Relies on simpler hardware
• Less compiler work to translate high-level language to assembly	• More compiler work to translate high-level language to assembly
• **Example:** x64 processors	• **Example:** ARM processors

Computer architectures: CISC versus RISC

Today …

CISC*ish*	RISC*ish*
• Based on complex, multi-clock instructions that do multiple operations	• Based on simple, single-clock instructions *(depends)*
• Memory instructions are incorporated in instructions	• Memory instructions are independent from instructions
• Smaller code size	• Larger code size
• Relies on more complex hardware	• Relies on simpler hardware *(depends)*
• Less compiler work to translate high-level language to assembly	• More compiler work to translate high-level language to assembly
• **Example:** x64 processors	• **Example:** ARM processors

Computer architectures: CISC versus RISC

- Differences between CISC and RISC-based processors have blurred in many aspects
- Modern CPUs (both CISC*ish* or RISC*ish*) incorporate many similar hardware optimizations

Computer architectures: CISC versus RISC

- Differences between CISC and RISC-based processors have blurred in many aspects
- Modern CPUs (both CISC*ish* or RISC*ish*) incorporate many similar hardware optimizations:
  - Pipelining
  - Branch prediction
  - Superscalar execution
  - Out-of-order execution

Computer architectures: CISC versus RISC

- Differences between CISC and RISC-based processors have blurred in many aspects
- Modern CPUs (both CISC*ish* or RISC*ish*) incorporate many similar hardware optimizations:
  - Pipelining
  - Branch prediction
  - Superscalar execution
  - Out-of-order execution

- Software implementations can still take advantage of coarse-grained CISC/RISC differences, but **without losing track of specific architectural characteristics on a CPU-by-CPU basis**

x64 Instruction Set Architecture

x64 Instruction Set Architecture

- Sixteen 64-bit general purpose registers (`rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsp`, `rsi`, `rdi`, `r8-r15`)
- Arithmetic, logical, memory-to-register, and register-to-memory instructions operate over 64-bit integers
- 64-bit virtual address format (2^{64} bytes)
- Fully backward compatible with 16- and 32-bit x86 code

x64 Instruction Set Architecture

- Sixteen 64-bit general purpose registers (`rax`, `rbx`, `rcx`, `rdx`, `rbp`, `rsp`, `rsi`, `rdi`, `r8-r15`)
- Arithmetic, logical, memory-to-register, and register-to-memory instructions operate over 64-bit integers
- 64-bit virtual address format (2^{64} bytes)
- Fully backward compatible with 16- and 32-bit x86 code

- Some relevant instructions:

  `add`, `adc`, `sub`, `sbb`, `shl`, `shr`, `shld`, `shrd`, `btr`, `bts`, `xor`, `and`

  `mov`, `push`, `pop`

  `mul` $(128 \leftarrow 64 \times 64)$

  `mulx` $(128 \leftarrow 64 \times 64)$, does not affect carry flag

PKC: implementation aspects

Side-channel attacks

Side-channel attacks

- There is a plethora of side channels that can be exploited by attackers: timing, power, electromagnetic emanation (EM), etc.

Side-channel attacks

- There is a plethora of side channels that can be exploited by attackers: timing, power, electromagnetic emanation (EM), etc.
- **First line of defense:** ideally, crypto software implementations should run in “constant time”:
  - Program flow (incl. memory accesses) should be independent of secret data
  - This blocks timing and cache attacks

Side-channel attacks

- There is a plethora of side channels that can be exploited by attackers: timing, power, electromagnetic emanation (EM), etc.
- **First line of defense:** ideally, crypto software implementations should run in “constant time”:
  - Program flow (incl. memory accesses) should be independent of secret data
  - This blocks timing and cache attacks
- (Depending on application) software implementations on small devices should also include protection against other attacks (e.g., power and EM attacks)
- We will mainly discuss about the **first line of defense** here

Timing and cache attacks

There are *two* main sources of timing leakage in a computer program:

- Conditional statements (e.g., in if-then-else, do-while, etc.)
- Memory accesses through table lookups

Represent a risk if **execution conditions** or **table lookup indices** are secret-dependent.

Timing and cache attacks

- Conditional statements: example with if-then-else

Timing and cache attacks

- Conditional statements: example with if-then-else

```
if condition then
   x ← do_this
else
   y ← do_that
end
```

Timing and cache attacks

- Conditional statements: example with if-then-else

```
if condition then
   x ← do_this
else
   y ← do_that
end
```

- Even when instr(`do_this`) ≡ instr(`do_that`), it can hold that time(`do_this`) ≠ time(`do_that`)
- This timing difference is mainly due to the **branch predictor**

Timing and cache attacks

- **Countermeasure:** replace conditional statements using masking and logical operations

Timing and cache attacks

- **Countermeasure:** replace conditional statements using masking and logical operations

Example: assume that `condition=1` (TRUE) or `0` (FALSE)

```
mask ← condition-1     (if condition=1 then mask is all 0s, else mask is all 1s)
xx ← do_this
yy ← do_that
x ← (mask & (x XOR xx)) XOR xx
y ← (mask & (y XOR yy)) XOR y
```

Timing and cache attacks

- **Countermeasure:** replace conditional statements using masking and logical operations

Example: assume that `condition=1` (TRUE) or `0` (FALSE)

```
mask ← condition-1     (if condition=1 then mask is all 0s, else mask is all 1s)
xx ← do_this
yy ← do_that
x ← (mask & (x XOR xx)) XOR xx
y ← (mask & (y XOR yy)) XOR y
```

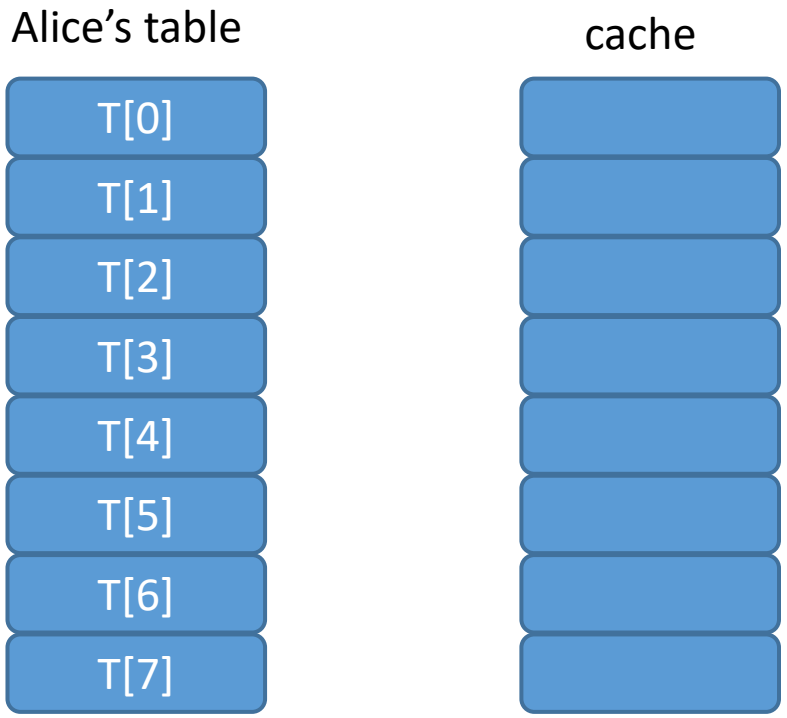
- In some cases, constant-time version can be *faster* than if-then-else version
  - Branch mispredictions can be expensive, especially in architectures with deep pipelines

Timing and cache attacks

- Memory accesses through table lookups

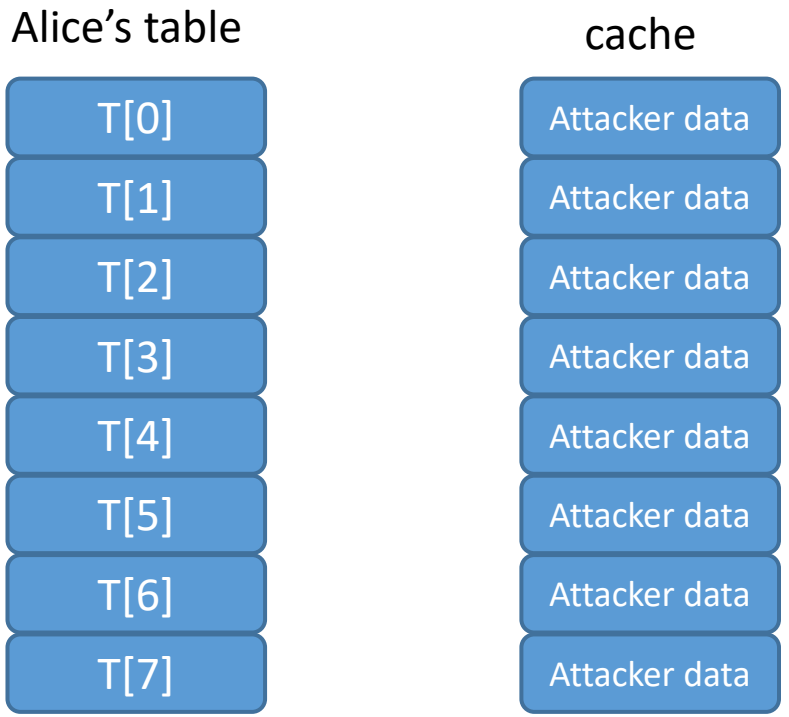
Timing and cache attacks

- Memory accesses through table lookups



Timing and cache attacks

- Memory accesses through table lookups



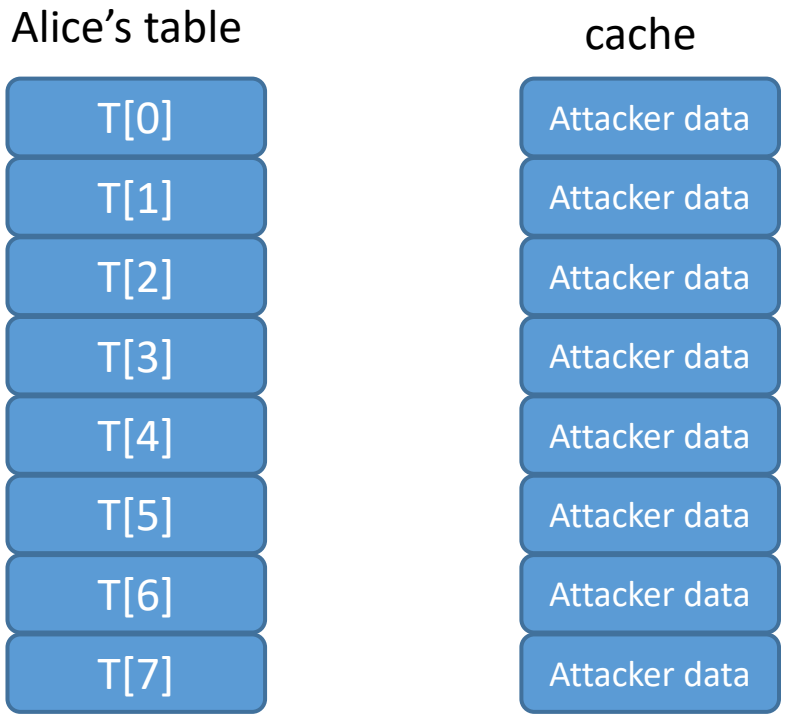
Attacker:

- Flushes and then populates cache memory

Timing and cache attacks

- Memory accesses through table lookups

- Alice needs first entry T[0]. She retrieves it



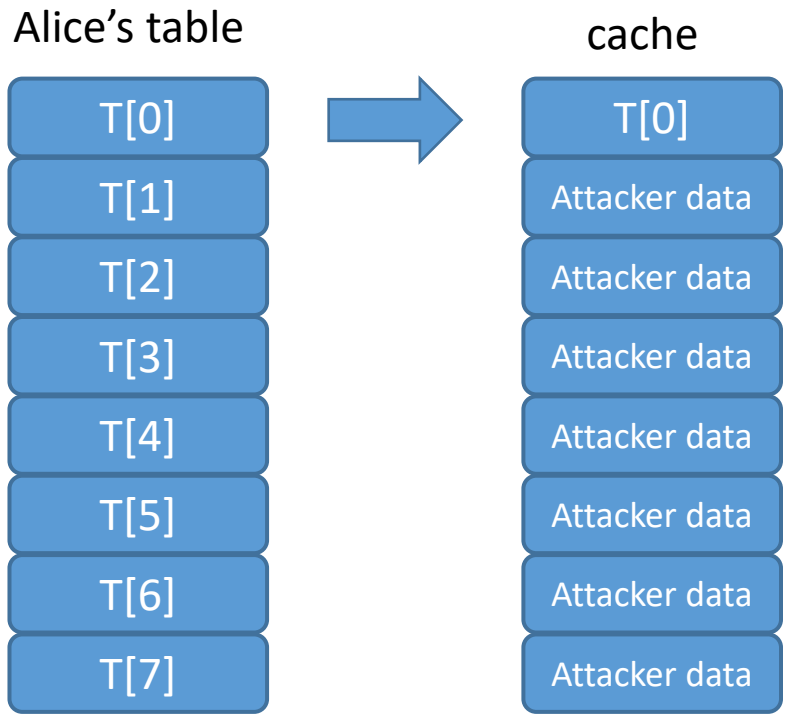
Attacker:

- Flushes and then populates cache memory

Timing and cache attacks

- Memory accesses through table lookups

- Alice needs first entry T[0]. She retrieves it



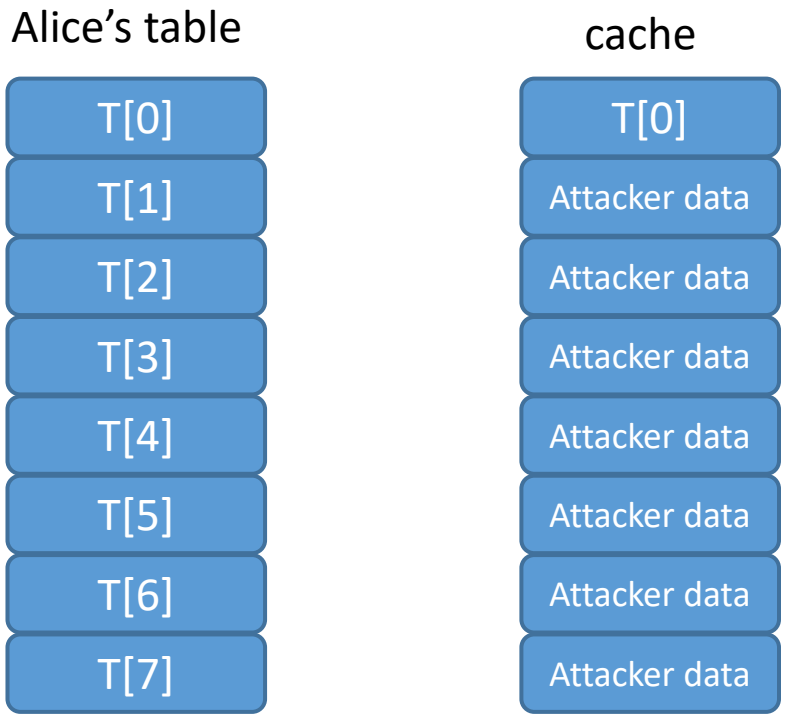
Attacker:

- Flushes and then populates cache memory

Timing and cache attacks

- Memory accesses through table lookups

- Alice needs first entry T[0]. She retrieves it



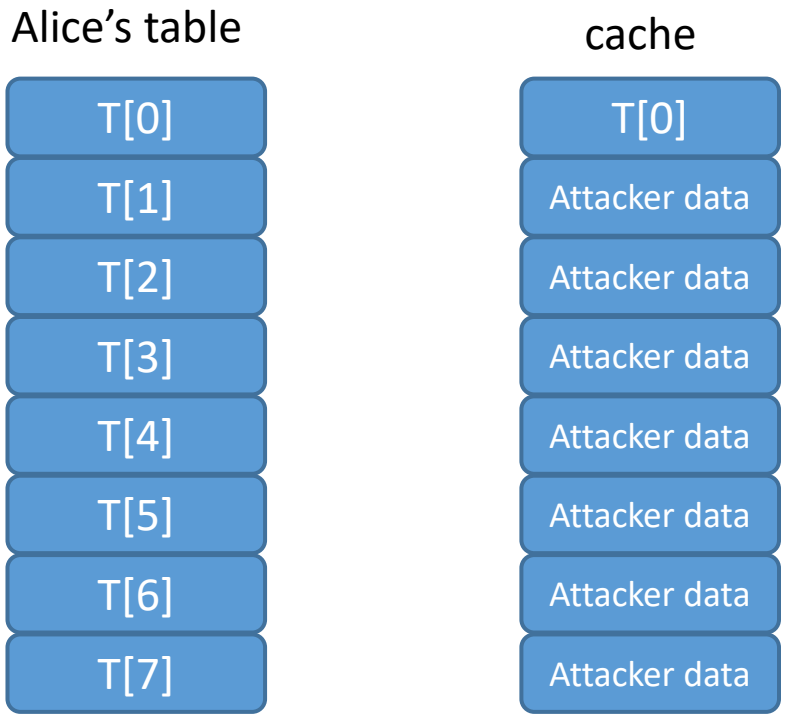
Attacker:

- Flushes and then populates cache memory
- Makes pass over data

Timing and cache attacks

- Memory accesses through table lookups

- Alice needs first entry T[0]. She retrieves it



Attacker:
- Flushes and then populates cache memory
- Makes pass over data

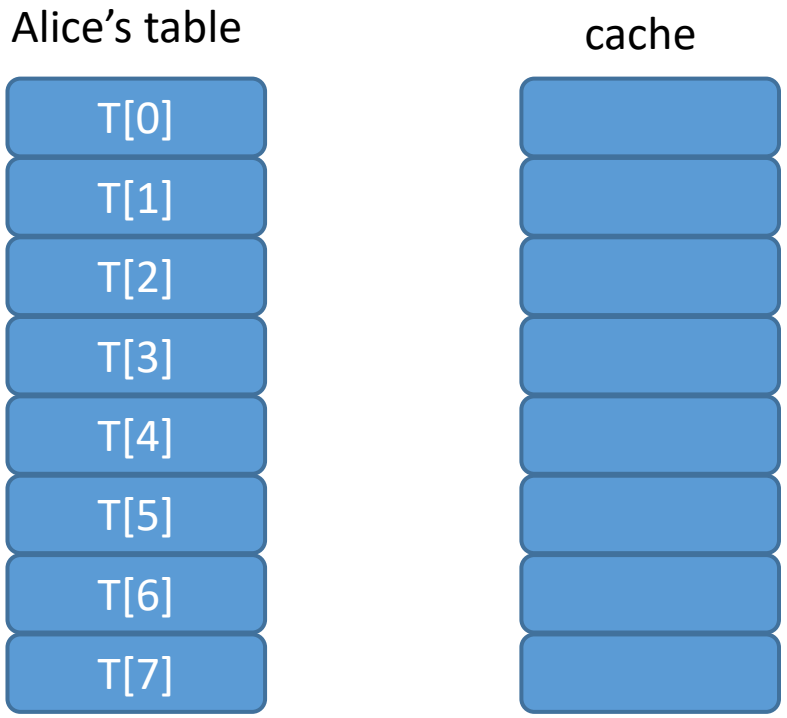
Reading first entry takes longer ➔ learns that Alice retrieved T[0]

Timing and cache attacks

- **Countermeasure:** do linear pass over the full table and retrieve value through masking

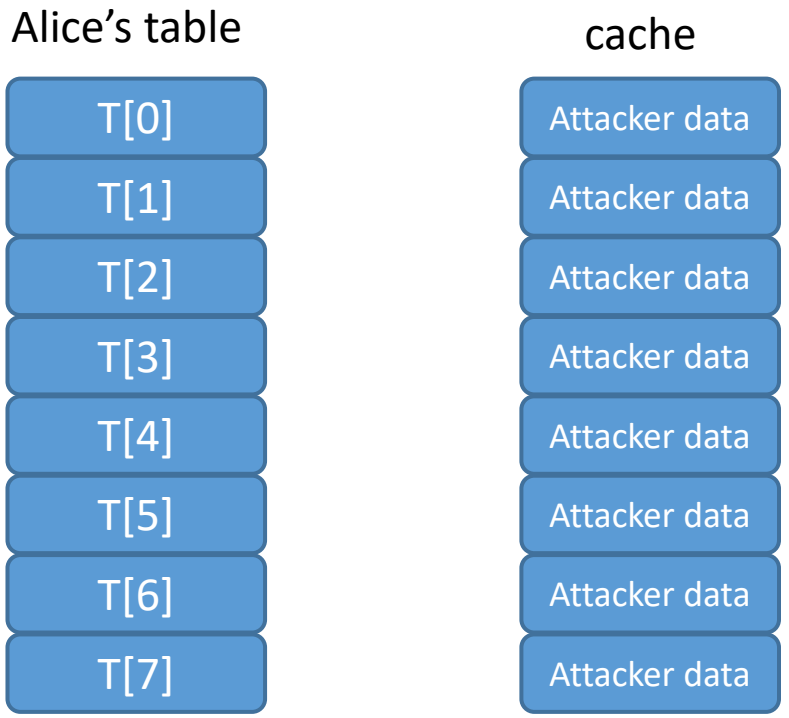
Timing and cache attacks

- **Countermeasure:** do linear pass over the full table and retrieve value through masking



Timing and cache attacks

- **Countermeasure:** do linear pass over the full table and retrieve value through masking



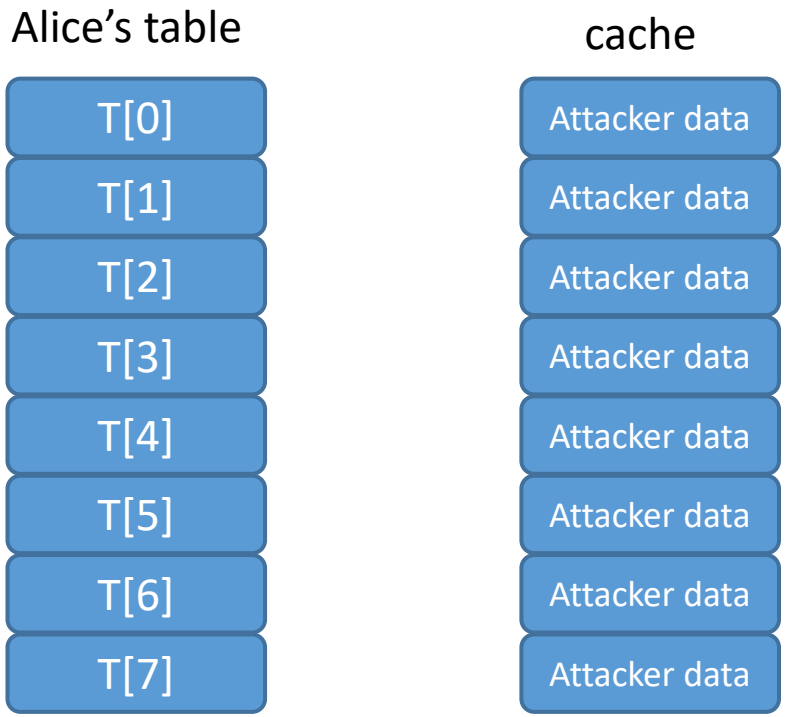
Attacker:

- Flushes and then populates cache memory

Timing and cache attacks

- **Countermeasure:** do linear pass over the full table and retrieve value through masking

- Alice needs first entry T[0]
- Reads T[0]. Applies masking
- Reads T[1]. Applies masking
  ⋮
- Reads T[7]. Applies masking



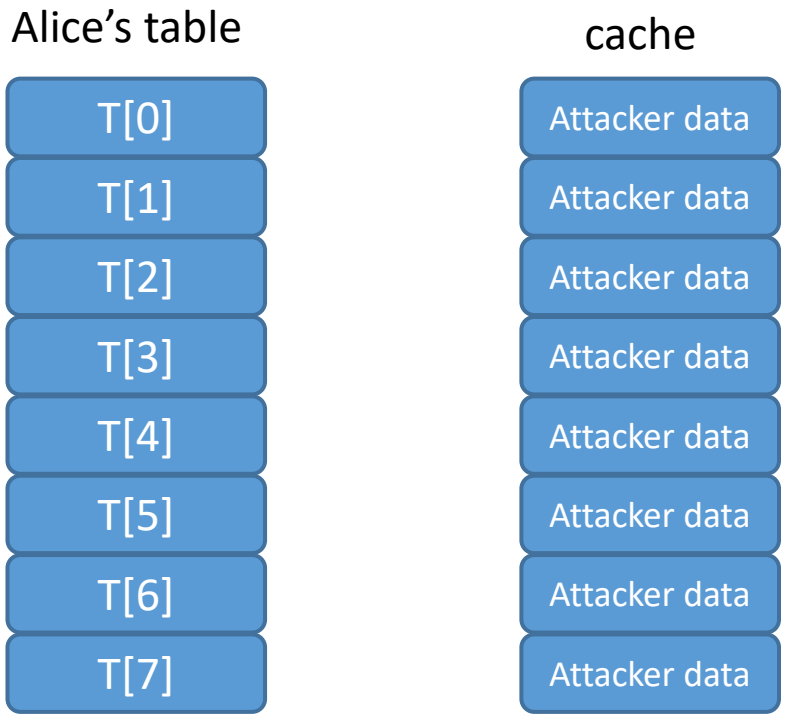
Attacker:

- Flushes and then populates cache memory

Timing and cache attacks

- **Countermeasure:** do linear pass over the full table and retrieve value through masking

- Alice needs first entry T[0]
- Reads T[0]. Applies masking
- Reads T[1]. Applies masking
  ⋮
- Reads T[7]. Applies masking



Attacker:
- Flushes and then populates cache memory

- Whatever the cache contains, it **does not leak information** about the exact value that Alice needed

PKC: implementation aspects

Computer representation

Computer representation

- A t-bit integer a is represented in radix 2^r by

$$a = \sum_{i=0}^{s-1} a_i 2^{ir}$$

with $s = \lceil t/r \rceil$ limbs.

Computer representation

- A t-bit integer a is represented in radix 2^r by

$$a = \sum_{i=0}^{s-1} a_i 2^{ir}$$

  with $s = \lceil t/r \rceil$ limbs.
- Let w be the computer wordsize (i.e., 8, 16, 32 or 64)

Computer representation

- A t-bit integer a is represented in radix 2^r by

$$a = \sum_{i=0}^{s-1} a_i 2^{ir}$$

  with $s = \lceil t/r \rceil$ limbs.
- Let w be the computer wordsize (i.e., 8, 16, 32 or 64)
  - When $r = w$, we call it *full-radix* representation
  - When $r < w$, we call it *reduced-radix* representation

Representation: full-radix vs. reduced-radix

Full-radix: e.g., view of $t = 128$-bit integer in radix 2^{32}

- #limbs $= \lceil t/w \rceil$
- No room for accumulating intermediate values without word spilling

Representation: full-radix vs. reduced-radix

Full-radix: e.g., view of $t = 128$-bit integer in radix 2^{32}

- #limbs $= \lceil t/w \rceil$
- No room for accumulating intermediate values without word spilling

Reduced-radix:

- #limbs $\geq \lceil (t + \delta)/w \rceil$, for some $\delta > 0$
- Extra room for accumulating intermediate values without word spilling
- Requires extra words if $\lceil (t + \delta)/w \rceil > \lceil t/w \rceil$

Representation: full-radix vs. reduced-radix

Full-radix representation:

- Example: a 127-bit integer a is represented by (a_0, a_1, a_2, a_3) in radix 2^{32}
- More amenable for “generic” libraries
- Efficient when instructions with carries are efficient, multiplication is relatively expensive
- Example: non-vector implementations on 64-bit AMD, Intel Atom, Intel Quark, 32-bit ARM

Reduced-radix representation:

Representation: full-radix vs. reduced-radix

Full-radix representation:

- Example: a 127-bit integer a is represented by (a_0, a_1, a_2, a_3) in radix 2^{32}
- More amenable for "generic" libraries
- Efficient when instructions with carries are efficient, multiplication is relatively expensive
- Example: non-vector implementations on 64-bit AMD, Intel Atom, Intel Quark, 32-bit ARM

Reduced-radix representation:

- Example: a 127-bit integer a is represented by $(a_0, a_1, a_2, a_3, a_4)$ in radix 2^{26}
- More amenable for "special purpose" libraries
- Efficient when instructions with carries are relatively expensive, multiplication is efficient
- Example: vector implementations on 64-bit Intel desktop/server and 32-bit ARM,
  (in some cases) non-vector implementations on 64-bit Intel desktop/server
- Convenient when there is no direct access to instructions with carries (e.g., generic C)

Representation: full-radix vs. reduced-radix

- Relative performance between Curve25519-based implementations:
  - **amd64-51** (reduced-radix 2^{51}) and **amd64-64** (full-radix 2^{64})
  - 255-bit finite field defined over $p = 2^{255} - 19$

(*) Source: SUPERCOP (2015)

Representation: full-radix vs. reduced-radix

- Relative performance between Curve25519-based implementations:
  - **amd64-51** (reduced-radix 2^{51}) and **amd64-64** (full-radix 2^{64})
  - 255-bit finite field defined over $p = 2^{255} - 19$
- Results in **BLACK** indicate amd64-51 is better, results in **RED** indicate amd64-64 is better

Intel Haswell (“wintermute”):	**10%**
Intel Ivy Bridge (“hydra8”):	**6%**
Intel Sandy Bridge (“hydra7”):	**5%**
Intel Atom (“h8atom”):	**-36%**
AMD Piledriver (“hydra9”):	**-39%**
AMD Bulldozer (“hydra6”):	**-38%**
AMD Bobcat (“h4e450”):	**-47%**

(*) Source: SUPERCOP (2015)

PKC: implementation aspects

Computing arithmetic

PKC: computational aspects

We will focus on two cases:

- **Classical side:** ECC
- **Post-quantum side:** SIKE

PKC: computational aspects

Computation tree:

- **Underlying arithmetic level**
- **Primitive level**
- **Protocol level**

PKC: computational aspects

Computation tree:

- **Underlying arithmetic level:** e.g., modular arithmetic
- **Primitive level**
- **Protocol level**

PKC: computational aspects

Computation tree:

- **Underlying arithmetic level:** e.g., modular arithmetic
- **Primitive level:** e.g., exponentiation, curve and isogeny arithmetic
- **Protocol level**

PKC: computational aspects

Computation tree:

- **Underlying arithmetic level:** e.g., modular arithmetic
- **Primitive level:** e.g., exponentiation, curve and isogeny arithmetic
- **Protocol level:** e.g., key-exchange, signature, encryption scheme

PKC: computational aspects

Computation tree: ECC

- **Field arithmetic**
- **Curve arithmetic**
- **Protocol level**

PKC: computational aspects

Computation tree: ECC

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_p$ for p prime
- **Curve arithmetic**
- **Protocol level**

PKC: computational aspects

Computation tree: ECC

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_p$ for p prime
- **Curve arithmetic:** point addition/doubling, scalar multiplication
- **Protocol level**

PKC: computational aspects

Computation tree: ECC

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_p$ for p prime
- **Curve arithmetic:** point addition/doubling, scalar multiplication
- **Protocol level:** ECDH key exchange, EC-based signatures (ECDSA, EdDSA)

PKC: computational aspects

Computation tree: SIKE

- **Field arithmetic**
- **Curve and isogeny arithmetic**
- **Protocol level**

PKC: computational aspects

Computation tree: SIKE

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_{p^2}$ for p prime
- **Curve and isogeny arithmetic**
- **Protocol level**

PKC: computational aspects

Computation tree: SIKE

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_{p^2}$ for p prime
- **Curve and isogeny arithmetic:** point addition/doubling/tripling, scalar multiplication, isogeny evaluation and computation, isogeny-tree traversal
- **Protocol level**

PKC: computational aspects

Computation tree: SIKE

- **Field arithmetic:** multiplication, squaring, addition, inversion over $\mathbb{F}_{p^2}$ for p prime
- **Curve and isogeny arithmetic:** point addition/doubling/tripling, scalar multiplication, isogeny evaluation and computation, isogeny-tree traversal
- **Protocol level:** key exchange (SIDH), IND-CCA2-secure key encapsulation (SIKE)

PKC: implementation aspects

Modular arithmetic

Modular arithmetic

- It is at the heart of most widely-used public-key cryptographic primitives

Modular multiplication

- It is at the heart of most widely-used public-key cryptographic primitives
- Typically, the most time-consuming part is **modular multiplication**: $a \times b \mod p$
  Integer multiplication + modular reduction

Modular multiplication

- It is at the heart of most widely-used public-key cryptographic primitives
- Typically, the most time-consuming part is **modular multiplication**: $a \times b \mod p$
  Integer multiplication + modular reduction
- Simplest approaches to integer multiplication:
  Schoolbook (operand scanning form)
  Comba (product scanning form)

Modular multiplication

- It is at the heart of most widely-used public-key cryptographic primitives
- Typically, the most time-consuming part is **modular multiplication**: $a \times b \mod p$
  Integer multiplication + modular reduction
- Simplest approaches to integer multiplication:
  Schoolbook (operand scanning form)
  Comba (product scanning form)
- These methods have quadratic complexity: n^2 multiplications
  - They work well for a small number of limbs: **usually optimal for the sizes managed in 256-bit ECC on 32-bit and 64-bit platforms**

Modular multiplication

- It is at the heart of most widely-used public-key cryptographic primitives
- Typically, the most time-consuming part is **modular multiplication**: $a \times b \mod p$
  Integer multiplication + modular reduction
- Simplest approaches to integer multiplication:
  Schoolbook (operand scanning form)
  Comba (product scanning form)
- These methods have quadratic complexity: n^2 multiplications
  - They work well for a small number of limbs: **usually optimal for the sizes managed in 256-bit ECC on 32-bit and 64-bit platforms**
- There are other methods with subquadratic complexity:
  Karatsuba method: asymptotic complexity $O(n^{\log_2 3})$
  Toom-Cook multiplication: asymptotic complexity $O(n^{\log_2 5})$

Modular multiplication

- It is at the heart of most widely-used public-key cryptographic primitives
- Typically, the most time-consuming part is **modular multiplication**: $a \times b \mod p$
  Integer multiplication + modular reduction
- Simplest approaches to integer multiplication:
  Schoolbook (operand scanning form)
  Comba (product scanning form)
- These methods have quadratic complexity: n^2 multiplications
  - They work well for a small number of limbs: **usually optimal for the sizes managed in 256-bit ECC on 32-bit and 64-bit platforms**
- There are other methods with subquadratic complexity:
  Karatsuba method: asymptotic complexity $O(n^{\log_2 3})$
  Toom-Cook multiplication: asymptotic complexity $O(n^{\log_2 5})$
- **Karatsuba becomes attractive for a relatively large number of limbs:** in some cases, useful for high-security ECC (e.g., NIST P-521)

Modular multiplication

- Many hybrid methods combining schoolbook/Comba/Karatsuba

Modular multiplication

- Many hybrid methods combining schoolbook/Comba/Karatsuba
- *Two* generic flavours:
  - **Interleaved multiplication/reduction:** usually faster when standalone
  - **Non-interleaved multiplication/reduction:** usually faster when lazy reduction applies

Modular reduction

- Two big categories for modular reduction:
  General-form modulus (RSA, ECC over general primes: e.g., Brainpool curves)
  Special-form modulus (high-performance ECC: e.g., NIST P-256, Curve25519, Four$\mathbb{Q}$)
- Two main reduction methods: Barrett reduction and Montgomery reduction

Modular reduction: special form primes

- Most popular options (for high-performance ECC):

Modular reduction: special form primes

- Most popular options (for high-performance ECC):
  - **Generalized pseudo-Mersenne primes** (a.k.a. Solinas primes)
    E.g., NIST P-256 prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$,
    Ed448-Goldilocks's prime $p = 2^{448} - 2^{224} - 1$

Modular reduction: special form primes

- Most popular options (for high-performance ECC):
  - **Generalized pseudo-Mersenne primes** (a.k.a. Solinas primes)
    E.g., NIST P-256 prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$,
    Ed448-Goldilocks's prime $p = 2^{448} - 2^{224} - 1$
  - **Pseudo-Mersenne primes** (a.k.a. Crandall primes)
    E.g., Curve25519's prime $p = 2^{255} - 19$

Modular reduction: special form primes

- Most popular options (for high-performance ECC):
  - **Generalized pseudo-Mersenne primes** (a.k.a. Solinas primes)
    E.g., NIST P-256 prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$,
    Ed448-Goldilocks's prime $p = 2^{448} - 2^{224} - 1$
  - **Pseudo-Mersenne primes** (a.k.a. Crandall primes)
    E.g., Curve25519's prime $p = 2^{255} - 19$
  - **Mersenne primes**
    E.g., Four$\mathbb{Q}$'s prime $p = 2^{127} - 1$,
    NIST P-521's prime $p = 2^{521} - 1$

Modular reduction: special form primes

- Most popular options (for high-performance ECC):
  - **Generalized pseudo-Mersenne primes** (a.k.a. Solinas primes)
    E.g., NIST P-256 prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$,
    Ed448-Goldilocks's prime $p = 2^{448} - 2^{224} - 1$
  - **Pseudo-Mersenne primes** (a.k.a. Crandall primes)
    E.g., Curve25519's prime $p = 2^{255} - 19$
  - **Mersenne primes**
    E.g., Four$\mathbb{Q}$'s prime $p = 2^{127} - 1$,
    NIST P-521's prime $p = 2^{521} - 1$
- NIST prime choices targeted 32-bit platforms (popular at that time). Except for P-521, these primes are relatively expensive in constant-time and vectorized implementations

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$
\begin{aligned}
[t_0, t] &= a_0 \times b_0 \\
[t_1, t] &= a_0 \times b_1 + a_1 \times b_0 + t \\
[t_2, t] &= a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t \\
[t_3, t] &= a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t \\
[t_4, t] &= a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t \\
[t_5, t] &= a_2 \times b_3 + a_3 \times b_2 + t \\
[t_6, t_7] &= a_3 \times b_3 + t
\end{aligned}
$$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$[t_0, t] = a_0 \times b_0$$
$$[t_1, t] = a_0 \times b_1 + a_1 \times b_0 + t$$
$$[t_2, t] = a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t$$
$$[t_3, t] = a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t$$
$$[t_4, t] = a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t$$
$$[t_5, t] = a_2 \times b_3 + a_3 \times b_2 + t$$
$$[t_6, t_7] = a_3 \times b_3 + t$$

integer mult result $\{t_0 - t_7\}$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$[t_0, t] = a_0 \times b_0$$
$$[t_1, t] = a_0 \times b_1 + a_1 \times b_0 + t$$
$$[t_2, t] = a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t$$
$$[t_3, t] = a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t$$
$$[t_4, t] = a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t$$
$$[t_5, t] = a_2 \times b_3 + a_3 \times b_2 + t$$
$$[t_6, t_7] = a_3 \times b_3 + t$$

integer mult result $\{t_0 - t_7\}$

Reduction (using $2^{256} \equiv 38$)

- Set output in the range $[0, 2^{256} - 38)$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$[t_0, t] = a_0 \times b_0$$
$$[t_1, t] = a_0 \times b_1 + a_1 \times b_0 + t$$
$$[t_2, t] = a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t$$
$$[t_3, t] = a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t$$
$$[t_4, t] = a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t$$
$$[t_5, t] = a_2 \times b_3 + a_3 \times b_2 + t$$
$$[t_6, t_7] = a_3 \times b_3 + t$$

integer mult result $\{t_0 - t_7\}$

Reduction (using $2^{256} \equiv 38$)

1st pass

$$[t_0, t] = t_0 + 38 \times t_4$$
$$[t_1, t] = t_1 + 38 \times t_5 + t$$
$$[t_2, t] = t_2 + 38 \times t_6 + t$$
$$[t_3, t] = t_3 + 38 \times t_7 + t$$

- Set output in the range $[0, 2^{256} - 38)$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$
\begin{aligned}
[t_0, t] &= a_0 \times b_0 \\
[t_1, t] &= a_0 \times b_1 + a_1 \times b_0 + t \\
[t_2, t] &= a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t \\
[t_3, t] &= a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t \\
[t_4, t] &= a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t \\
[t_5, t] &= a_2 \times b_3 + a_3 \times b_2 + t \\
[t_6, t_7] &= a_3 \times b_3 + t
\end{aligned}
$$

integer mult result $\{t_0 - t_7\}$

Reduction (using $2^{256} \equiv 38$)

$$
\begin{aligned}
[t_0, t] &= t_0 + 38 \times t_4 \\
[t_1, t] &= t_1 + 38 \times t_5 + t \\
[t_2, t] &= t_2 + 38 \times t_6 + t \\
[t_3, t] &= t_3 + 38 \times t_7 + t \\
[t_0, c] &= t_0 + 38 \times (t + 1) \\
[t_1, c] &= t_1 + c \\
[t_2, c] &= t_2 + c \\
[t_3, c] &= t_3 + c
\end{aligned}
$$

1st pass

2nd pass
Add extra 38

- Set output in the range $[0, 2^{256} - 38)$

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (64-bit)
- Inputs $[a_0, a_1, a_2, a_3]$ and $[b_0, b_1, b_2, b_3]$

integer mult

$$
\begin{aligned}
[t_0, t] &= a_0 \times b_0 \\
[t_1, t] &= a_0 \times b_1 + a_1 \times b_0 + t \\
[t_2, t] &= a_0 \times b_2 + a_2 \times b_0 + a_1 \times b_1 + t \\
[t_3, t] &= a_0 \times b_3 + a_3 \times b_0 + a_1 \times b_2 + a_2 \times b_1 + t \\
[t_4, t] &= a_1 \times b_3 + a_3 \times b_1 + a_2 \times b_2 + t \\
[t_5, t] &= a_2 \times b_3 + a_3 \times b_2 + t \\
[t_6, t_7] &= a_3 \times b_3 + t
\end{aligned}
$$

integer mult result $\{t_0 - t_7\}$

Reduction (using $2^{256} \equiv 38$)

$$
\begin{aligned}
[t_0, t] &= t_0 + 38 \times t_4 \\
[t_1, t] &= t_1 + 38 \times t_5 + t \\
[t_2, t] &= t_2 + 38 \times t_6 + t \\
[t_3, t] &= t_3 + 38 \times t_7 + t \\
[t_0, c] &= t_0 + 38 \times (t + 1) \\
[t_1, c] &= t_1 + c \\
[t_2, c] &= t_2 + c \\
[t_3, c] &= t_3 + c \\
[t_0, c] &= t_0 - 38 \times c \text{ if } c = 1 \\
[t_1, c] &= t_1 - c \\
[t_2, c] &= t_2 - c \\
[t_3, c] &= t_3 - c
\end{aligned}
$$

Output $[t_0, t_1, t_2, t_3]$

1st pass

2nd pass
Add extra 38

3rd pass with condition

- t holds 2×64 bits.
- t_x hold 64 bits.
- c is a carry or borrow bit.

- Set output in the range $[0, 2^{256} - 38)$

Modular reduction: special form primes

- *Constant-time* field multiplication with $p = 2^{255} - 19$ (x64 assembly)

```
mov  rax, [reg_p1]
mul qword ptr [reg_p2]     // a0*b0
mov  r13, rax              // t0
mov  r12, rdx

mov  rbx, [reg_p2+8]
mov  rax, [reg_p1]
mul  rbx                   // a0*b1
add  r12, rax
mov  r10, rdx
adc  r10, 0

xor  r11, r11
mov  rax, [reg_p1+8]       // a1*b0
mul qword ptr [reg_p2]
add  r12, rax
mov  [reg_p3+8], r12       // t1
adc  r10, rdx
adc  r11, 0

mov  rax, [reg_p2]
mul qword ptr [reg_p1+16] // a2*b0
add  r10, rax
adc  r11, rdx

xor  rcx, rcx
mov  rax, [reg_p1+8]
mul  rbx                   // a1*b1
add  r10, rax
adc  r11, rdx
adc  rcx, 0

mov  rax, [reg_p2+16]
mul qword ptr [reg_p1]     // a0*b2
add  r10, rax
mov  [reg_p3+16], r10      // t2
adc  r11, rdx
adc  rcx, 0
```

```
xor  r10, r10
mov  rax, [reg_p2]
mul qword ptr [reg_p1+24] // a3*b0
add  r11, rax
adc  rcx, rdx
adc  r10, 0

mov  rax, [reg_p2+8]
mul qword ptr [reg_p1+16] // a2*b1
add  r11, rax
adc  rcx, rdx
adc  r10, 0

mov  rax,[reg_p2+16]
mul  rbx                   // a1*b2
add  r11, rax
adc  rcx, rdx
adc  r10, 0

mov  rax, [reg_p2+24]
mul qword ptr [reg_p1]     // a0*b3
add  r11, rax
mov  [reg_p3+24], r11      // t3
adc  rcx, rdx
adc  r10, 0

xor  r11, r11
mov  rax, [reg_p2+8]
mul qword ptr [reg_p1+24] // a3*b1
add  rcx, rax
adc  r10, rdx
adc  r11, 0

mov  rax, [reg_p2+16]
mul qword ptr [reg_p1+16] // a2*b2
add  rcx, rax
adc  r10, rdx
adc  r11, 0
```

```
mov  rax, [reg_p2+24]
mul  rbx                   // a1*b3
add  rcx, rax              // t4
adc  r10, rdx
adc  r11, 0

xor  r8, r8
mov  rax, [reg_p2+16]
mul qword ptr [reg_p1+24] // a3*b2
add  r10, rax
adc  r11, rdx
adc  r8, 0

mov  rax, [reg_p2+24]
mul qword ptr [reg_p1+16] // a2*b3
add  r10, rax              // t5
adc  r11, rdx
adc  r8, 0

mov  rax, [reg_p2+24]
mul qword ptr [reg_p1+24] // a3*b3
add  r11, rax              // t6
adc  r8, rdx               // t7

// Reduction

mov  rax, constant_38
mul  rcx
add  r13, rax              // r13 <- t0
adc  rdx, 0
mov  rcx, rdx

xor  r12, r12
mov  rax, constant_38
mul  r10
add  rax, rcx
adc  r12, 0
mov  r10, [reg_p3+8]
add  r10, rax              // r10 <- t1
adc  r12, rdx
```

```
xor  rcx, rcx
mov  rax, constant_38
mul  r11
add  rax, r12
adc  rcx, 0
mov  r11, [reg_p3+16]
add  r11, rax              // r11 <- t2
adc  rcx, rdx

xor  r12, r12
mov  rax, constant_38
mul  r8
add  rax, rcx
adc  r12, 1
mov  r8, [reg_p3+24]
add  r8, rax               // r8 <- t3
adc  rdx, r12              // rdx <- t4 + 1

xor  r12, r12
mov  rax, constant_38
mul  rdx
add  r13, rax              // t0
adc  r10, 0                // t1
adc  r11, 0                // t2
adc  r8, 0                 // t3

mov  rax, constant_38      // final correction
cmovc rax, r12
sub  r13, rax              // t0
mov  [reg_p3], r13
sbb  r10, 0                // t1
mov  [reg_p3+8], r10
sbb  r11, 0                // t2
mov  [reg_p3+16], r11
sbb  r8, 0                 // t3
mov  [reg_p3+24], r8
```

Modular addition

- *Constant-time* field addition $a + b \bmod p$
  - If $r = a + b \geq p$ then $r = r - p$. Output r
  - Eliminating **if-then-else statement** is usually straightforward

Modular addition

- *Constant-time* field addition $a + b \bmod p$
  - If $r = a + b \geq p$ then $r = r - p$. Output r
  - Eliminating **if-then-else statement** is usually straightforward

Example: field addition with $p = 2^{127} - 1$ (64-bit)

$[t_0, c] = a_0 + b_0$
$t_1 = a_1 + b_1 + c$

Write $c = t_1 \gg 63,\ t_1 = t_1 \,\&\, \text{mask\_63}$ (using $2^{127} \equiv 1$)
$[t_0, c] = t_0 + c$
$t_1 = t_1 + c$
Output $[t_0, t_1]$

Modular addition

- **Example:** *constant-time* field addition with $p = 2^{127} - 1$ (x64 assembly)

```
mov     r8, [reg_p1]
mov     r9, [reg_p1+8]
add     r8, [reg_p2]
adc     r9, [reg_p2+8]

btr     r9, 63
adc     r8, 0
mov     [reg_p3], r8
adc     r9, 0
mov     [reg_p3+8], r9
```

Modular addition

- **Example:** *constant-time* field addition with $p = 2^{127} - 1$ (x64 assembly)

```
mov     r8, [reg_p1]
mov     r9, [reg_p1+8]
add     r8, [reg_p2]
adc     r9, [reg_p2+8]

btr     r9, 63
adc     r8, 0
mov     [reg_p3], r8
adc     r9, 0
mov     [reg_p3+8], r9
```

- **Example:** *constant-time* field subtraction with $p = 2^{127} - 1$ (x64 assembly)

```
mov     r8, [reg_p1]
mov     r9, [reg_p1+8]
sub     r8, [reg_p2]
sbb     r9, [reg_p2+8]

btr     r9, 63
sbb     r8, 0
mov     [reg_p3], r8
sbb     r9, 0
mov     [reg_p3+8], r9
```

Modular arithmetic: case with reduced-radix

- So far, we have explored operations using full-radix. What about **reduced-radix**?
- Extra room in each limb can be used to accumulate carries

Modular arithmetic: case with reduced-radix

- So far, we have explored operations using full-radix. What about **reduced-radix**?
- Extra room in each limb can be used to accumulate carries

Example: Set $w = 64$. An element in $\mathbb{F}_{2^{127}-1}$ can be represented as (a_0, a_1, a_2) in radix 2^{43}. When fully reduced, element values are [43, 43, 41]-bit long

Modular arithmetic: case with reduced-radix

- So far, we have explored operations using full-radix. What about **reduced-radix**?
- Extra room in each limb can be used to accumulate carries

Example: Set $w = 64$. An element in $\mathbb{F}_{2^{127}-1}$ can be represented as (a_0, a_1, a_2) in radix 2^{43}. When fully reduced, element values are [43, 43, 41]-bit long

127-bit field addition

$t_0 = a_0 + b_0$
$t_1 = a_1 + b_1$
$t_2 = a_2 + b_2$

Modular arithmetic: case with reduced-radix

- So far, we have explored operations using full-radix. What about **reduced-radix**?
- Extra room in each limb can be used to accumulate carries

Example: Set $w = 64$. An element in $\mathbb{F}_{2^{127}-1}$ can be represented as (a_0, a_1, a_2) in radix 2^{43}. When fully reduced, element values are [43, 43, 41]-bit long

127-bit field addition

$$t_0 = a_0 + b_0$$
$$t_1 = a_1 + b_1$$
$$t_2 = a_2 + b_2$$

With full-radix:

$$[t_0, c] = a_0 + b_0$$
$$t_1 = a_1 + b_1 + c$$
$$c = t_1 \gg 63,\ t_1 \,\&=\, \text{mask\_63}$$
$$[t_0, c] = t_0 + c$$
$$t_1 = t_1 + c$$

Modular arithmetic: case with reduced-radix

- So far, we have explored operations using full-radix. What about **reduced-radix**?
- Extra room in each limb can be used to accumulate carries

Example: Set $w = 64$. An element in $\mathbb{F}_{2^{127}-1}$ can be represented as (a_0, a_1, a_2) in radix 2^{43}. When fully reduced, element values are [43, 43, 41]-bit long

127-bit field addition

$$t_0 = a_0 + b_0$$
$$t_1 = a_1 + b_1$$
$$t_2 = a_2 + b_2$$

127-bit field subtraction

$$t_0 = a_0 - b_0$$
$$t_1 = a_1 - b_1$$
$$t_2 = a_2 - b_2$$

With full-radix:

$$[t_0, c] = a_0 + b_0$$
$$t_1 = a_1 + b_1 + c$$
$$c = t_1 \gg 63,\ t_1\ \&=\ \text{mask\_63}$$
$$[t_0, c] = t_0 + c$$
$$t_1 = t_1 + c$$

Modular arithmetic: case with reduced-radix

- Field addition and subtraction are now a *vector addition* or *vector subtraction*
- We can safely perform many adds/subs without producing an overflow

Modular arithmetic: case with reduced-radix

- Field addition and subtraction are now a *vector addition* or *vector subtraction*
- We can safely perform many adds/subs without producing an overflow
- Eventually, we need to do a ***carry correction*** (e.g., after multiplications)

Modular arithmetic: case with reduced-radix

- Field addition and subtraction are now a *vector addition* or *vector subtraction*
- We can safely perform many adds/subs without producing an overflow
- Eventually, we need to do a ***carry correction*** (e.g., after multiplications)

After result of additions/subtractions in $\mathbb{F}_{2^{127}-1}$:

$t_0 = t_0 + (t_2 \gg 41)$
$t_2 = t_2 \,\&\, \text{mask\_}41$
$t_1 = t_1 + (t_0 \gg 43)$
$t_0 = t_0 \,\&\, \text{mask\_}43$
$t_2 = t_2 + (t_1 \gg 43)$
$t_1 = t_1 \,\&\, \text{mask\_}43$

Modular arithmetic: case with reduced-radix

- Field addition and subtraction are now a *vector addition* or *vector subtraction*
- We can safely perform many adds/subs without producing an overflow
- Eventually, we need to do a ***carry correction*** (e.g., after multiplications)

After result of additions/subtractions in $\mathbb{F}_{2^{127}-1}$:

$t_0 = t_0 + (t_2 \gg 41)$
$t_2 = t_2 \,\&\, \text{mask\_41}$
$t_1 = t_1 + (t_0 \gg 43)$
$t_0 = t_0 \,\&\, \text{mask\_43}$
$t_2 = t_2 + (t_1 \gg 43)$
$t_1 = t_1 \,\&\, \text{mask\_43}$

- A reduced representation [43, 43, 41]-bit is obtained by running **two rounds** of the carry correction routine
- In some cases, it might be enough to run **one round** (after multiplications and squarings)

Modular arithmetic: case with reduced-radix

- For our example, if there is access to `adc` then **full-radix is more efficient than reduced-radix** (e.g., when using assembly or intrinsics)

Modular arithmetic: case with reduced-radix

- For our example, if there is access to `adc` then **full-radix is more efficient than reduced-radix** (e.g., when using assembly or intrinsics)
- Otherwise, reduced radix can be more efficient (e.g., when using high-level languages)

Modular arithmetic: case with reduced-radix

- For our example, if there is access to `adc` then **full-radix is more efficient than reduced-radix** (e.g., when using assembly or intrinsics)
- Otherwise, reduced radix can be more efficient (e.g., when using high-level languages)
- When using a high-level language, **reduced-radix can be simpler to implement in some cases**

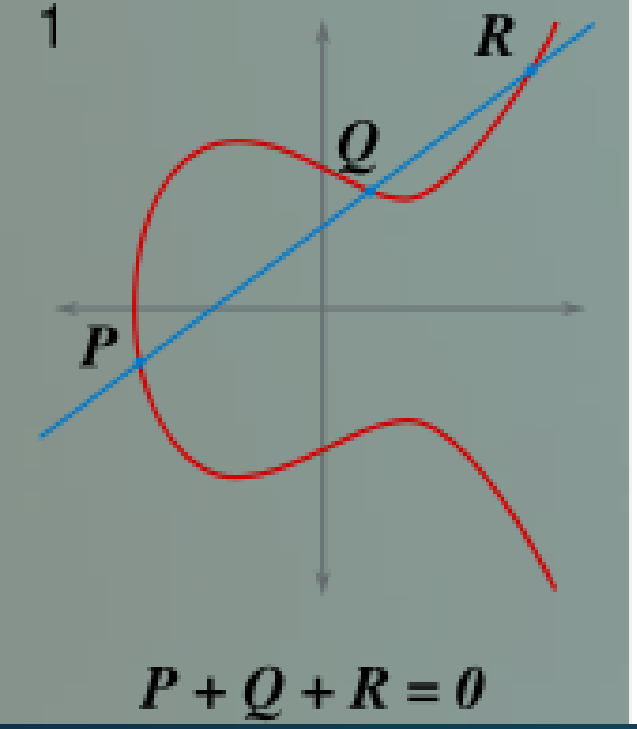
Modular arithmetic: case with reduced-radix

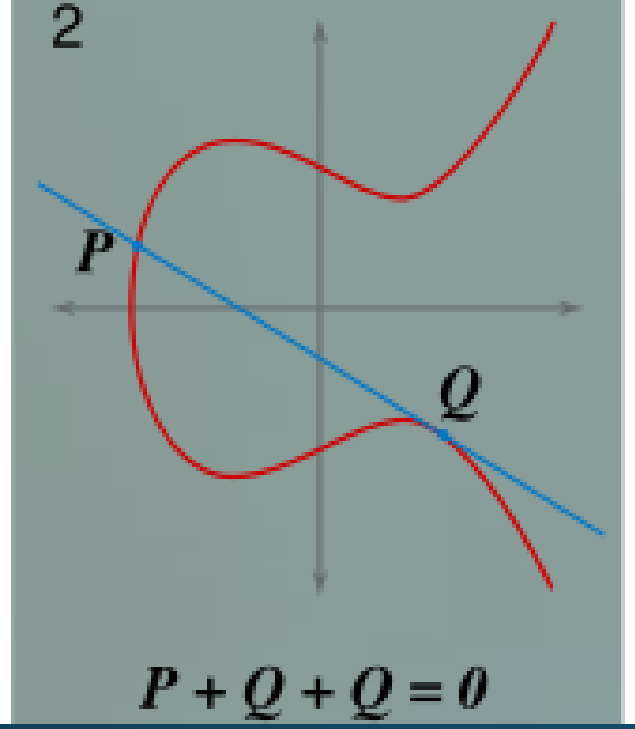
- **Example:** C implementation of field multiplication with $p = 2^{127} - 1$ on Linux
- GNU GCC and clang support 128-bit datatypes (`int128_t`, `uint128_t`), not in ANSI C

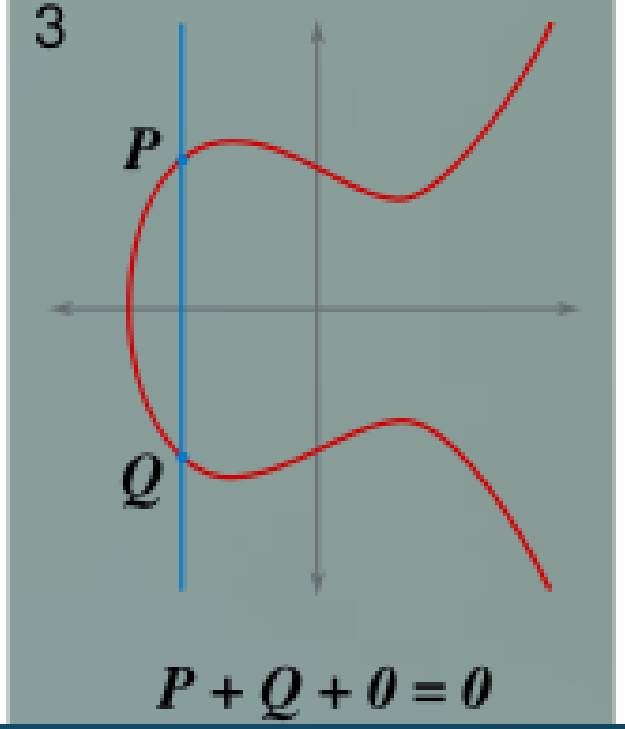
Modular arithmetic: case with reduced-radix

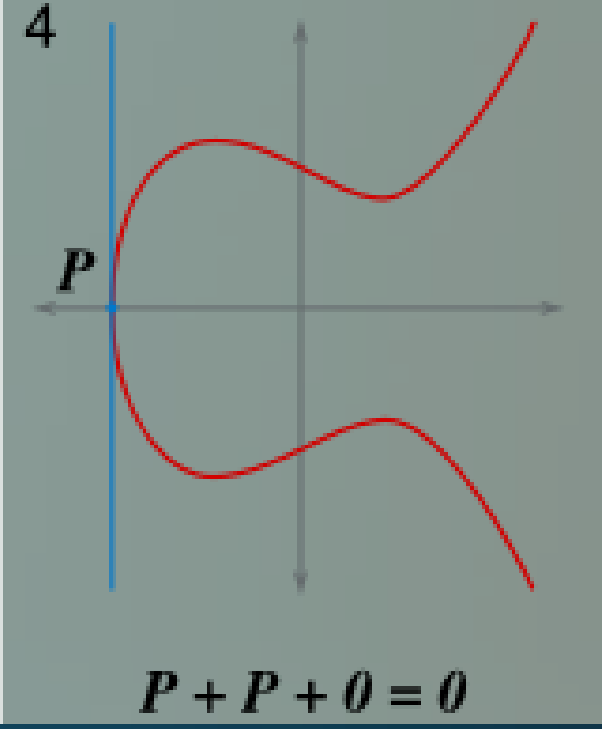
- **Example:** C implementation of field multiplication with $p = 2^{127} - 1$ on Linux
- GNU GCC and clang support 128-bit datatypes (`int128_t`, `uint128_t`), not in ANSI C

```
void fpmul1271(const int64_t* a, const int64_t* b, int64_t* c)
{
   int128_t t[3];
   int64_t  d1, d2;

   a0 = a[0]; a1 = a[1]; b0 = b[0]; b1 = b[1];

   d1 = a1 << 2;
   d2 = a2 << 2;
   t[0] = (int128_t)a0*b0 + (int128_t)d1*b2 + (int128_t)d2*b1;
   t[1] = (int128_t)a0*b1 + (int128_t)a1*b0 + (int128_t)d2*b2;
   t[2] = (int128_t)a0*b2 + (int128_t)a2*b0 + (int128_t)a1*b1;

   t[0] += (int64_t)(t[2] >> 41); c[2] = (int64_t)t[2] & mask_41;
   t[1] += (int64_t)(t[0] >> 43); c[0] = (int64_t)t[0] & mask_43;
   c[2] += (int64_t)(t[1] >> 43); c[1] = (int64_t)t[1] & mask_43;
   c[0] += (int64_t)(c[2] >> 41); c[2] &= mask_41;
}
```









The case of ECC

ECC basics

- The main operation is **scalar multiplication**

ECC basics

- The main operation is **scalar multiplication**
- Let E be an elliptic curve defined over a finite field K, and $E(K)$ be a large prime order subgroup with n elements

ECC basics

- The main operation is **scalar multiplication**
- Let E be an elliptic curve defined over a finite field K, and $E(K)$ be a large prime order subgroup with n elements
- Scalar multiplication consists in computing

$$[k]P = P + P + \cdots + P \ \ (k \text{ times}),$$

where k is an integer in $[0, n)$ and point $P \in E(K)$

ECC basics

- The main operation is **scalar multiplication**
- Let E be an elliptic curve defined over a finite field K, and $E(K)$ be a large prime order subgroup with n elements
- Scalar multiplication consists in computing

$$[k]P = P + P + \cdots + P \ \ (k \text{ times}),$$

  where k is an integer in $[0, n)$ and point $P \in E(K)$

- Elliptic curve-based protocols require different variants of scalar multiplication:
  - **Variable-base scalar multiplication:** $[k]P$ with variable point P
  - **Fixed-base scalar multiplication:** $[k]P$ with fixed point P
  - **Double-scalar multiplication:** $[k]P + [l]Q$ with fixed point P and variable point Q

ECC basics

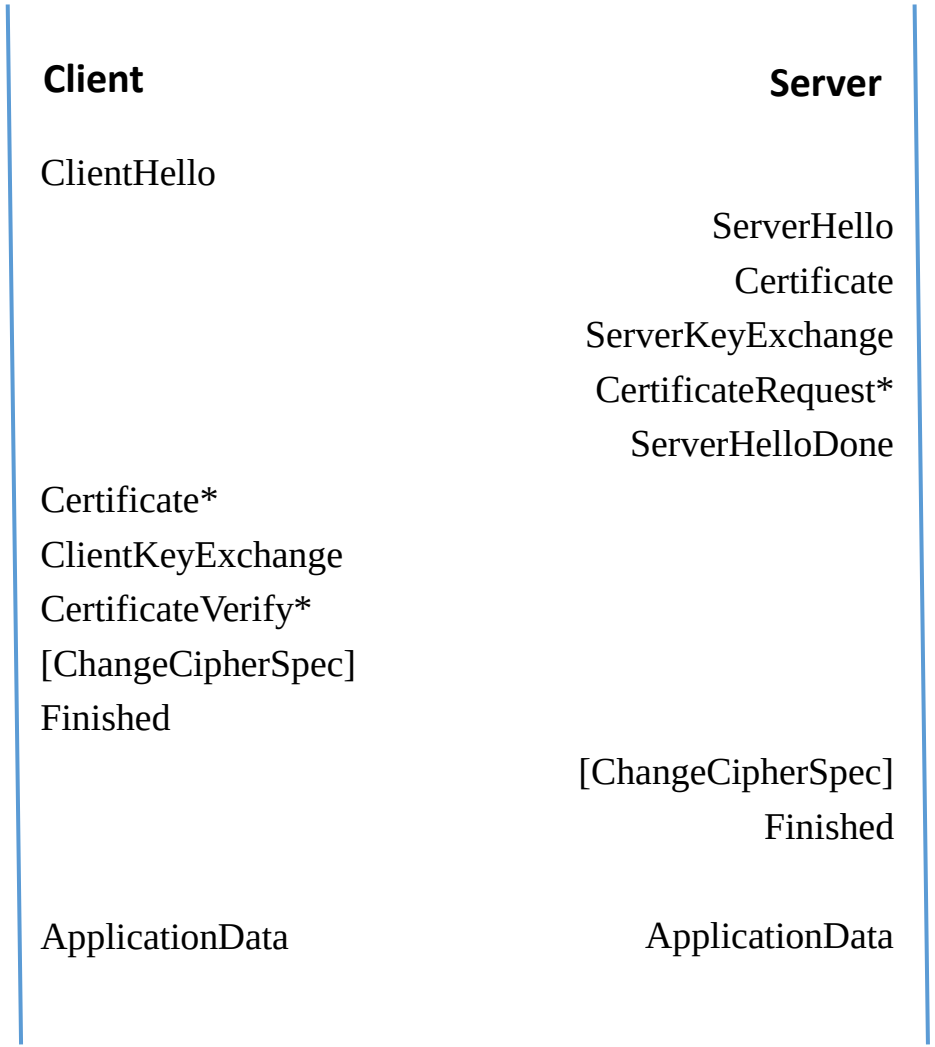
- The main operation is **scalar multiplication**
- Let E be an elliptic curve defined over a finite field K, and $E(K)$ be a large prime order subgroup with n elements
- Scalar multiplication consists in computing

$$[k]P = P + P + \cdots + P \ \ (k \text{ times}),$$

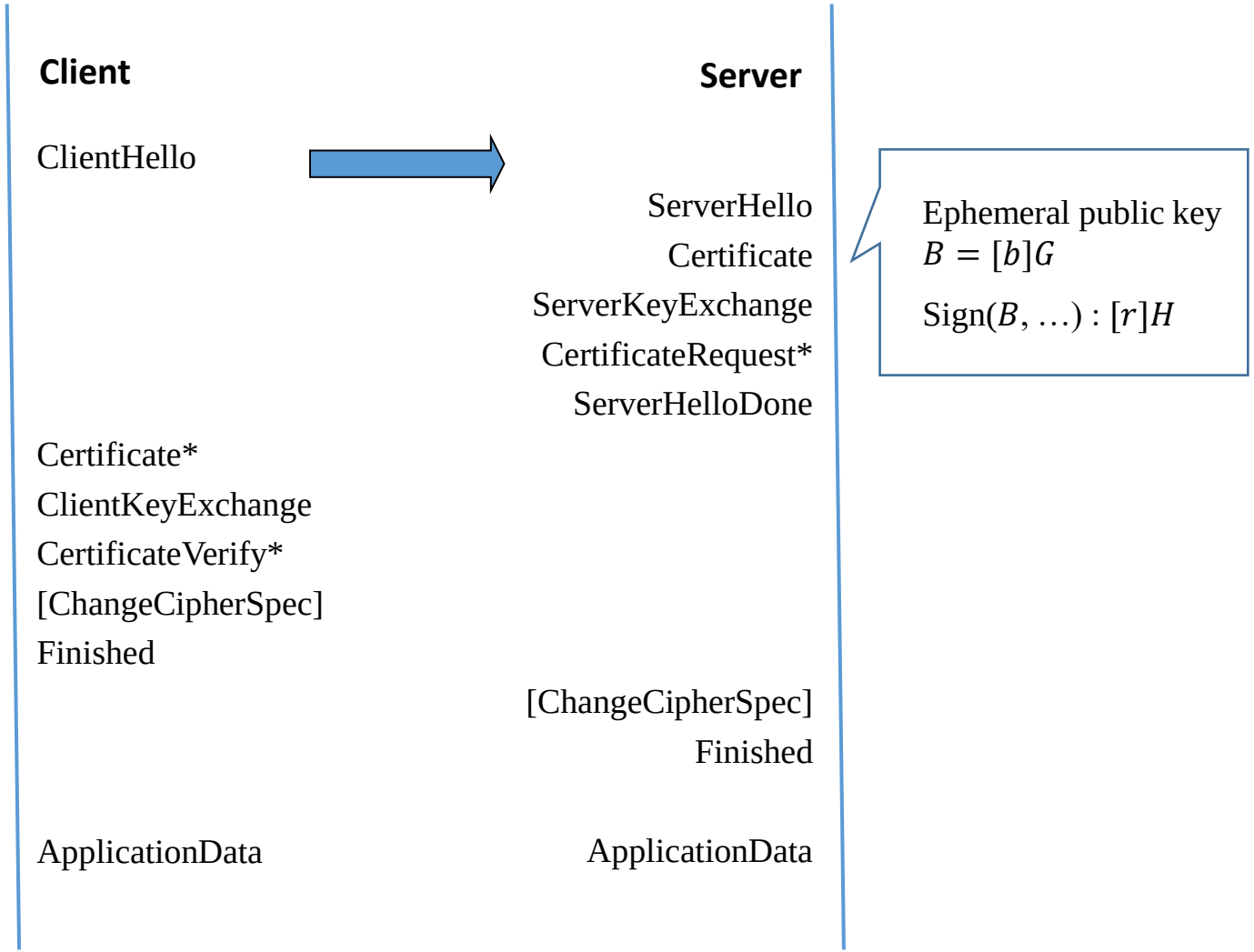
  where k is an integer in $[0, n)$ and point $P \in E(K)$

- Elliptic curve-based protocols require different variants of scalar multiplication:
  - **Variable-base scalar multiplication:** $[k]P$ with variable point P
  - **Fixed-base scalar multiplication:** $[k]P$ with fixed point P
  - **Double-scalar multiplication:** $[k]P + [l]Q$ with fixed point P and variable point Q
- We will focus on **finite fields with large prime characteristic** ($\mathbb{F}_p$ for p prime)

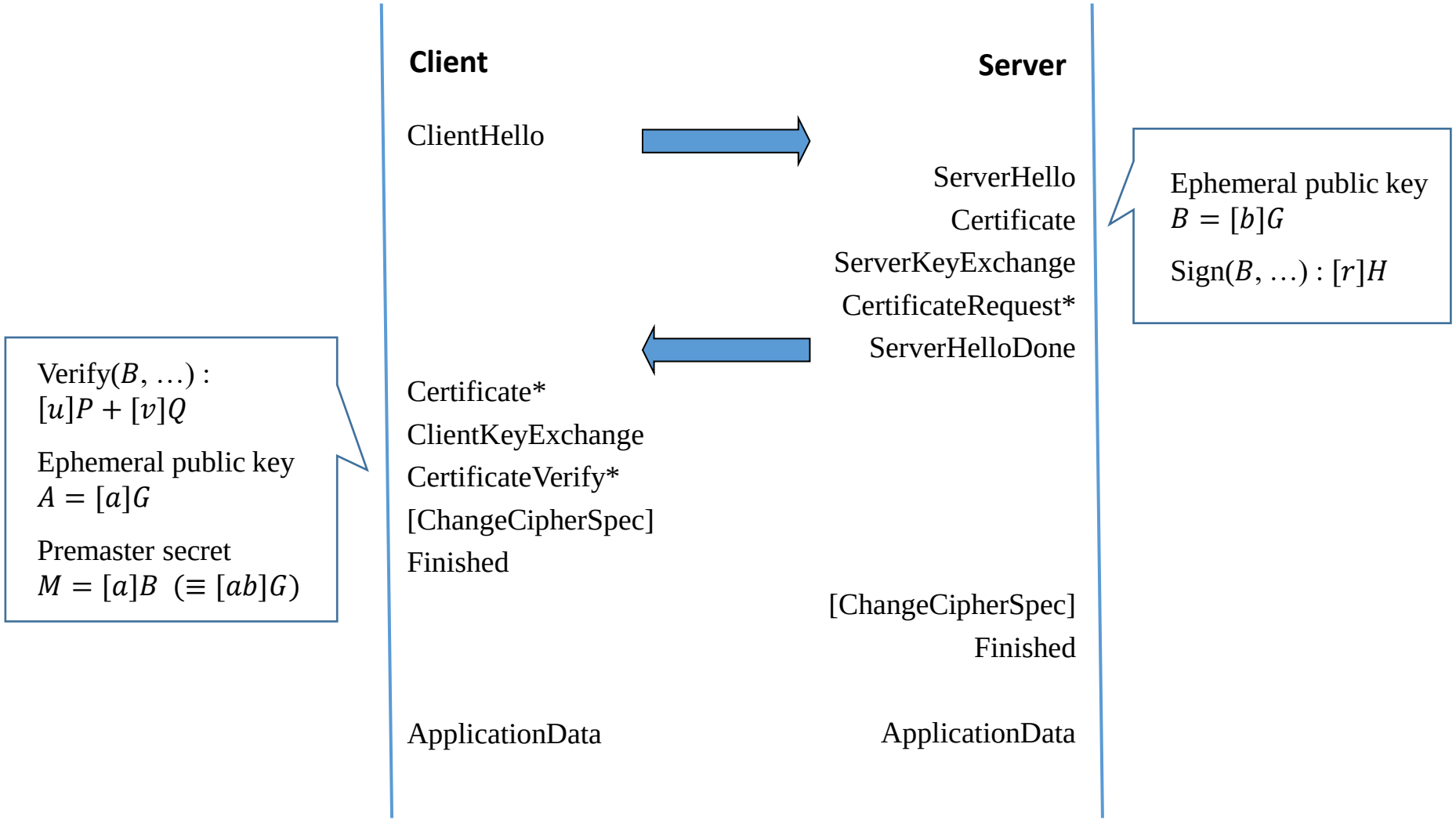
TLS Handshake with ECDHE_ECDSA: an example



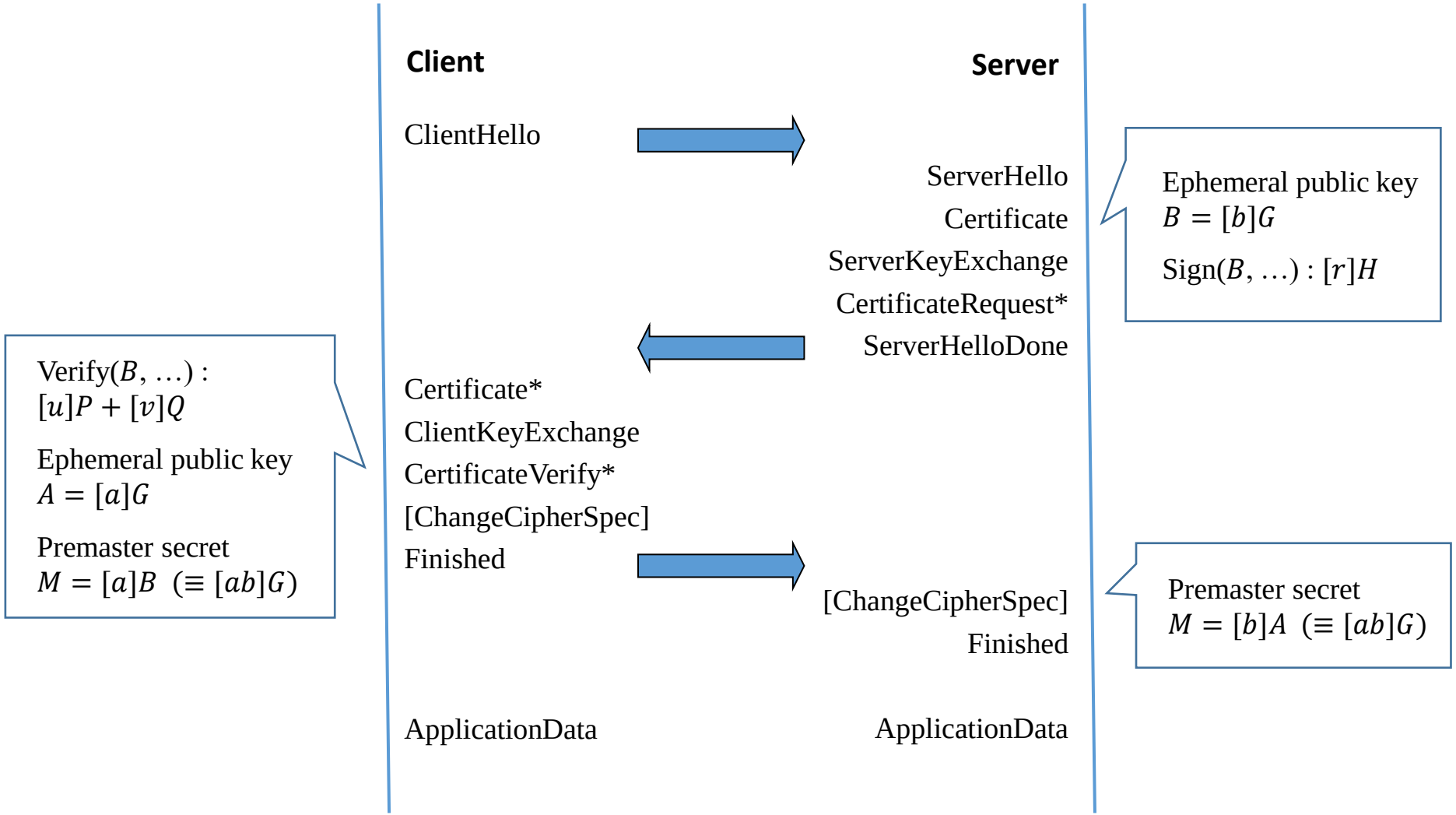
TLS Handshake with ECDHE_ECDSA: an example



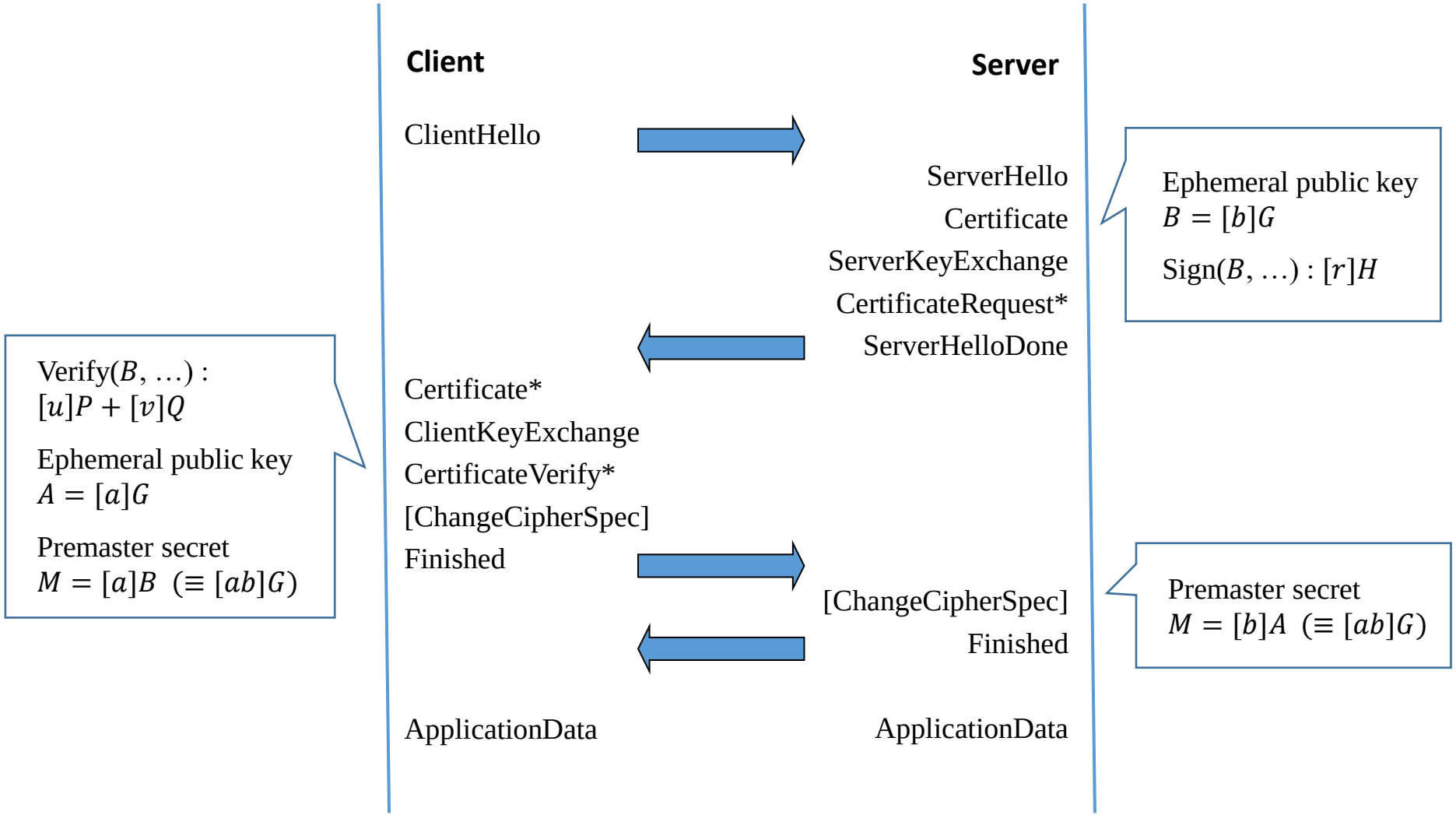
TLS Handshake with ECDHE_ECDSA: an example



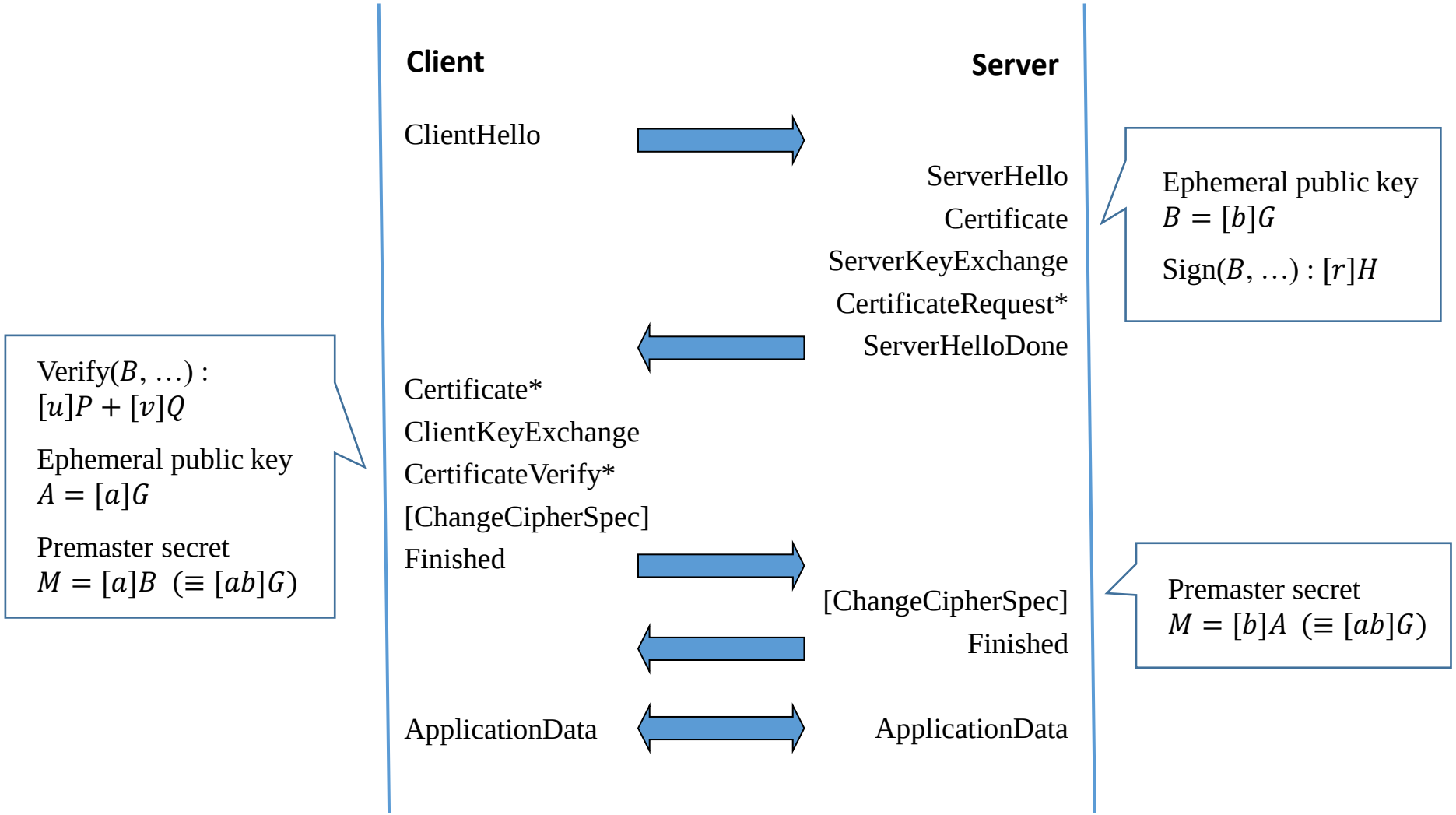
TLS Handshake with ECDHE_ECDSA: an example



TLS Handshake with ECDHE_ECDSA: an example



TLS Handshake with ECDHE_ECDSA: an example



ECC basics

A (short Weierstrass) elliptic curve over a prime field $\mathbb{F}_p$ is given by:

$$E/\mathbb{F}_p\colon\ y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

ECC basics

A (short Weierstrass) elliptic curve over a prime field $\mathbb{F}_p$ is given by:

$$E/\mathbb{F}_p\colon\ y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

- $\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\infty\}$ is an abelian group with group operation (+)
- Given points P and Q, any of the following cases can arise for $R = P + Q$:

$P = \infty \Rightarrow R = Q$
$Q = \infty \Rightarrow R = P$
$P = -Q \Rightarrow R = \infty$
$P = Q \Rightarrow R = 2P$ (point doubling)
otherwise $R = P + Q$ (point addition)

ECC basics

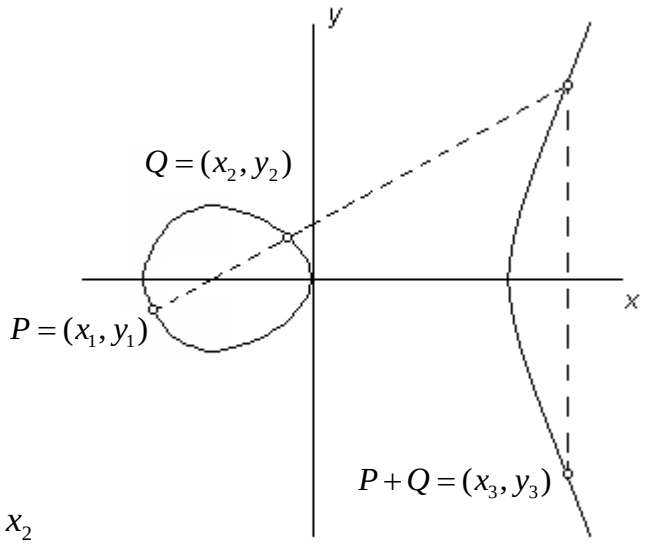
A (short Weierstrass) elliptic curve over a prime field $\mathbb{F}_p$ is given by:

$$E/\mathbb{F}_p\colon\ y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

- $\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\infty\}$ is an abelian group with group operation (+)
- Given points P and Q, any of the following cases can arise for $R = P + Q$:

$$\begin{aligned}
&P = \infty && \Rightarrow R = Q \\
&Q = \infty && \Rightarrow R = P \\
&P = -Q && \Rightarrow R = \infty \\
&P = Q && \Rightarrow R = 2P && \text{(point doubling)} \\
&\text{otherwise} && R = P + Q && \text{(point addition)}
\end{aligned}$$



$$\begin{cases}
\lambda = \dfrac{y_2 - y_1}{x_2 - x_1} \\
x_3 = \lambda^2 - x_1 - x_2 \\
y_3 = \lambda(x_1 - x_3) - y_1
\end{cases}$$

point addition

ECC basics

A (short Weierstrass) elliptic curve over a prime field $\mathbb{F}_p$ is given by:

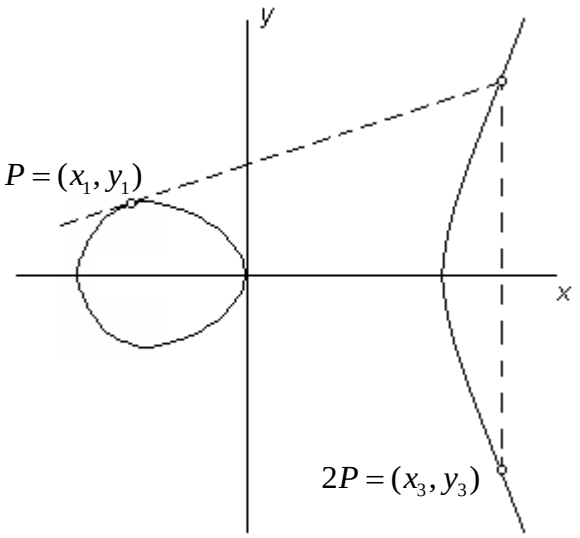
$$E/\mathbb{F}_p\colon\ y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

- $\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\infty\}$ is an abelian group with group operation (+)
- Given points P and Q, any of the following cases can arise for $R = P + Q$:

$$\begin{aligned}
&P = \infty && \Rightarrow R = Q \\
&Q = \infty && \Rightarrow R = P \\
&P = -Q && \Rightarrow R = \infty \\
&P = Q && \Rightarrow R = 2P && \text{(point doubling)} \\
&\text{otherwise} && R = P + Q && \text{(point addition)}
\end{aligned}$$

$$\begin{cases}
\lambda = \dfrac{3x_1^2 + a}{2y_1} \\
x_3 = \lambda^2 - 2x_1 \\
y_3 = \lambda(x_1 - x_3) - y_1
\end{cases}$$

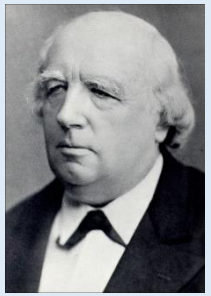


point doubling

Elliptic curve forms

Weierstrass curves

$$y^2 = x^3 + ax + b$$

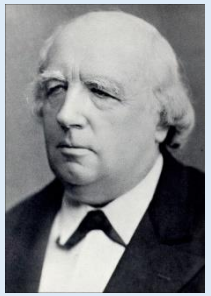


- Most general form
- Prime order possible
- Exceptions in group law (but complete formulas exist)
- NIST and Brainpool curves

Elliptic curve forms

Weierstrass curves

$$y^2 = x^3 + ax + b$$

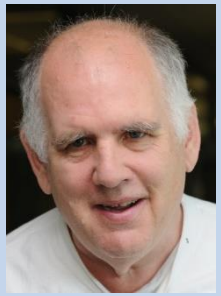


- Most general form
- Prime order possible
- Exceptions in group law (but complete formulas exist)
- NIST and Brainpool curves

Montgomery curves

$$By^2 = x^3 + Ax^2 + x$$

- Subset of curves
- Not prime order
- Montgomery ladder

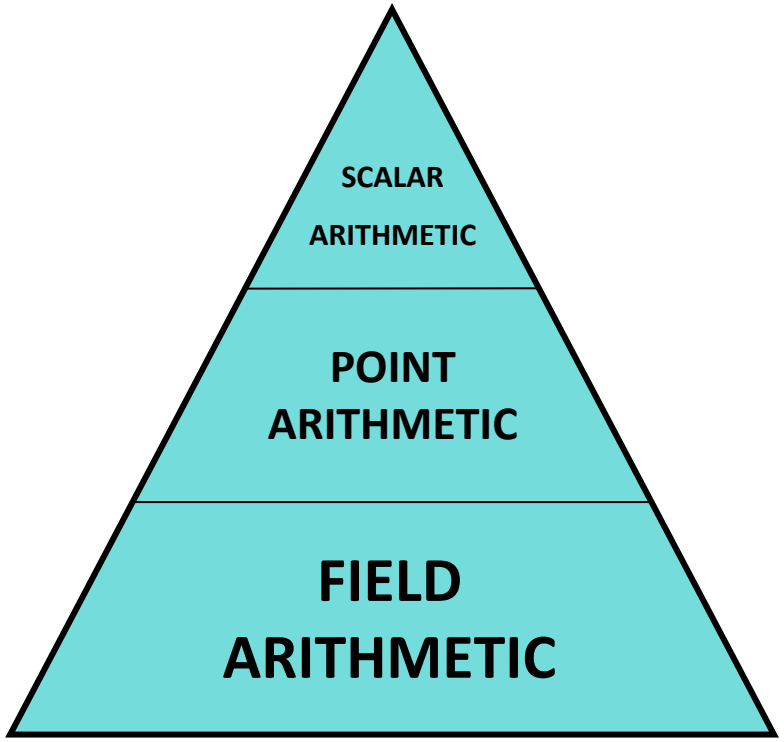


(Twisted) Edwards curves

$$ax^2 + y^2 = 1 + dx^4y^4$$

- Subset of curves
- Not prime order
- Fastest arithmetic
- Some have complete group law

ECC arithmetic layers



Scalar multiplication operations
E.g., $[k]P$, $[u]P+[v]Q$, *etc.*

Doubling and addition of points:
$2P$, $P+Q$

Field addition, subtraction, multiplication, squaring, inversion

Implementation Security

Exception attacks

- Failures during computations may leak information

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- “Clear” small torsion on Montgomery and twisted Edwards curves

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- “Clear” small torsion on Montgomery and twisted Edwards curves:
  - At the end, as part of scalar multiplication (e.g., Curve25519)

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- "Clear" small torsion on Montgomery and twisted Edwards curves:
  - At the end, as part of scalar multiplication (e.g., Curve25519)
  - Before computing scalar multiplication (e.g, Four$\mathbb{Q}$)

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- "Clear" small torsion on Montgomery and twisted Edwards curves:
  - At the end, as part of scalar multiplication (e.g., Curve25519)
  - Before computing scalar multiplication (e.g, Four$\mathbb{Q}$)
  - Use a quotient group (e.g., $E/E[4]$) using encodings like Decaf/Ristretto

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- "Clear" small torsion on Montgomery and twisted Edwards curves:
  - At the end, as part of scalar multiplication (e.g., Curve25519)
  - Before computing scalar multiplication (e.g, Four$\mathbb{Q}$)
  - Use a quotient group (e.g., $E/E[4]$) using encodings like Decaf/Ristretto

Invalid curve attacks

- *Simple and inexpensive* solution (for Weierstrass and twisted Edwards): **validate input points**

Implementation Security

Exception attacks

- Failures during computations may leak information
  Solution: build exception-free scalar multiplications (e.g., using *complete* addition laws)

Small subgroup attacks

- Not a problem for prime-order Weierstrass curves
- "Clear" small torsion on Montgomery and twisted Edwards curves:
  - At the end, as part of scalar multiplication (e.g., Curve25519)
  - Before computing scalar multiplication (e.g, Four$\mathbb{Q}$)
  - Use a quotient group (e.g., $E/E[4]$) using encodings like Decaf/Ristretto

Invalid curve attacks

- *Simple and inexpensive* solution (for Weierstrass and twisted Edwards): **validate input points**
- Not a problem for twist-secure curves

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Regular scalar multiplications:

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Regular scalar multiplications:

- **Fixed-window method** [Okeya-Takagi 2003]: no curve restriction, adjustable window size, efficient for variable-base

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Regular scalar multiplications:

- **Fixed-window method** [Okeya-Takagi 2003]: no curve restriction, adjustable window size, efficient for variable-base
- **Regular comb methods:** no curve restriction, adjustable window size, efficient for fixed-base
  E.g., [Faz-Longa-Sanchez 2013]

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Regular scalar multiplications:

- **Fixed-window method** [Okeya-Takagi 2003]: no curve restriction, adjustable window size, efficient for variable-base
- **Regular comb methods:** no curve restriction, adjustable window size, efficient for fixed-base E.g., [Faz-Longa-Sanchez 2013]
- **Ladder:** compact, especially efficient on Montgomery curves and for variable-base

Implementation Security

Side-channel attacks: timing, SSCA, DSCA, etc.

- Use “regular” arithmetic to deal with simple side-channels
- **Alternatives:** regular algorithms for scalar multiplication, use of unified addition

Regular scalar multiplications:

- **Fixed-window method** [Okeya-Takagi 2003]: no curve restriction, adjustable window size, efficient for variable-base
- **Regular comb methods:** no curve restriction, adjustable window size, efficient for fixed-base E.g., [Faz-Longa-Sanchez 2013]
- **Ladder:** compact, especially efficient on Montgomery curves and for variable-base

Unified addition:

- Simple and compact, but expensive

Performance

Prime ECC has experienced dramatic speed-ups over the years due to improvements in:

computer architectures + elliptic curves algorithms + finite field algorithms

Performance

Prime ECC has experienced dramatic speed-ups over the years due to improvements in:

computer architectures + elliptic curves algorithms + finite field algorithms

$[k]P$ ran in…

1,920,000 cc	Brown-Hankerson-López-Menezes 2000 (NIST P-256)
625,000 cc	**Bernstein 2006 (Curve25519)**
307,000 cc	Gaudry-Thomé 2007 (Curve25519)
293,000 cc	Galbraith-Lin-Scott 2008 (gls1271)
181,000 cc	Longa 2010 (gls1271)
137,000 cc	**Longa-Sica 2012 (GLV-GLS)**
89,000 cc	**Faz-Longa-Sánchez 2013 (Ted127-glv4)**
50,000 cc	**Costello-Longa 2015 (Four$\mathbb{Q}$)**

- Implementations that run in constant-time are in **bold**

Performance

Prime ECC has experienced dramatic speed-ups over the years due to improvements in:

computer architectures + elliptic curves algorithms + finite field algorithms

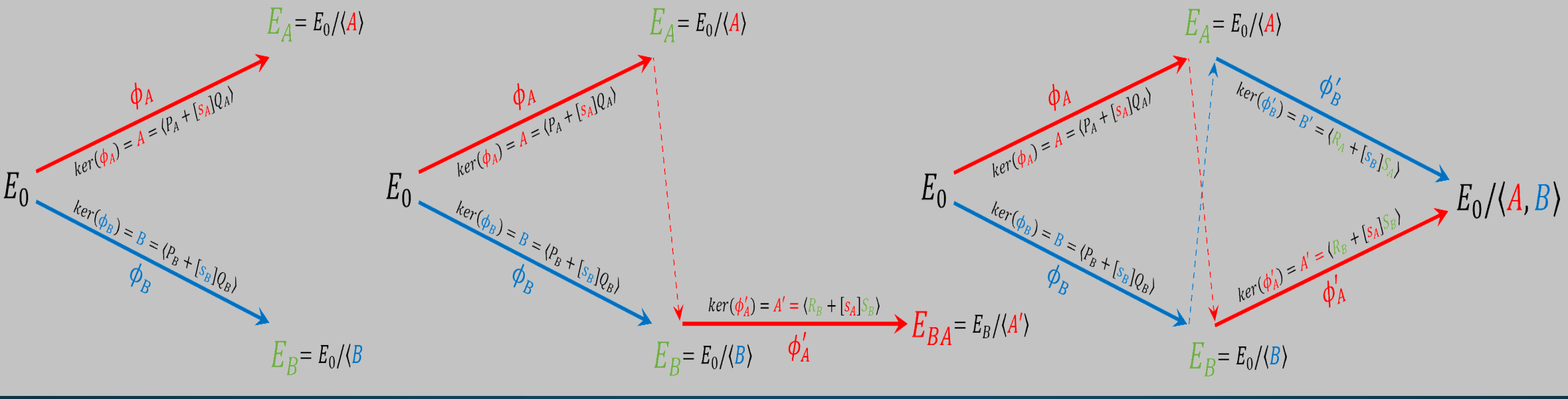
$[k]P$ ran in…

1,920,000 cc	Brown-Hankerson-López-Menezes 2000 (NIST P-256)
625,000 cc	**Bernstein 2006 (Curve25519)**
307,000 cc	Gaudry-Thomé 2007 (Curve25519)
293,000 cc	Galbraith-Lin-Scott 2008 (gls1271)
181,000 cc	Longa 2010 (gls1271)
137,000 cc	**Longa-Sica 2012 (GLV-GLS)**
89,000 cc	**Faz-Longa-Sánchez 2013 (Ted127-glv4)**
50,000 cc	**Costello-Longa 2015 (Four$\mathbb{Q}$)**

Today, it runs in…

46,000 cc	**Costello-Longa 2015 (Four$\mathbb{Q}$)**

- Implementations that run in constant-time are in **bold**



The case of SIKE

Supersingular isogeny key exchange

Supersingular isogeny Diffie-Hellman key exchange (SIDH)

- Proposed by Jao and De Feo in 2011.
- Compared to “predecessors” based on *ordinary isogenies*, SIDH has:
  - Much better performance
  - Exponential complexity of best classical and quantum attacks.

Supersingular isogeny key exchange

Supersingular isogeny Diffie-Hellman key exchange (SIDH)

- Proposed by Jao and De Feo in 2011.
- Compared to “predecessors” based on *ordinary isogenies*, SIDH has:
  - Much better performance
  - Exponential complexity of best classical and quantum attacks.

Supersingular isogeny key encapsulation (SIKE)

- Designed by Costello–De Feo–Jao–L–Naehrig–Renes in 2017.
- IND-CCA secure key encapsulation protocol based on SIDH.
- Submitted to the NIST PQC standardization process.

Elliptic curves and isogenies

- Every elliptic curve over a field K with $\mathrm{char}(K) > 3$ can be defined in (short) Weierstrass form by

$$E: y^2 = x^3 + ax + b,$$

where $a, b \in K$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Elliptic curves and isogenies

- Every elliptic curve over a field K with $\mathrm{char}(K) > 3$ can be defined in (short) Weierstrass form by

$$E\colon y^2 = x^3 + ax + b,$$

where $a, b \in K$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

- For an extension field L of K, the set of L-rational points on E

$$E(L) = \{(x, y) \in L \times L\colon y^2 - x^3 - ax - b = 0\} \cup \{\mathcal{O}\},$$

together with the *group addition law*, forms an abelian group with identity $\mathcal{O}$.

Elliptic curves and isogenies

- Every elliptic curve over a field K with $\mathrm{char}(K) > 3$ can be defined in (short) Weierstrass form by

$$E\colon y^2 = x^3 + ax + b,$$

  where $a, b \in K$ and $\Delta = -16(4a^3 + 27b^2) \neq 0$.

- For an extension field L of K, the set of L-rational points on E

$$E(L) = \{(x, y) \in L \times L\colon y^2 - x^3 - ax - b = 0\} \cup \{\mathcal{O}\},$$

  together with the *group addition law*, forms an abelian group with identity $\mathcal{O}$.

- **Isomorphism classes** are determined by the ***j*-invariant**: $j(E) = 1728 \cdot \frac{4a^3}{4a^3 + 27b^2}$

Elliptic curves and isogenies

- Let E_1 and E_2 be elliptic curves defined over an extension field L.
- An isogeny is a (non-constant) rational map

$$\phi: E_1 \rightarrow E_2$$

that preserves identity, i.e., $\phi(\mathcal{O}_{E_1}) \rightarrow \mathcal{O}_{E_2}$.

Elliptic curves and isogenies

- Let E_1 and E_2 be elliptic curves defined over an extension field L.
- An isogeny is a (non-constant) rational map

$$\phi: E_1 \to E_2$$

that preserves identity, i.e., $\phi(\mathcal{O}_{E_1}) \to \mathcal{O}_{E_2}$.

Relevant properties:

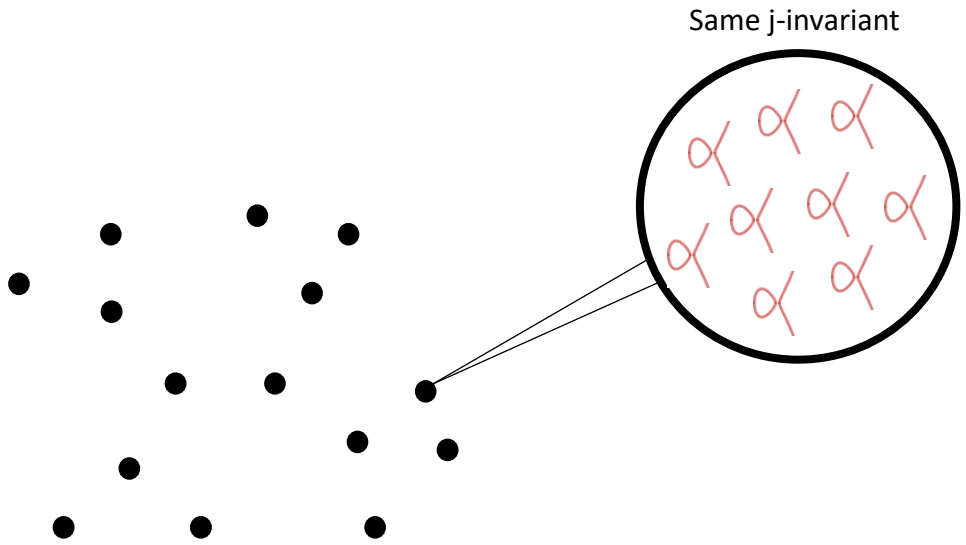
- Isogenies are group homomorphisms.
- For every finite subgroup $\langle G \rangle \subseteq E_1$, there is a unique curve E_2 (up to isomorphism) and isogeny $\phi: E_1 \to E_2$ with kernel $\langle G \rangle$. Write $E_2 = \phi(E_1) = E_1/\langle G \rangle$.
- (Separable) isogenies have $\deg(\phi) = \#\ker(\phi)$.

Supersingular curves

- An elliptic curve E/L is supersingular if $\#E(L) \equiv 1 (\mathrm{mod}\ p)$.
- All supersingular curves can be defined over $\mathbb{F}_{p^2}$.
- There are $\sim\lfloor \boldsymbol{p/12} \rfloor$ **isomorphism classes** of supersingular curves.

Supersingular isogeny graphs

- Vertices: the $\sim\lfloor p/12 \rfloor$ isomorphism classes of supersingular curves over $\mathbb{F}_{p^2}$.

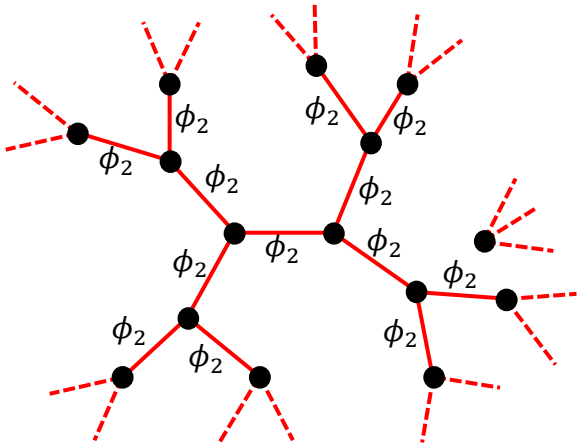


Supersingular isogeny graphs

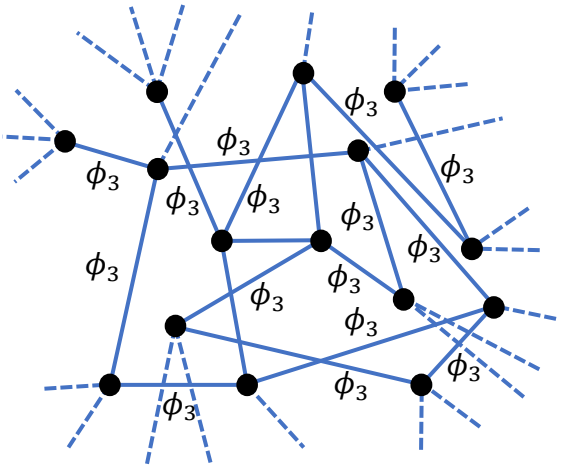
- Vertices: the $\sim\lfloor p/12 \rfloor$ isomorphism classes of supersingular curves over $\mathbb{F}_{p^2}$.
- Edges: isogenies of a fixed prime degree $\ell \nmid p$

Supersingular isogeny graphs

- Vertices: the $\sim\lfloor p/12 \rfloor$ isomorphism classes of supersingular curves over $\mathbb{F}_{p^2}$.
- Edges: isogenies of a fixed prime degree $\ell \nmid p$



$\ell = 2$

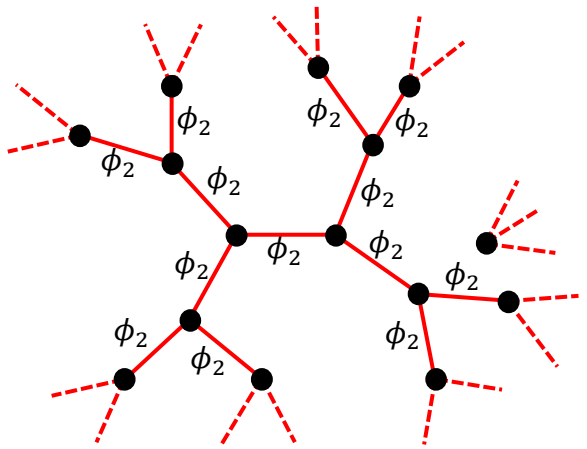


$\ell = 3$

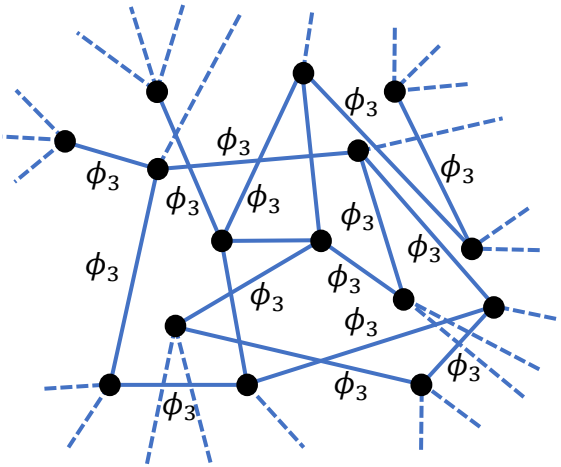
Supersingular isogeny graphs

- Vertices: the $\sim\lfloor p/12 \rfloor$ isomorphism classes of supersingular curves over $\mathbb{F}_{p^2}$.
- Edges: isogenies of a fixed prime degree $\ell \nmid p$

For any prime $\ell \nmid p$, there exist $(\ell + 1)$ isogenies of degree ℓ originating from every supersingular curve.



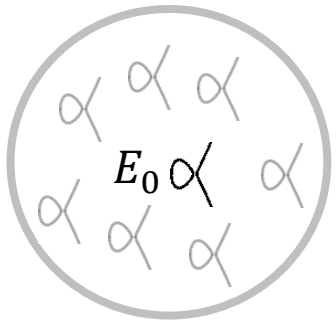
$\ell = 2$

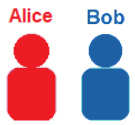


$\ell = 3$

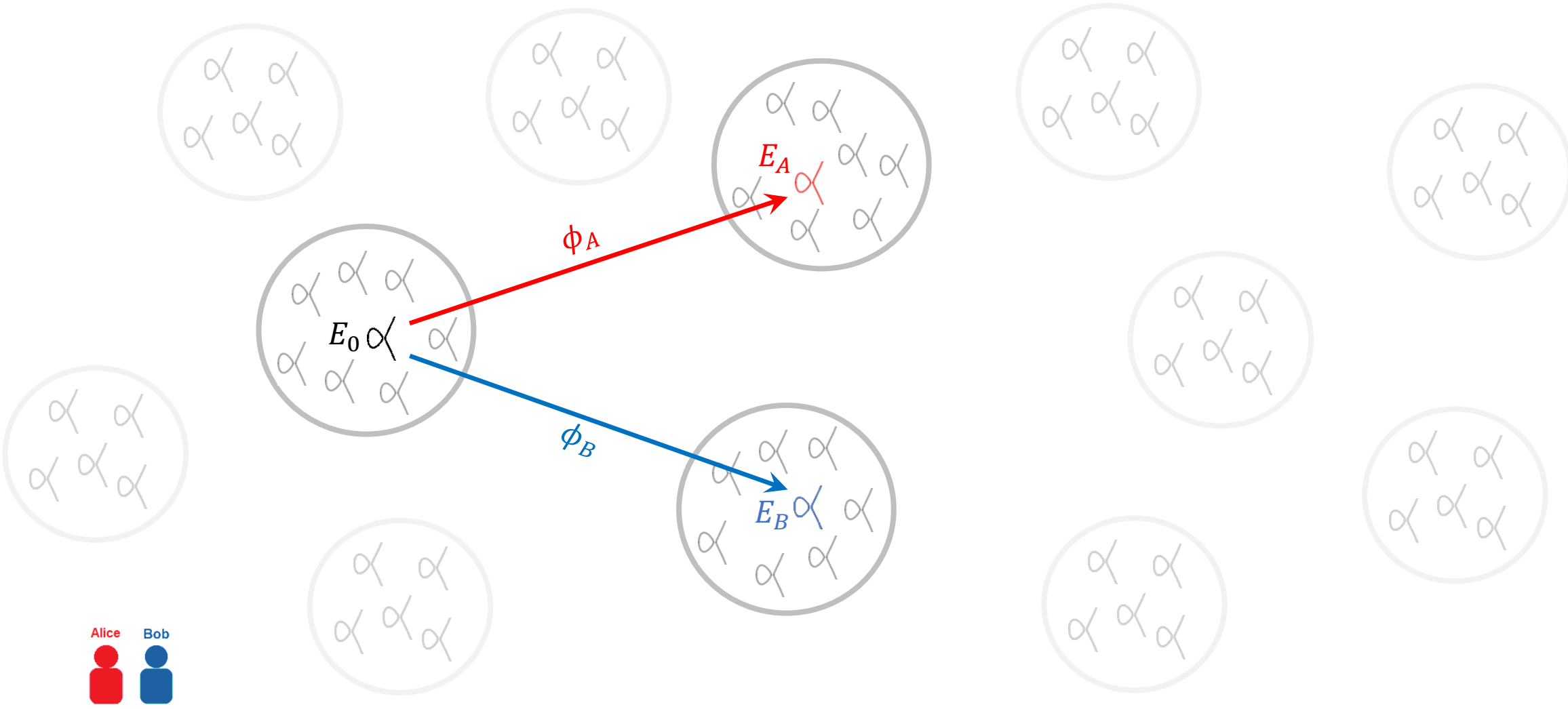
SIDH in a nutshell

SIDH in a nutshell

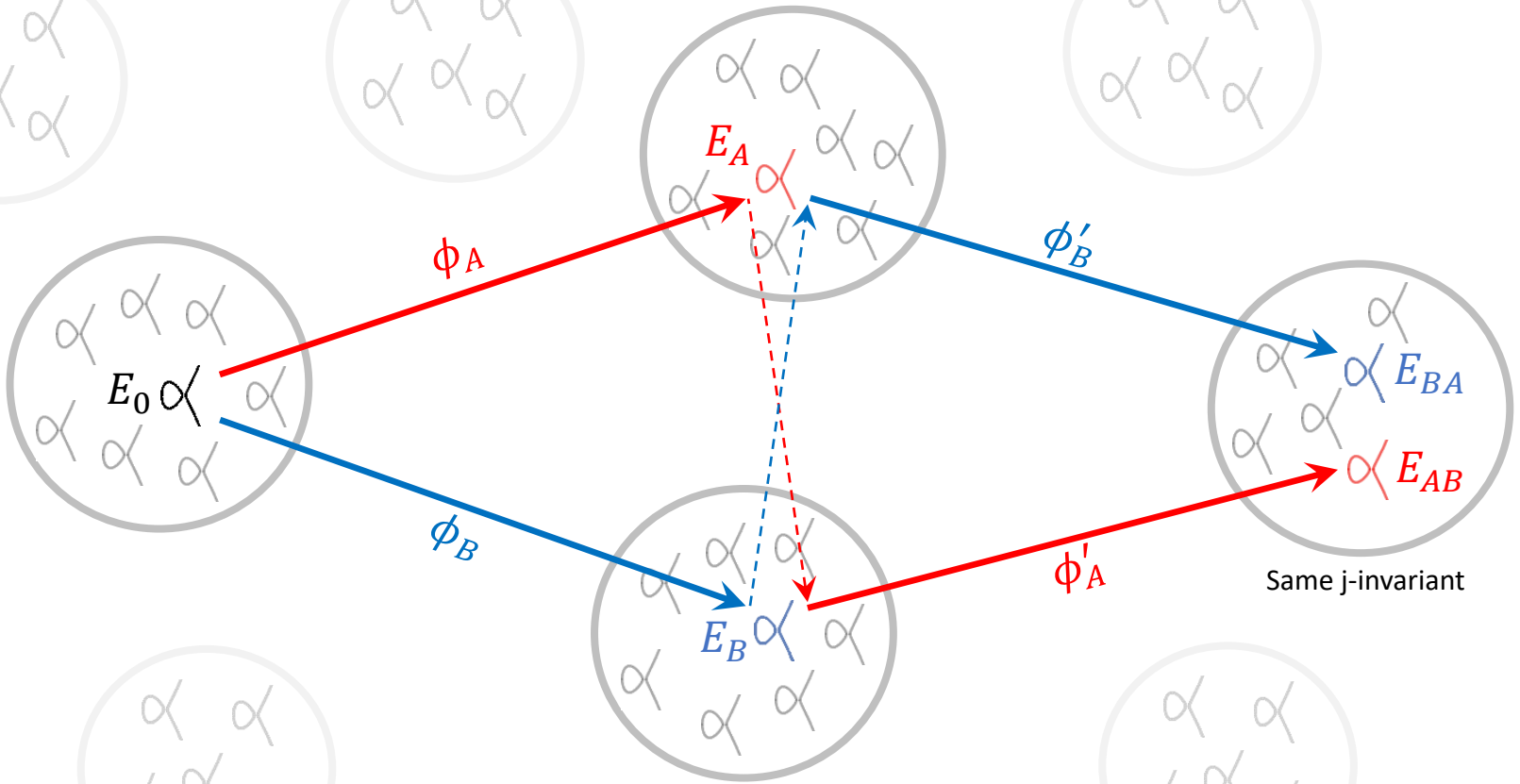


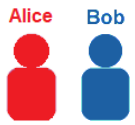


SIDH in a nutshell



SIDH in a nutshell





SIDH: setup

Set $\ell \in \{2,3\}$, supersingular curve $E_0/\mathbb{F}_{p^2}$ with a prime $p = f \cdot 2^{e_A}3^{e_B} - 1$ such that $2^{e_A} \approx 3^{e_B}$ and f small.

- Then: $E[2^{e_A}], E[3^{e_B}] \subset E_0(\mathbb{F}_{p^2})$

Alice

works over $E[2^{e_A}]$ using **2-isogenies** and linearly independent points P_A, Q_A.

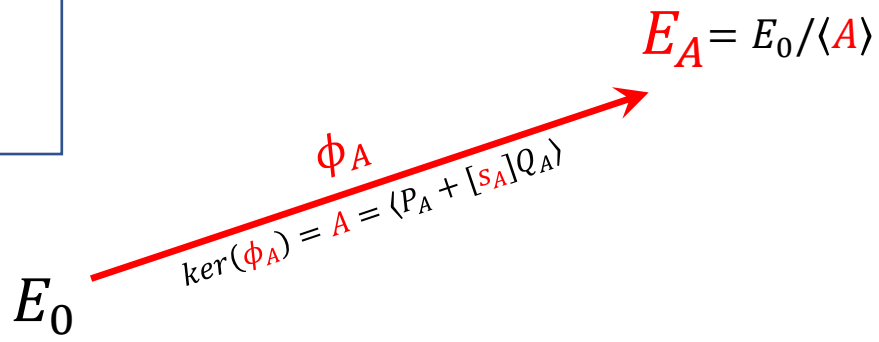
Bob

works over $E[3^{e_B}]$ using **3-isogenies** and linearly independent points P_B, Q_B.

SIDH protocol

private Alice private Bob
public params

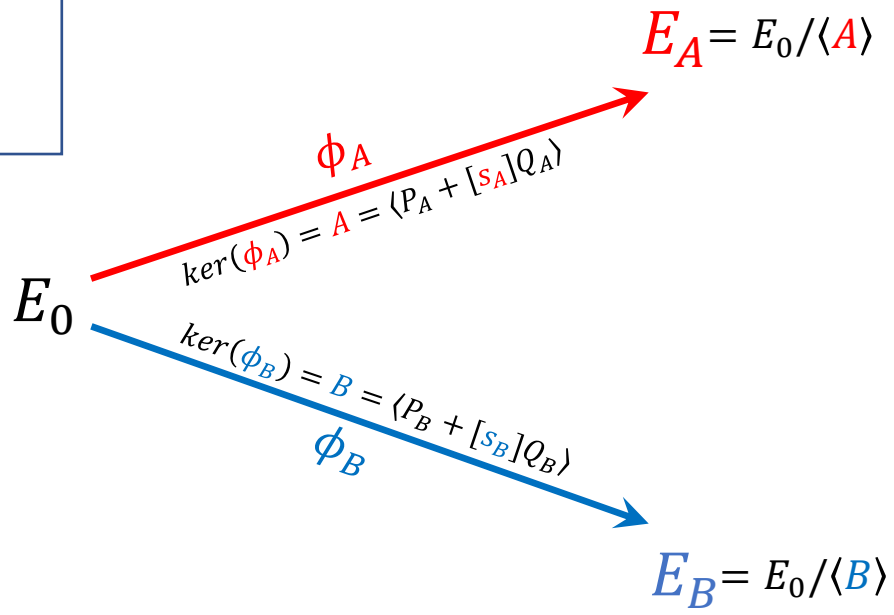
E's are isogenous curves
P's, Q's, R's, S's are points



SIDH protocol

private Alice private Bob
public params

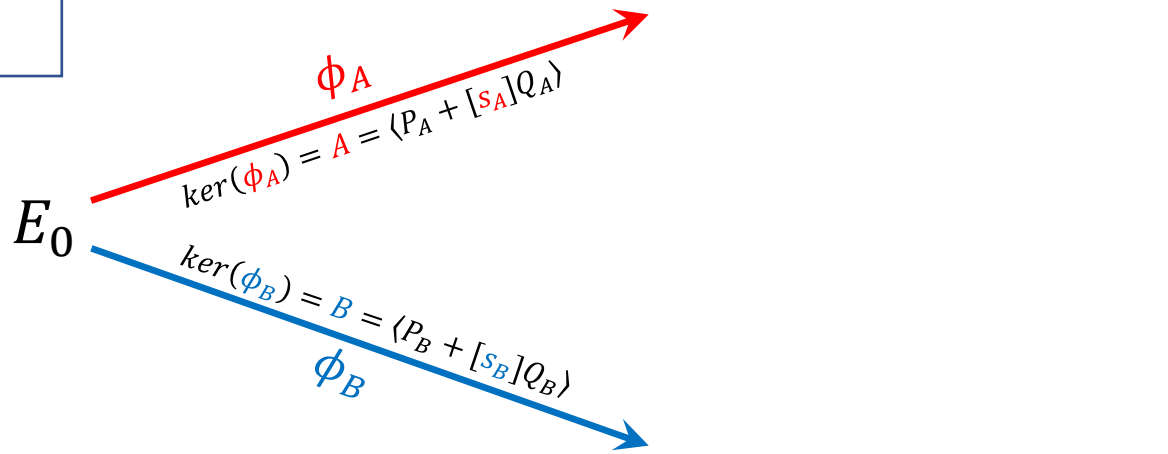
E's are isogenous curves
P's, Q's, R's, S's are points



SIDH protocol

private Alice private Bob
public params

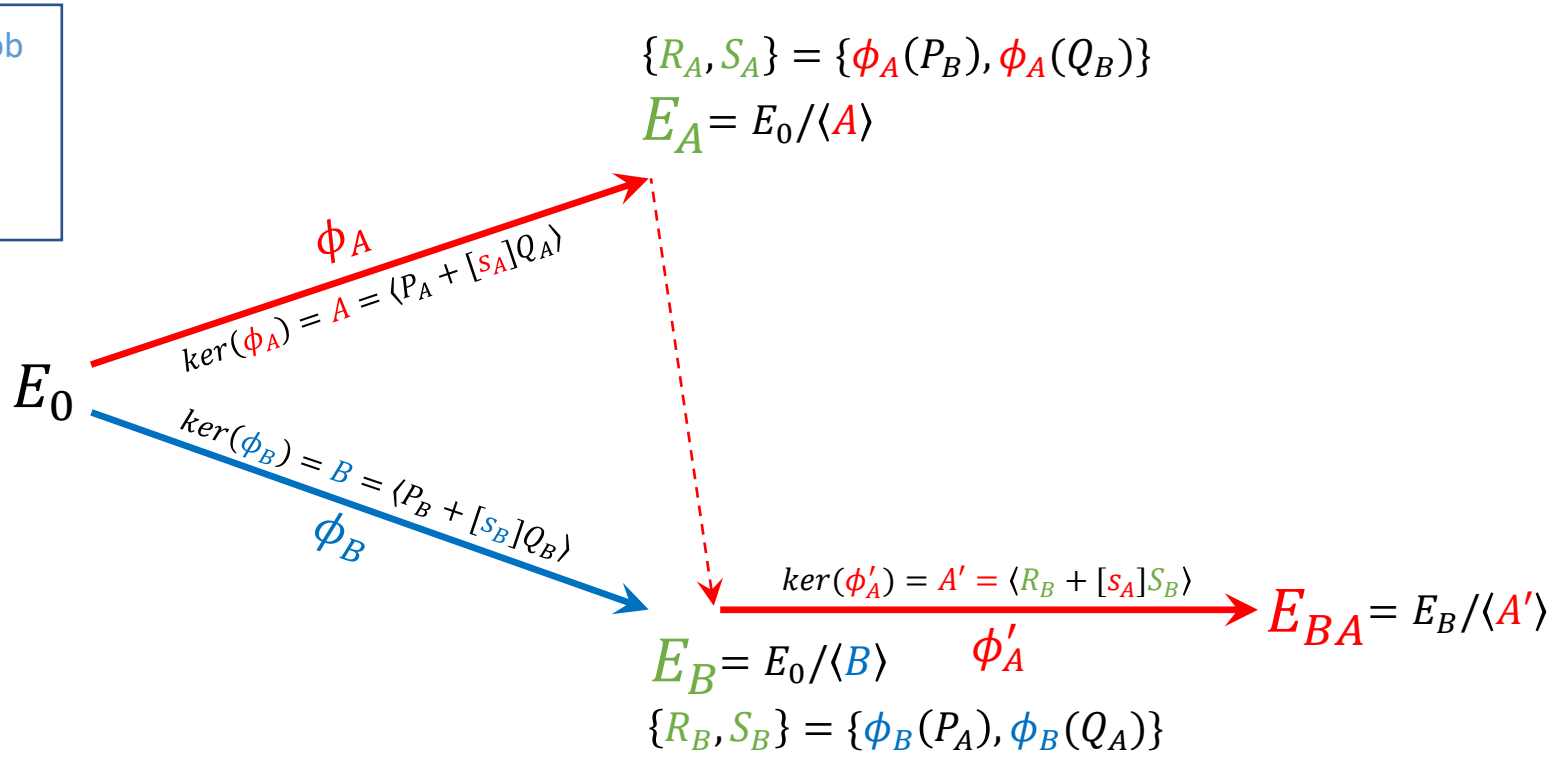
E's are isogenous curves
P's, Q's, R's, S's are points



SIDH protocol

private Alice private Bob
public params

E's are isogenous curves
P's, Q's, R's, S's are points

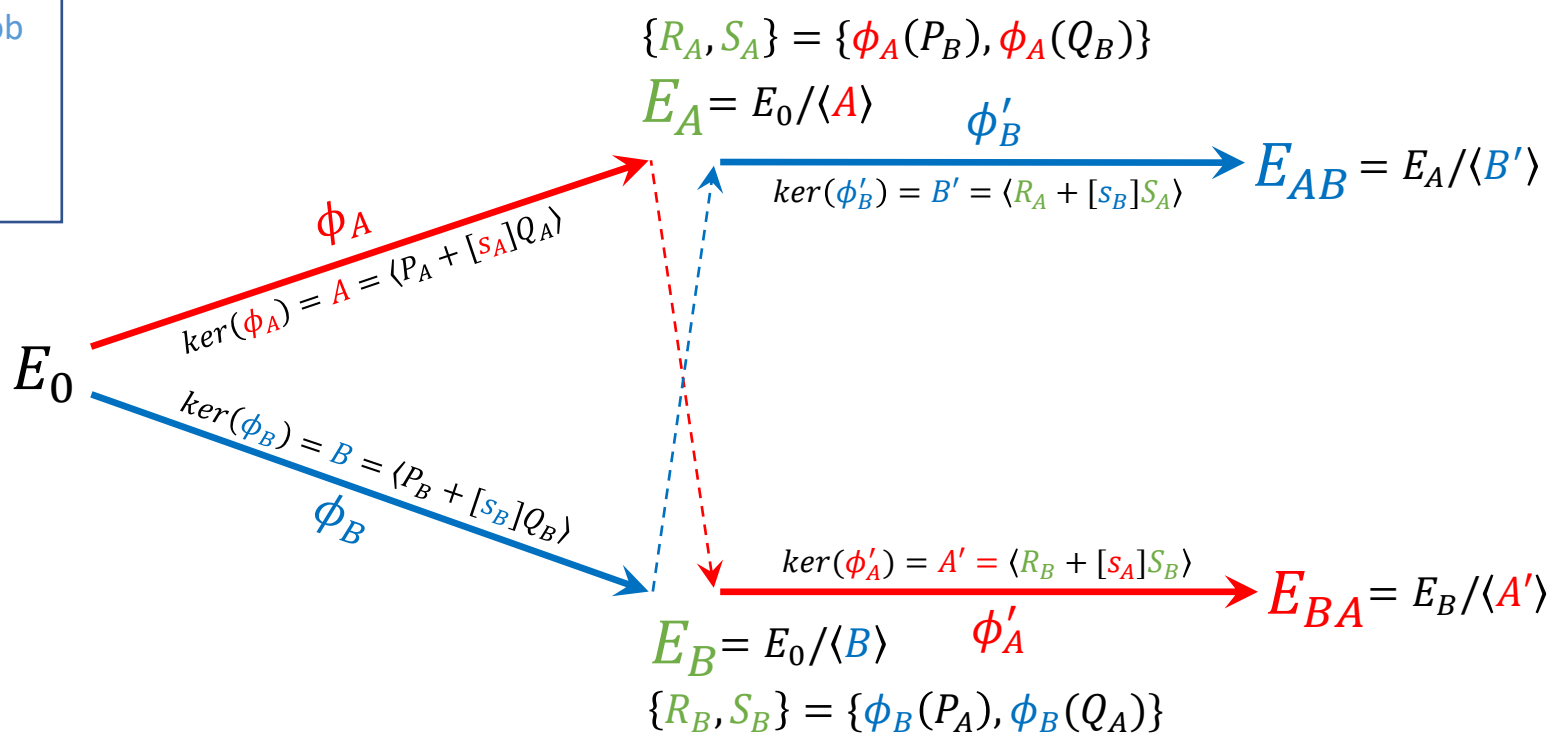


$$A' = \langle \phi_B(P_A) + [s_A]\phi_B(Q_A) \rangle = \langle \phi_B(P_A + [s_A]Q_A) \rangle = \langle \phi_B(A) \rangle$$

SIDH protocol

private Alice private Bob
public params

E's are isogenous curves
P's, Q's, R's, S's are points



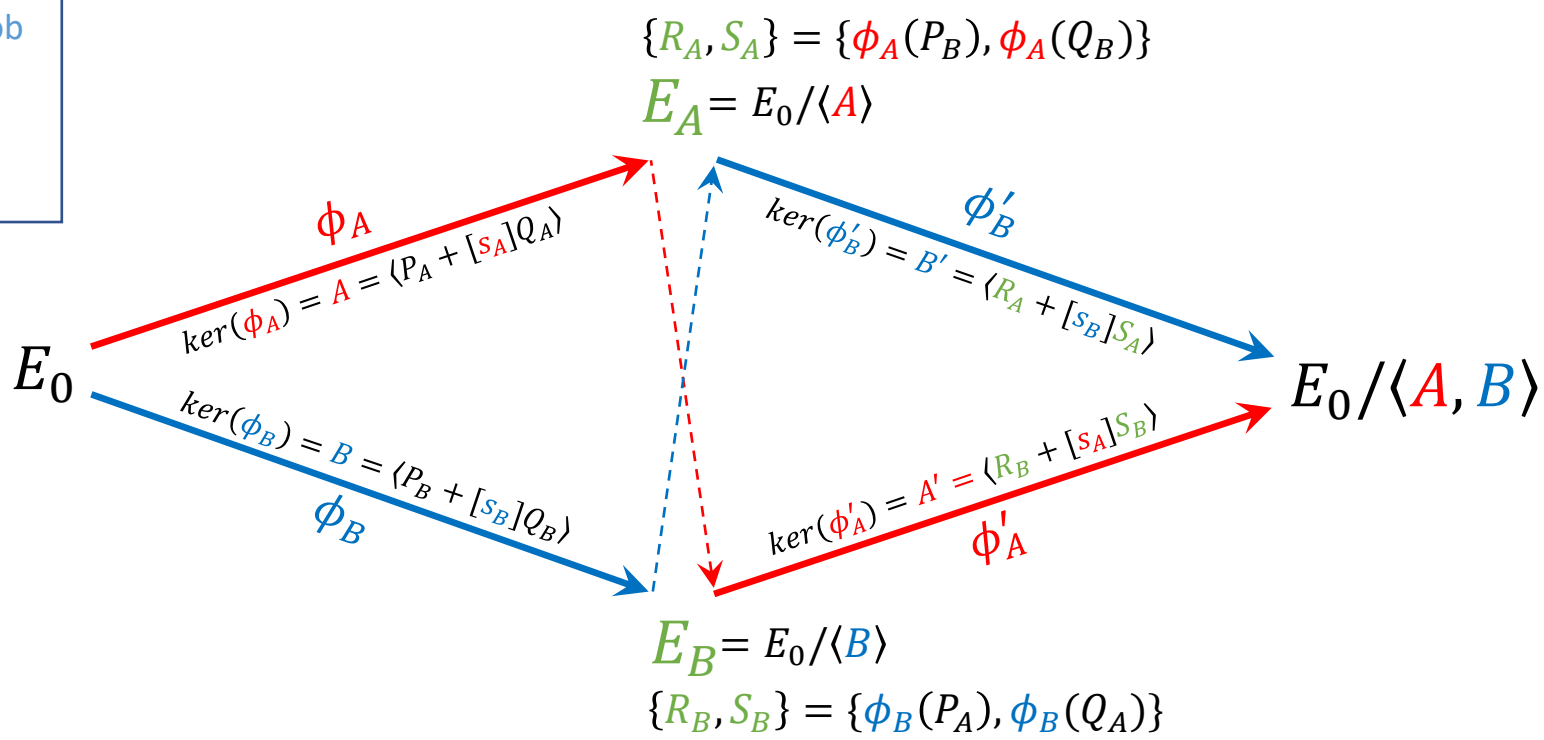
$$A' = \langle \phi_B(P_A) + [s_A]\phi_B(Q_A) \rangle = \langle \phi_B(P_A + [s_A]Q_A) \rangle = \langle \phi_B(A) \rangle$$
$$B' = \langle \phi_A(P_B) + [s_B]\phi_A(Q_B) \rangle = \langle \phi_A(P_B + [s_B]Q_B) \rangle = \langle \phi_A(B) \rangle$$

$$E_{AB} = \phi'_B(\phi_A(E_0)) \cong E_0/\langle P_A + [s_A]Q_A, P_B + [s_B]Q_B \rangle \cong E_{BA} = \phi'_A\ (\phi_B(E_0))$$

SIDH protocol

private Alice private Bob
public params

E's are isogenous curves
P's, Q's, R's, S's are points



$$A' = \langle \phi_B(P_A) + [s_A]\phi_B(Q_A) \rangle = \langle \phi_B(P_A + [s_A]Q_A) \rangle = \langle \phi_B(A) \rangle$$
$$B' = \langle \phi_A(P_B) + [s_B]\phi_A(Q_B) \rangle = \langle \phi_A(P_B + [s_B]Q_B) \rangle = \langle \phi_A(B) \rangle$$

$$E_{AB} = \phi'_B(\phi_A(E_0)) \cong E_0/\langle P_A + [s_A]Q_A, P_B + [s_B]Q_B \rangle \cong E_{BA} = \phi'_A\ (\phi_B(E_0))$$

SIDH protocol

Drawback:

- SIDH is not secure when keys are reused (Galbraith-Petit-Shani-Ti 2016)
  - Only recommended in **ephemeral mode**

Supersingular isogeny key encapsulation (SIKE)

- IND-CCA secure key encapsulation: no problem reusing keys!
- Uses a variant of Hofheinz–Hövelmanns–Kiltz (HHK) transform:

IND-CPA PKE → IND-CCA KEM

- HHK transform is secure in **both the classical and quantum ROM models**
- Offline key generation gives performance boost (no perf loss SIDH → SIKE)

Supersingular isogeny key encapsulation (SIKE)

KeyGen

1. $s_B \in_R [0, 2^{\lfloor \log_2 3^{e_B} \rfloor})$
2. Set $ker(\phi_B) = \langle P_B + [s_B]Q_B \rangle$
3. $\mathrm{pk}_B = \{\phi_B(E_0), \phi_B(P_A), \phi_B(Q_A)\}$
4. $s \in_R \{0,1\}^n$
5. **keypair:** $\{\mathrm{sk}_B = (s, s_B), \mathrm{pk}_B\}$

F, G, H instantiated with cSHAKE256.

Supersingular isogeny key encapsulation (SIKE)

KeyGen

1. $s_B \in_R [0, 2^{\lfloor \log_2 3^{e_B} \rfloor})$
2. Set $ker(\phi_B) = \langle P_B + [s_B]Q_B \rangle$
3. $\mathrm{pk}_B = \{\phi_B(E_0), \phi_B(P_A), \phi_B(Q_A)\}$
4. $s \in_R \{0,1\}^n$
5. **keypair:** $\{\mathrm{sk}_B = (s, s_B), \mathrm{pk}_B\}$

pk_B →

Encaps

1. message $m \in_R \{0,1\}^n$
2. $r = G(m, \mathrm{pk}_B) \bmod 2^{e_A}$
3. Set $ker(\phi_A) = \langle P_A + [r]Q_A \rangle$
4. $\mathrm{pk}_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
5. $j = j(E_{AB}) = j(\phi'_A(\phi_B(E_0)))$
6. **Shared key:** $ss = H(m, c)$

F, G, H instantiated with cSHAKE256.

Supersingular isogeny key encapsulation (SIKE)

KeyGen

1. $s_B \in_R [0, 2^{\lfloor \log_2 3^{e_B} \rfloor})$
2. Set $ker(\phi_B) = \langle P_B + [s_B]Q_B \rangle$
3. $\mathrm{pk}_B = \{\phi_B(E_0), \phi_B(P_A), \phi_B(Q_A)\}$
4. $s \in_R \{0,1\}^n$
5. **keypair:** $\{\mathrm{sk}_B = (s, s_B), \mathrm{pk}_B\}$

pk_B →

Encaps

1. message $m \in_R \{0,1\}^n$
2. $r = G(m, \mathrm{pk}_B) \bmod 2^{e_A}$

encryption

3. Set $ker(\phi_A) = \langle P_A + [r]Q_A \rangle$
4. $\mathrm{pk}_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
5. $j = j(E_{AB}) = j(\phi'_A(\phi_B(E_0)))$
6. **Shared key:** $ss = H(m, c)$

F, G, H instantiated with cSHAKE256.

Supersingular isogeny key encapsulation (SIKE)

KeyGen

1. $s_B \in_R [0, 2^{\lfloor \log_2 3^{e_B} \rfloor})$
2. Set $ker(\phi_B) = \langle P_B + [s_B]Q_B \rangle$
3. $\mathrm{pk}_B = \{\phi_B(E_0), \phi_B(P_A), \phi_B(Q_A)\}$
4. $s \in_R \{0,1\}^n$
5. **keypair:** $\{\mathrm{sk}_B = (s, s_B), \mathrm{pk}_B\}$

Decaps

1. $j' = j(E_{BA}) = j(\phi'_B(\phi_A(E_0)))$
2. $m' = F(j') \oplus c[2]$
3. $r' = G(m', \mathrm{pk}_B) \bmod 2^{e_A}$
4. Set $ker(\phi_A) = \langle P_A + [r']Q_A \rangle$
5. $\mathrm{pk}'_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
6. If $\mathrm{pk}'_A = c[1]$ then
   Shared key: $ss = H(m', c)$
7. Else $ss = H(s, c)$

pk_B →

← $c = (\mathrm{pk}_A, F(j) \oplus m)$

Encaps

1. message $m \in_R \{0,1\}^n$
2. $r = G(m, \mathrm{pk}_B) \bmod 2^{e_A}$

encryption

3. Set $ker(\phi_A) = \langle P_A + [r]Q_A \rangle$
4. $\mathrm{pk}_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
5. $j = j(E_{AB}) = j(\phi'_A(\phi_B(E_0)))$
6. **Shared key:** $ss = H(m, c)$

F, G, H instantiated with cSHAKE256.

Supersingular isogeny key encapsulation (SIKE)

KeyGen

1. $s_B \in_R [0, 2^{\lfloor \log_2 3^{e_B} \rfloor})$
2. Set $ker(\phi_B) = \langle P_B + [s_B]Q_B \rangle$
3. $\mathrm{pk}_B = \{\phi_B(E_0), \phi_B(P_A), \phi_B(Q_A)\}$
4. $s \in_R \{0,1\}^n$
5. **keypair:** $\{\mathrm{sk}_B = (s, s_B), \mathrm{pk}_B\}$

pk_B →

Encaps

1. message $m \in_R \{0,1\}^n$
2. $r = G(m, \mathrm{pk}_B) \bmod 2^{e_A}$ *encryption*
3. Set $ker(\phi_A) = \langle P_A + [r]Q_A \rangle$
4. $\mathrm{pk}_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
5. $j = j(E_{AB}) = j(\phi'_A(\phi_B(E_0)))$
6. **Shared key:** $ss = H(m, c)$

← $c = (\mathrm{pk}_A, F(j) \oplus m)$

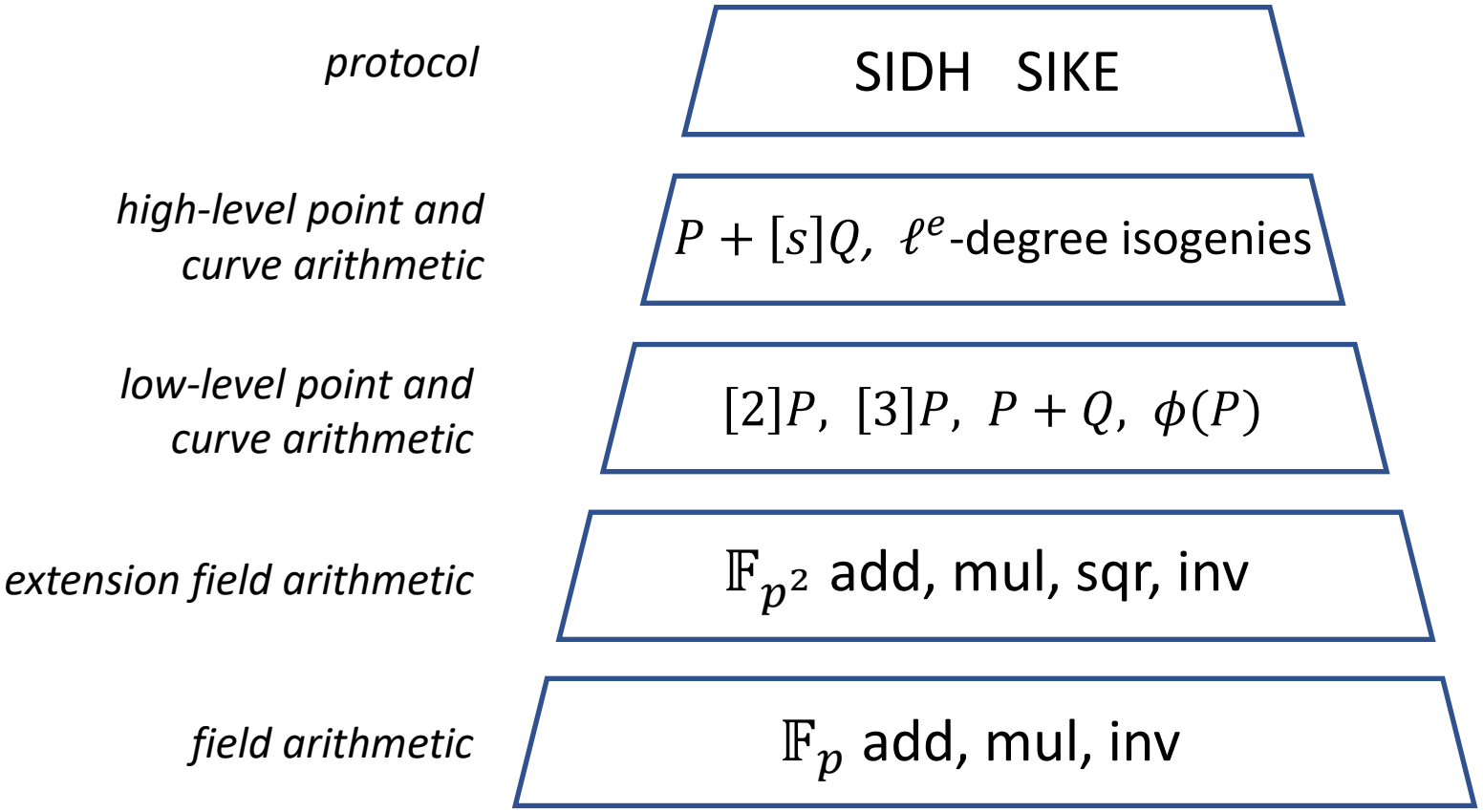
Decaps

1. $j' = j(E_{BA}) = j(\phi'_B(\phi_A(E_0)))$
2. $m' = F(j') \oplus c[2]$
3. $r' = G(m', \mathrm{pk}_B) \bmod 2^{e_A}$ *decryption*
4. Set $ker(\phi_A) = \langle P_A + [r']Q_A \rangle$
5. $\mathrm{pk}'_A = \{\phi_A(E_0), \phi_A(P_B), \phi_A(Q_B)\}$
6. If $\mathrm{pk}'_A = c[1]$ then *partial re-encryption*
   Shared key: $ss = H(m', c)$
7. Else $ss = H(s, c)$

F, G, H instantiated with cSHAKE256.

Computational aspects

Computation layers



Computational aspects:
High-level point and curve arithmetic

High-level point and curve arithmetic

Two main internal computations:

- **Double-scalar multiplications** to construct kernels $\langle P + [s]Q \rangle$
- **Smooth, ℓ^e-degree isogeny** computations ϕ: $E_0 \rightarrow E'$

Computing $P + [s]Q$

Three-point differential ladder (x-only, variable point)

- De Feo-Jao-Plût (2014), step cost = **1DBL + 2ADD**
- Faz-Hernández et al. (2018), step cost = **1DBL + 1ADD**

Computing $P + [s]Q$

[Faz-Hernández–López–Ochoa-Jiménez–Rodríguez-Henríquez 2018]

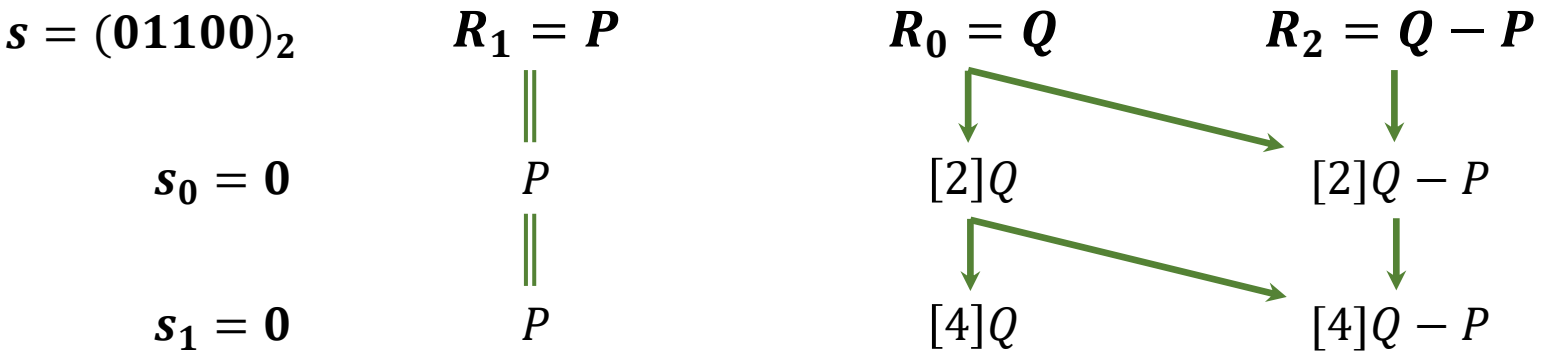
$\boldsymbol{s = (01100)_2}$ $\quad$ $\boldsymbol{R_1 = P}$ $\quad$ $\boldsymbol{R_0 = Q}$ $\quad$ $\boldsymbol{R_2 = Q - P}$

Computing $P + [s]Q$

[Faz-Hernández–López–Ochoa-Jiménez–Rodríguez-Henríquez 2018]

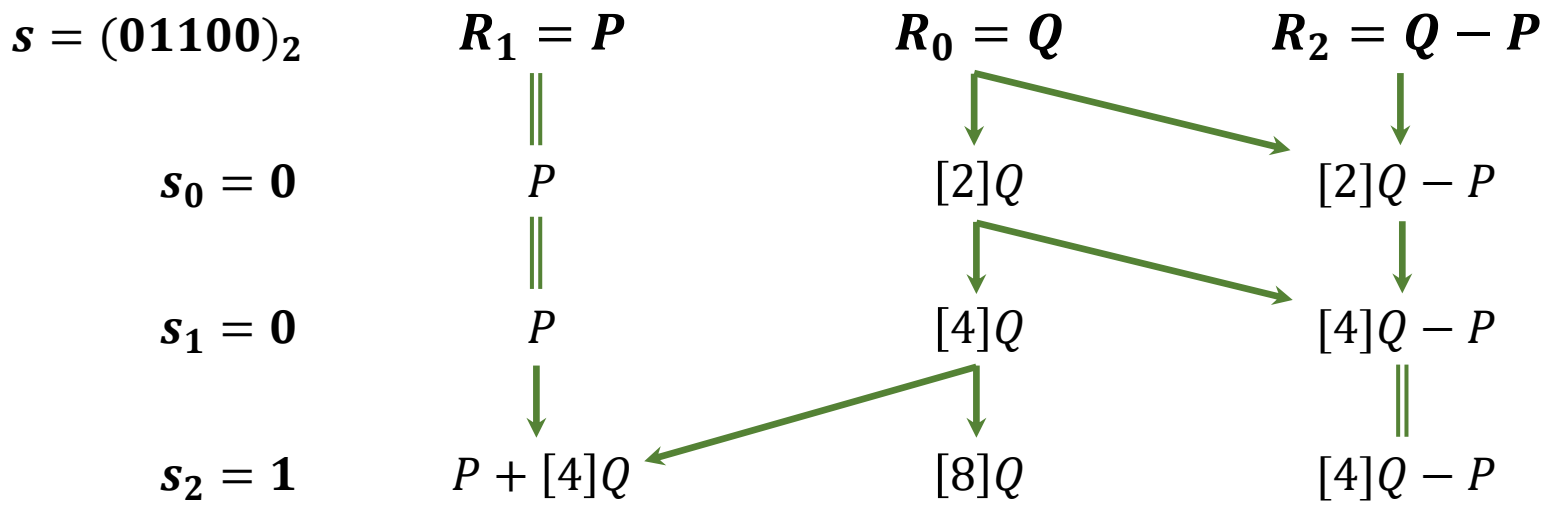
Computing $P + [s]Q$

[Faz-Hernández–López–Ochoa-Jiménez–Rodríguez-Henríquez 2018]



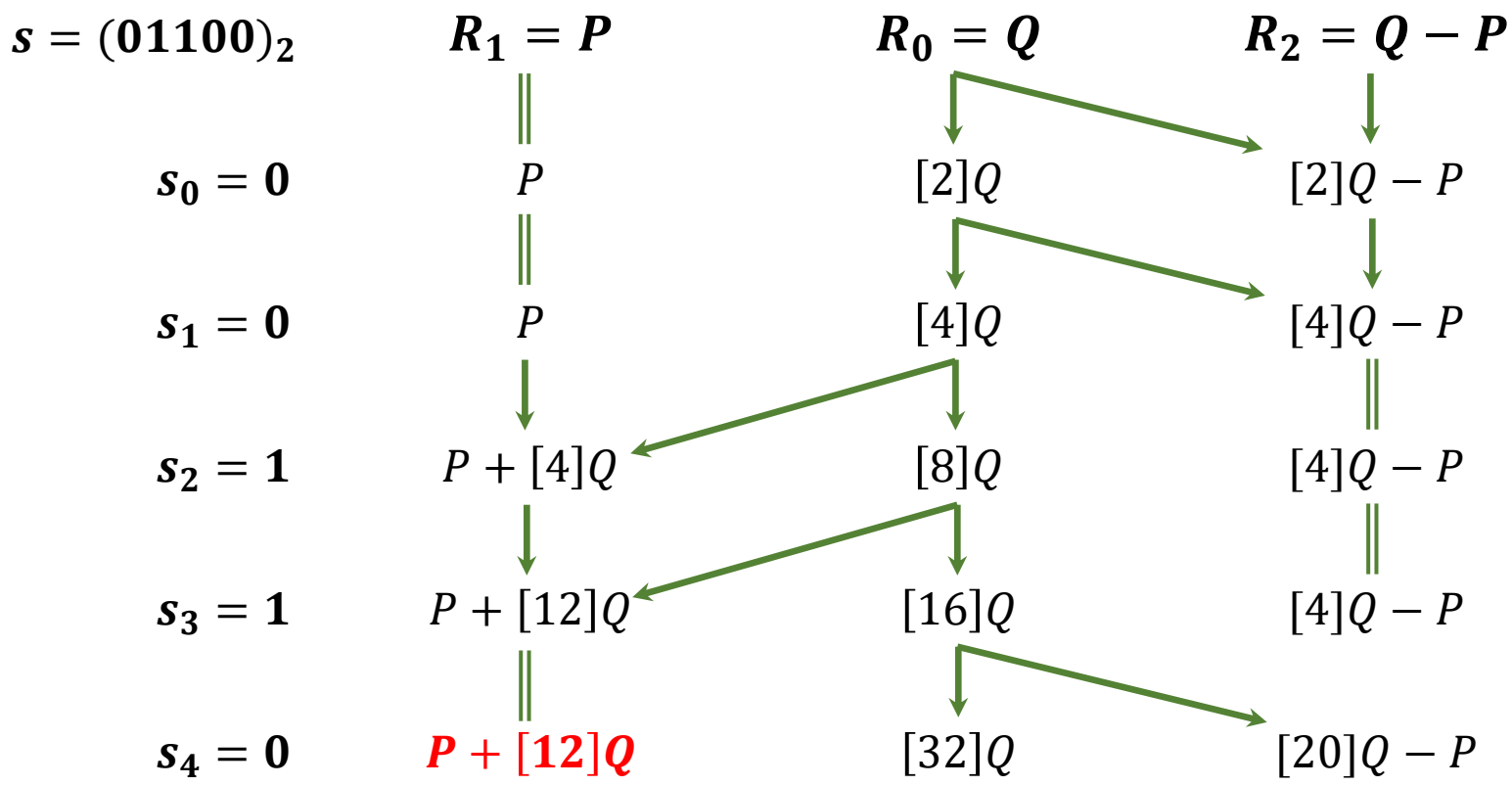
Computing $P + [s]Q$

[Faz-Hernández–López–Ochoa-Jiménez–Rodríguez-Henríquez 2018]



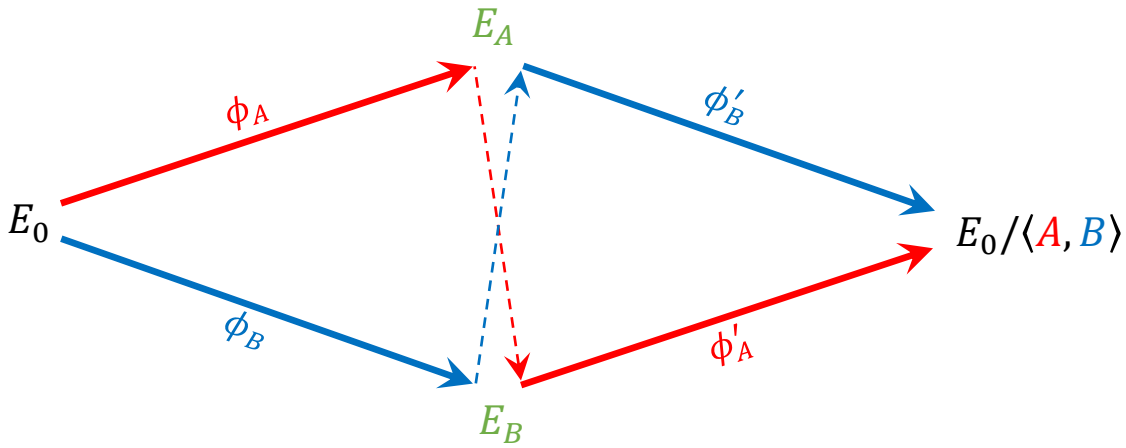
Computing $P + [s]Q$

[Faz-Hernández–López–Ochoa-Jiménez–Rodríguez-Henríquez 2018]



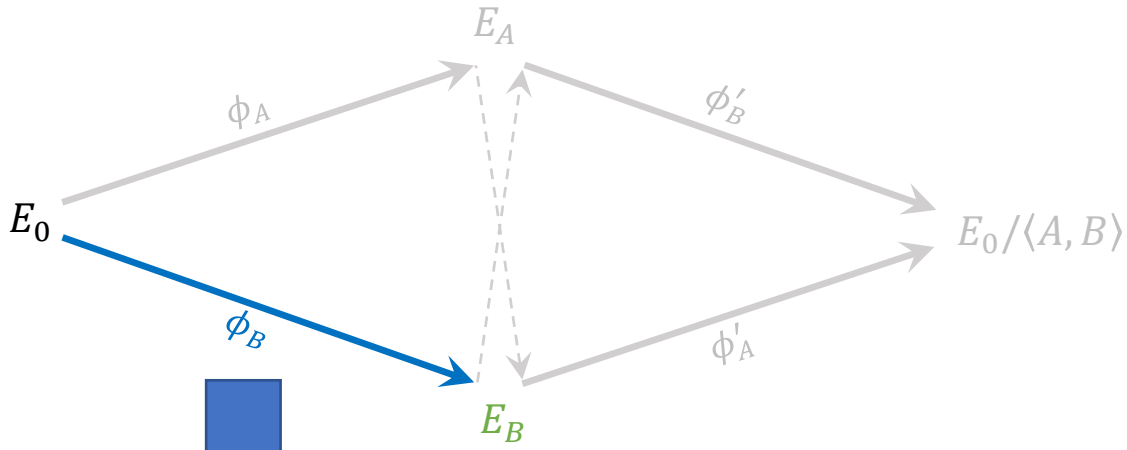
Computing ℓ^e-degree isogenies

- Construct it as a composition of multiple (small, prime-degree) isogenies



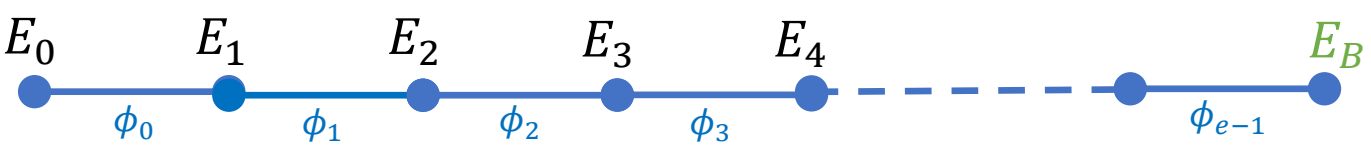
Computing ℓ^e-degree isogenies

- Construct it as a composition of multiple (small, prime-degree) isogenies



$$\phi_B \colon E_0 \to E_B$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \cdots \cdot \phi_{e-1}$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

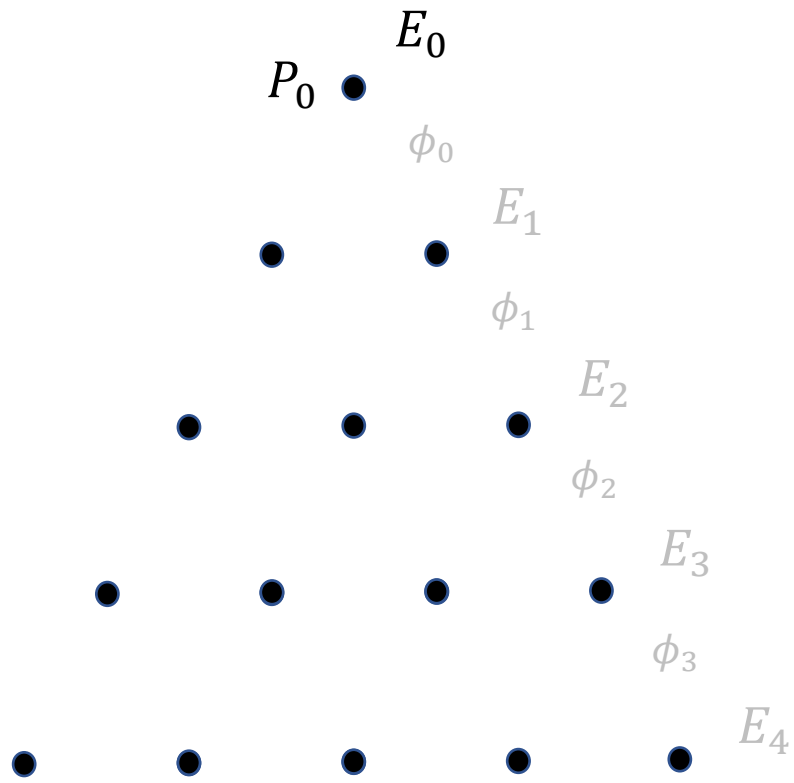
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



- Iteratively compute:

$$E_{i+1} = E_i/\langle [\ell^{e-i-1}]P_i \rangle$$

Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

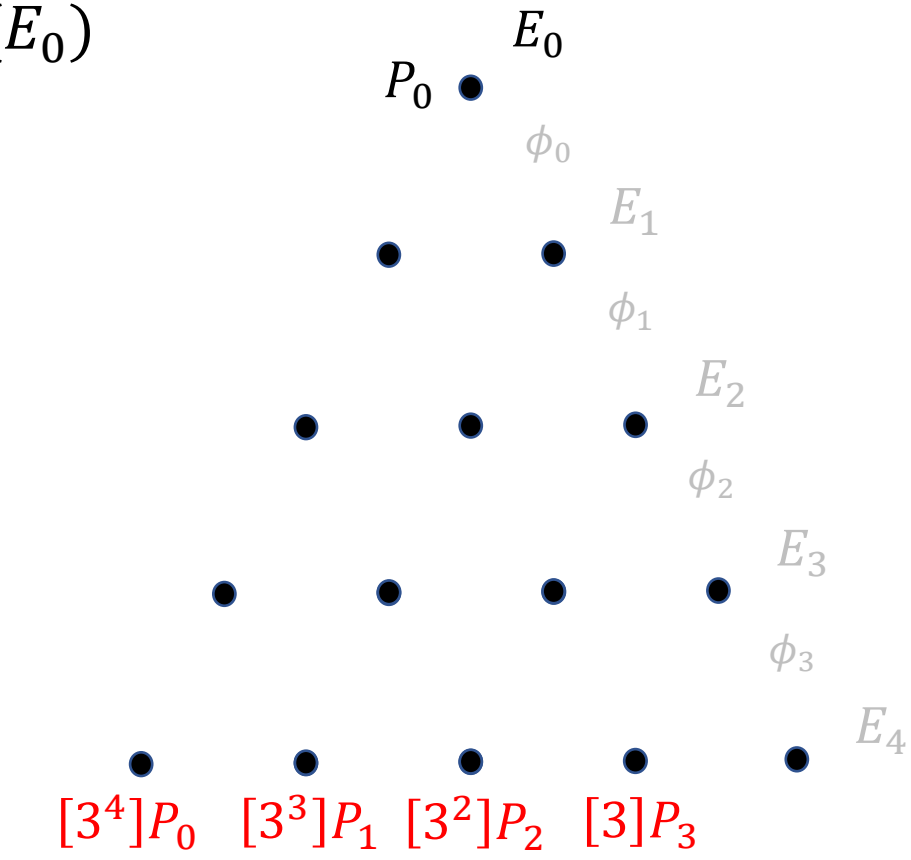
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



- Iteratively compute:

$$E_{i+1} = E_i/\langle [\ell^{e-i-1}]P_i \rangle$$

Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

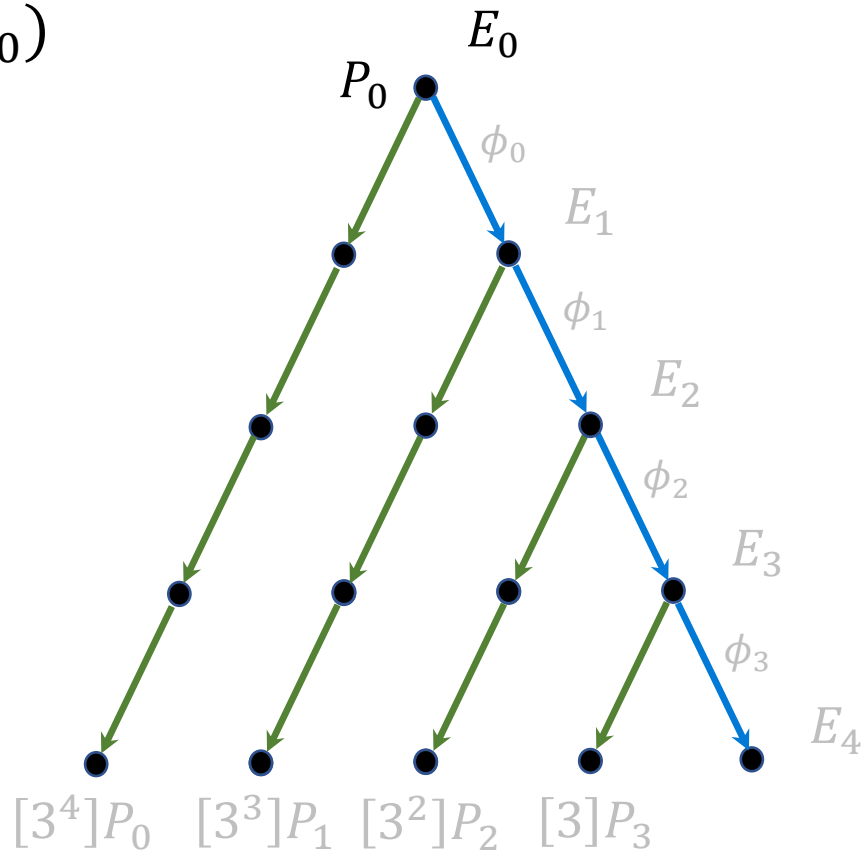
$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

- Iteratively compute:

$$E_{i+1} = E_i/\langle [\ell^{e-i-1}]P_i \rangle$$



(+) slope: point operations

(−) slope: isogeny operations

Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

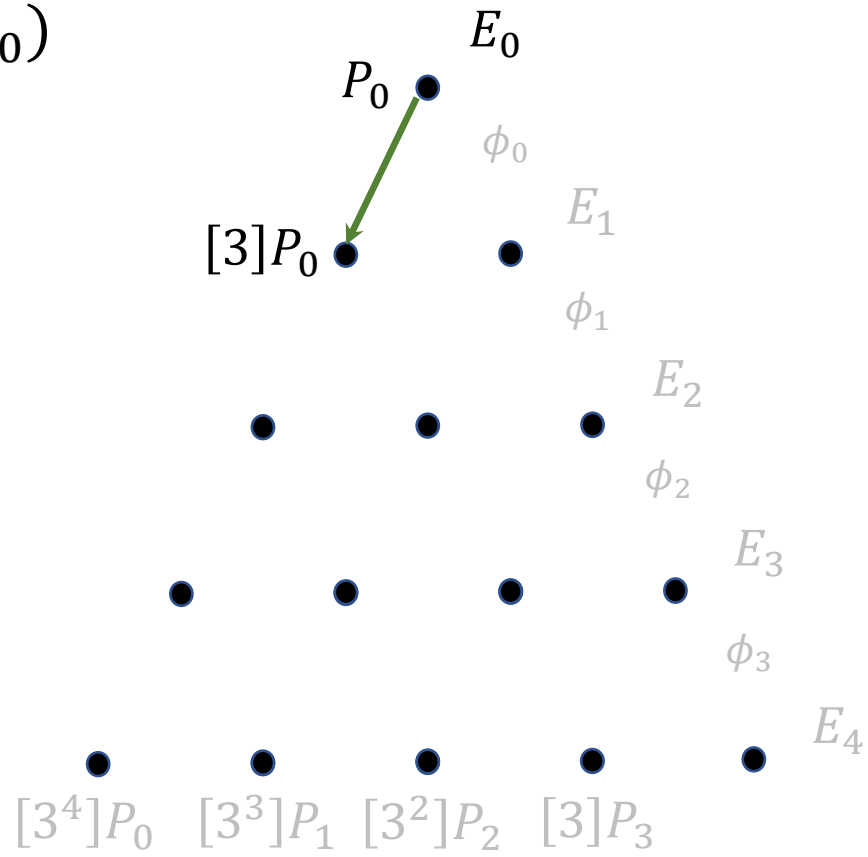
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

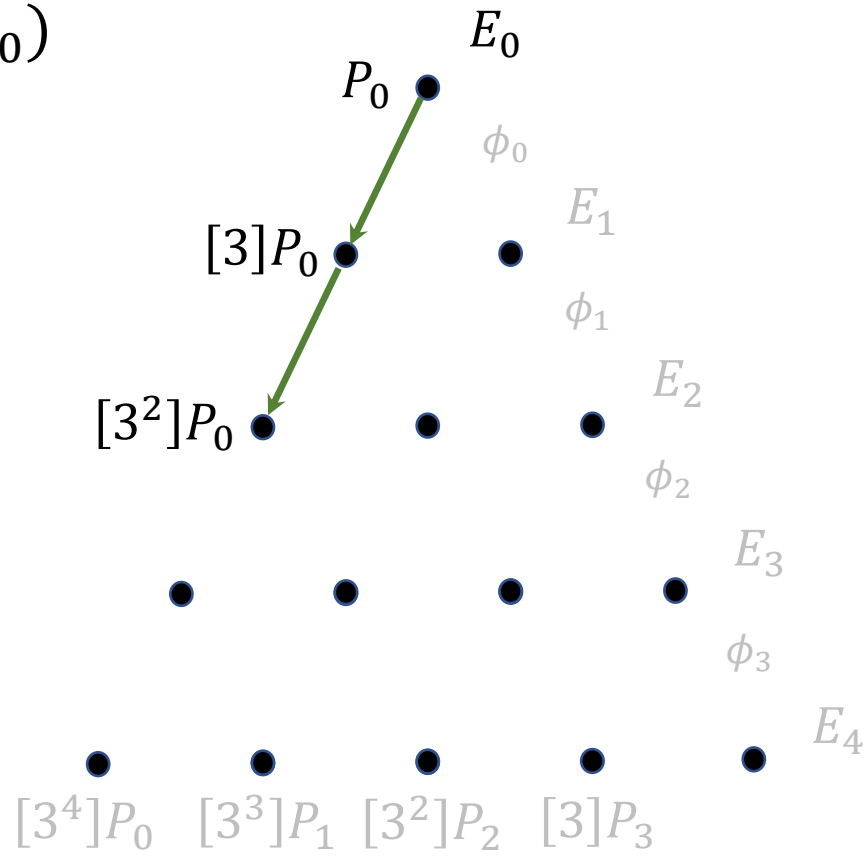
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

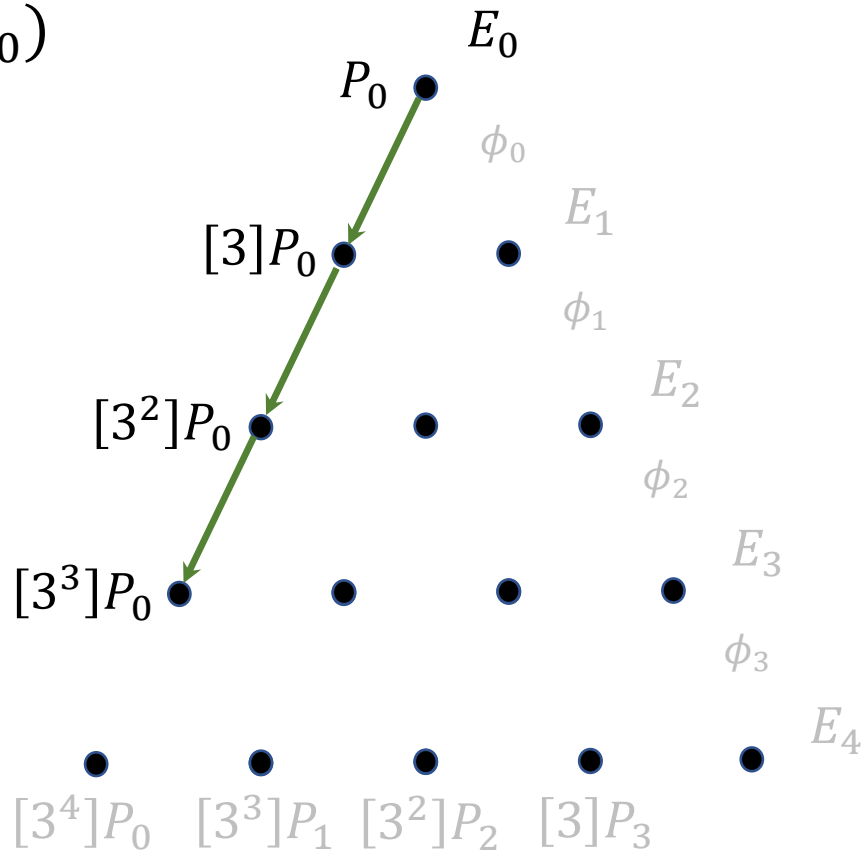
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

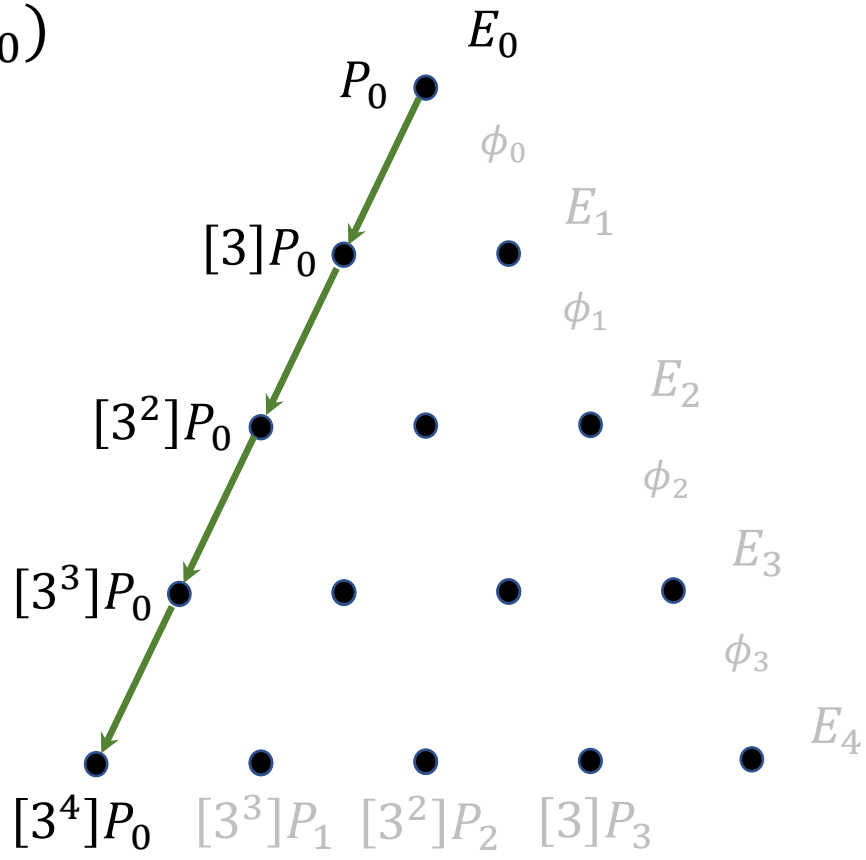
  Let base point $P_0 \in E_0$. Assume $e = 4$

  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

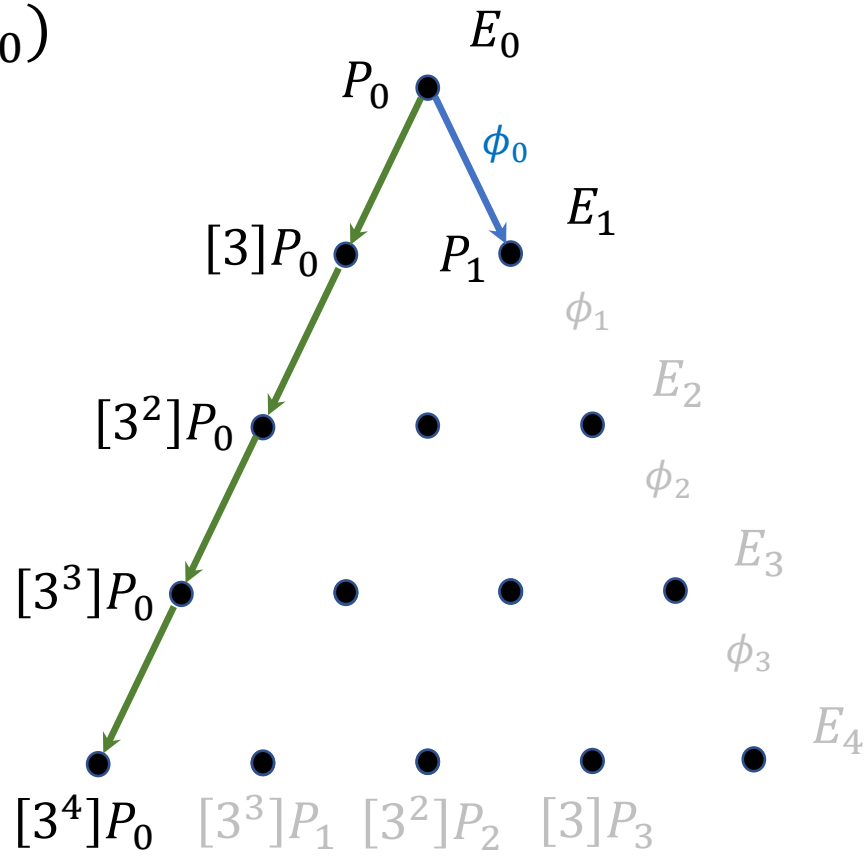
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_0 = E_0/\langle 81P_0 \rangle$$
$$E_1 = \phi_0(E_0)$$
$$P_1 = \phi_0(P_0)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

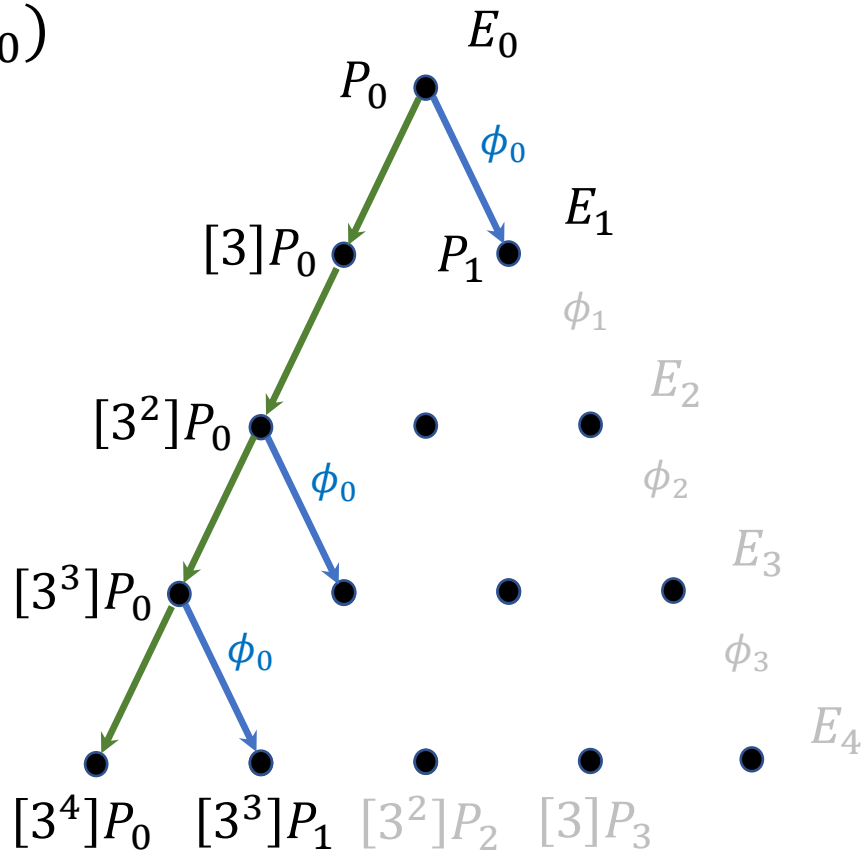
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_0 = E_0/\langle 81P_0 \rangle$$
$$E_1 = \phi_0(E_0)$$
$$P_1 = \phi_0(P_0)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

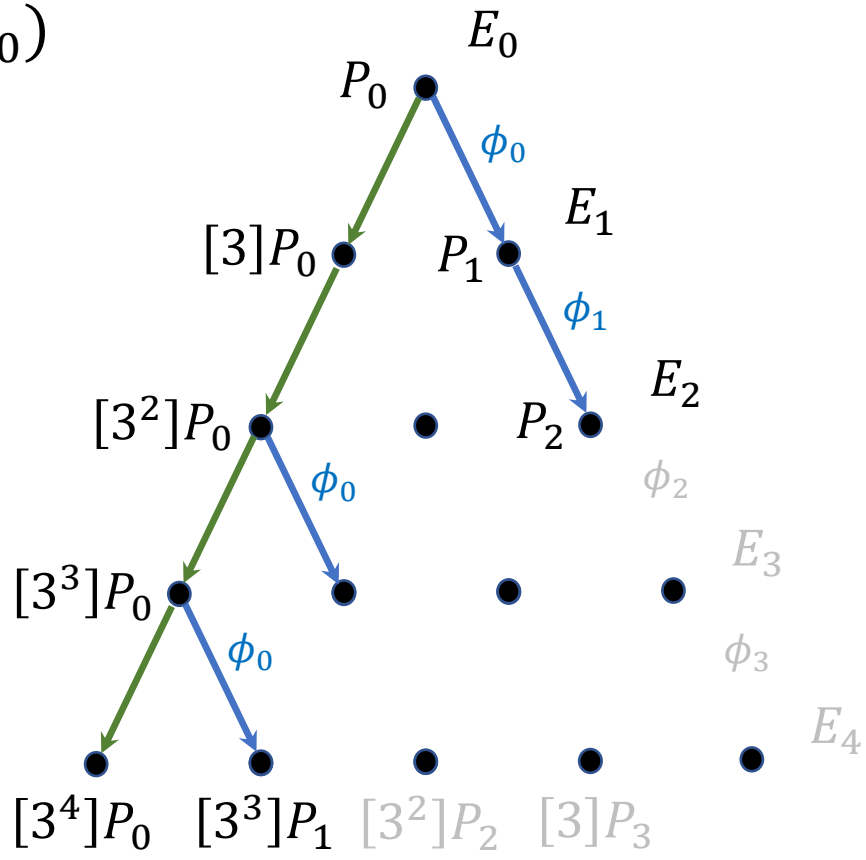
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_1 = E_1/\langle 27P_0 \rangle$$
$$E_2 = \phi_1(E_1)$$
$$P_2 = \phi_1(P_1)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

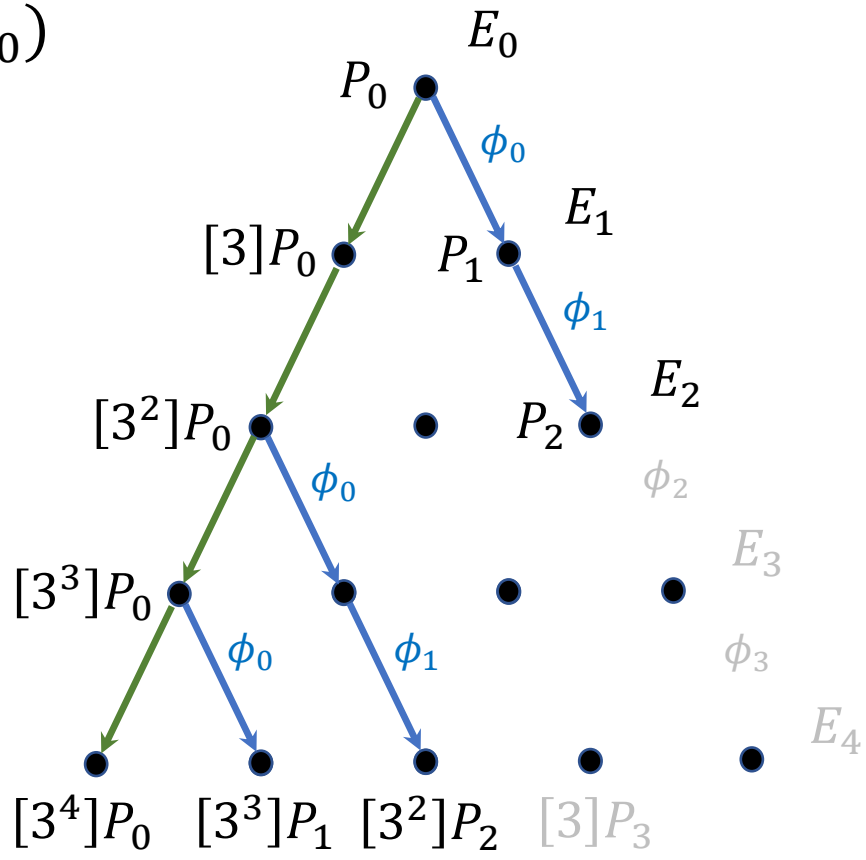
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_1 = E_1/\langle 27P_0 \rangle$$
$$E_2 = \phi_1(E_1)$$
$$P_2 = \phi_1(P_1)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

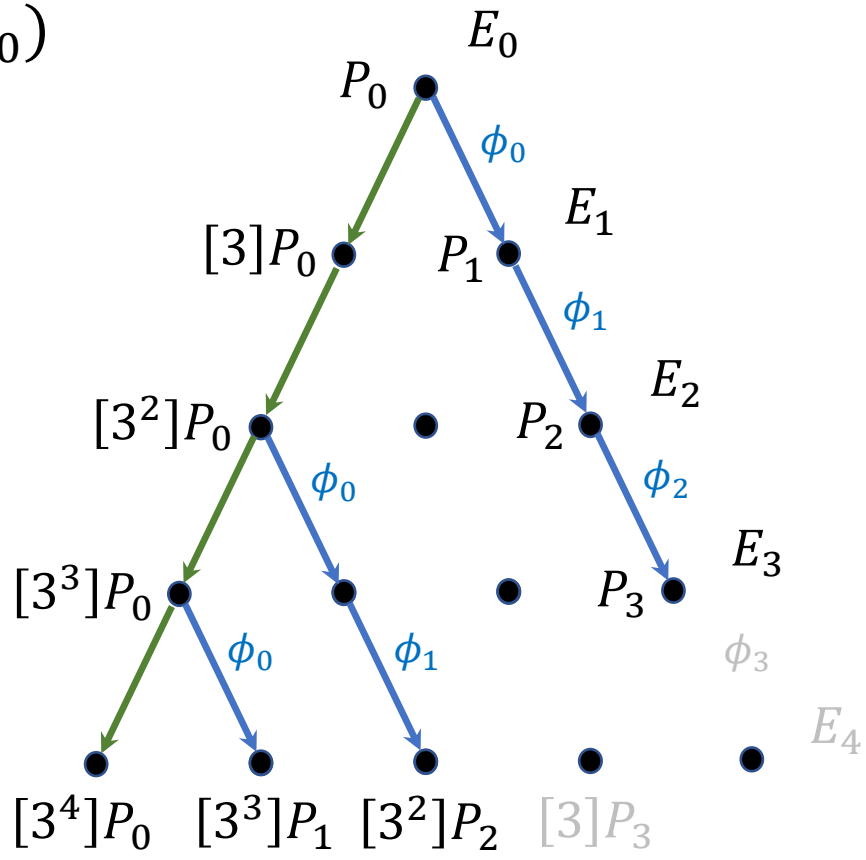
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_2 = E_2/\langle 9P_2 \rangle$$
$$E_3 = \phi_2(E_2)$$
$$P_3 = \phi_2(P_2)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

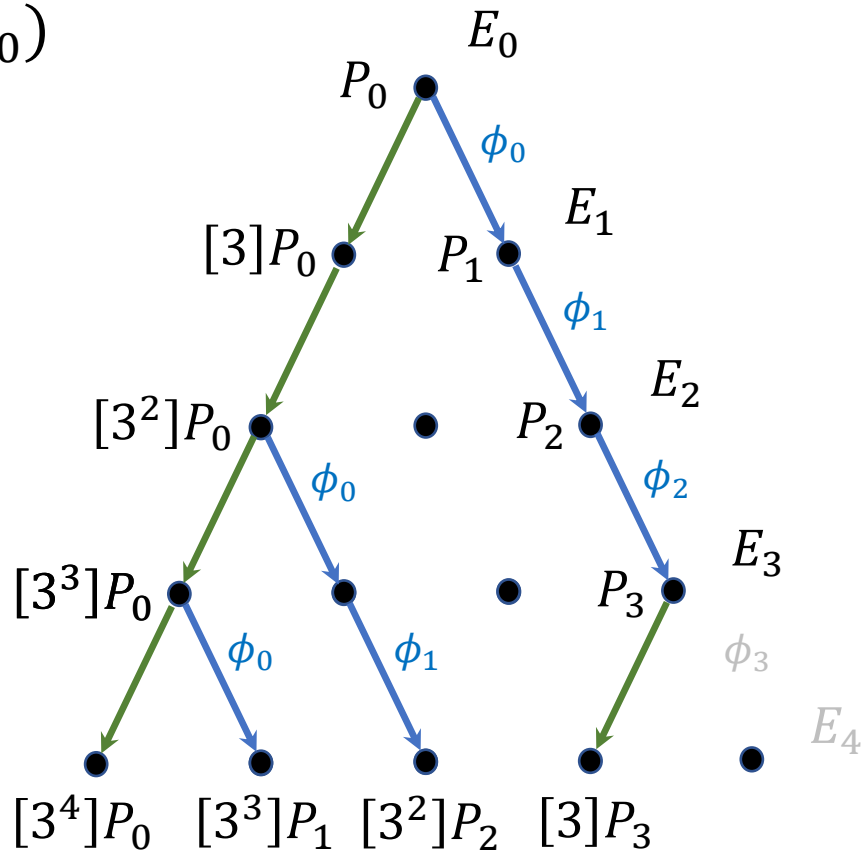
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

$$\phi_2 = E_2/\langle 9P_2 \rangle$$
$$E_3 = \phi_2(E_2)$$
$$P_3 = \phi_2(P_2)$$



Computing ℓ^e-degree isogenies

- **Example:** Bob ($\ell = 3$) computes $E_B = \phi_B(E_0)$

  Let base point $P_0 \in E_0$. Assume $e = 4$

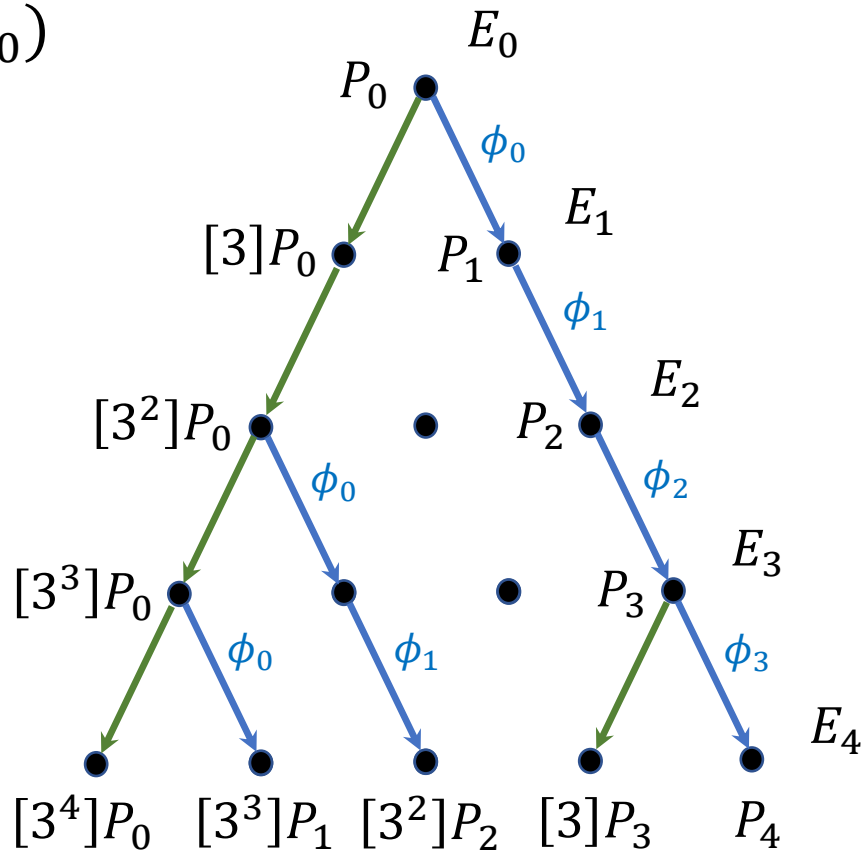
  Compute 3^4-degree isogeny:

$$\phi_B \colon E_0 \to E_4$$

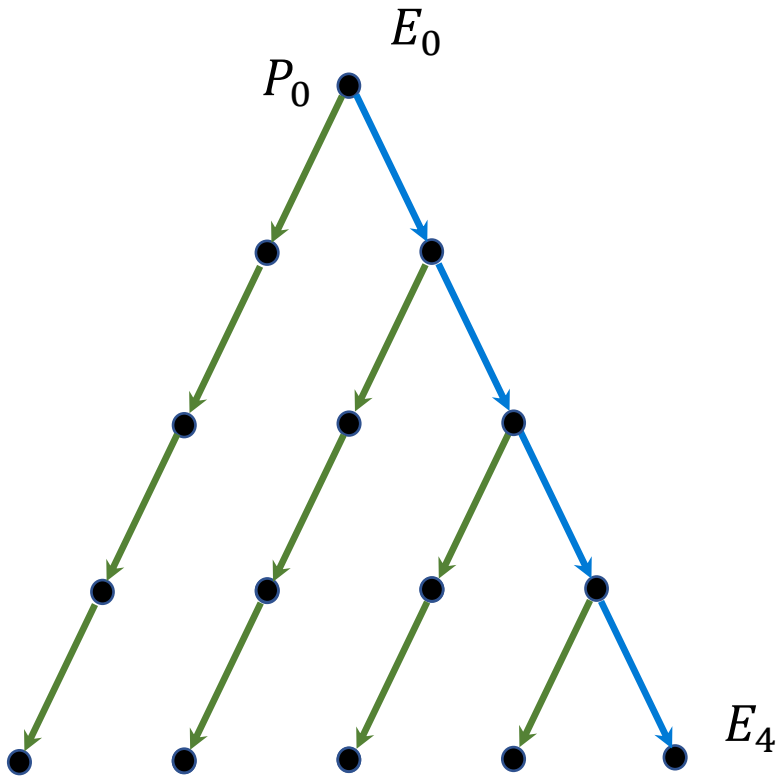
$$\phi_B = \phi_0 \cdot \phi_1 \cdot \phi_2 \cdot \phi_3$$

$$E_4 = E_0/\langle P_0 \rangle$$

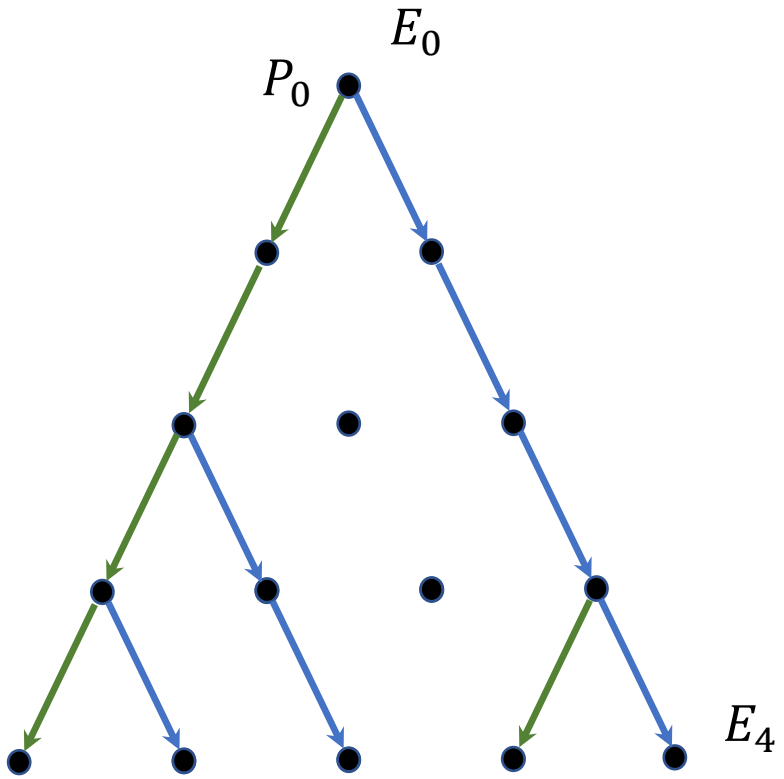
$$\phi_3 = E_3/\langle 3P_3 \rangle$$
$$E_4 = \phi_3(E_3)$$
$$P_4 = \phi_3(P_3)$$



Computing ℓ^e-degree isogenies



naïve



optimal

Optimal strategy: reduction from $\mathcal{O}(e^2)$ operations to $\mathcal{O}(e \log e)$

Computational aspects: Field arithmetic

Extension field arithmetic

Constructing degree-2 extension field $\mathbb{F}_{p^2}$ of a finite field $\mathbb{F}_p$:

Fix $\mathbb{F}_{p^2} = \mathbb{F}_p(\alpha)$, with degree-2 irreducible polynomial $f(x)$ in $\mathbb{F}_p[x]$ s.t. $f(\alpha) = 0$

Extension field arithmetic

Constructing degree-2 extension field $\mathbb{F}_{p^2}$ of a finite field $\mathbb{F}_p$:

Fix $\mathbb{F}_{p^2} = \mathbb{F}_p(\alpha)$, with degree-2 irreducible polynomial $f(x)$ in $\mathbb{F}_p[x]$ s.t. $f(\alpha) = 0$

In our case: for a prime $p \equiv 3 \bmod 4$, take $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b$
- $a \times b$
- a^2
- a^{-1}

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b = (a_0 \pm b_0, a_1 \pm b_1)$

  cost: 2 $\mathbb{F}_p$ add/sub

- $a \times b$

- a^2

- a^{-1}

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b = (a_0 \pm b_0, a_1 \pm b_1)$

  cost: 2 $\mathbb{F}_p$ add/sub

- $a \times b = (a_0 \times b_0 - a_1 \times b_1, a_0 \times b_1 + a_1 \times b_0)$

  cost: 4 $\mathbb{F}_p$ mul + 2 $\mathbb{F}_p$ add/sub

- a^2

- a^{-1}

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b = (a_0 \pm b_0, a_1 \pm b_1)$

  cost: 2 $\mathbb{F}_p$ add/sub

- $a \times b = (a_0 \times b_0 - a_1 \times b_1, a_0 \times b_1 + a_1 \times b_0)$

  cost: 4 $\mathbb{F}_p$ mul + 2 $\mathbb{F}_p$ add/sub

  $= (a_0 \times b_0 - a_1 \times b_1, (a_0 + a_1) \times (b_0 + b_1) - a_0 \times b_0 - a_1 \times b_1)$

  cost: 3 $\mathbb{F}_p$ mul + 5 $\mathbb{F}_p$ add/sub *or* 3 mul + 5 add/sub + 2 rdc

- a^2

- a^{-1}

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b = (a_0 \pm b_0, a_1 \pm b_1)$

  cost: 2 $\mathbb{F}_p$ add/sub

- $a \times b = (a_0 \times b_0 - a_1 \times b_1, a_0 \times b_1 + a_1 \times b_0)$

  cost: 4 $\mathbb{F}_p$ mul + 2 $\mathbb{F}_p$ add/sub

  $= (a_0 \times b_0 - a_1 \times b_1, (a_0 + a_1) \times (b_0 + b_1) - a_0 \times b_0 - a_1 \times b_1)$

  cost: 3 $\mathbb{F}_p$ mul + 5 $\mathbb{F}_p$ add/sub *or* 3 mul + 5 add/sub + 2 rdc

- $a^2 = \big((a_0 + a_1) \times (a_0 - a_1), 2a_0 \times a_1\big)$

  cost: 2 $\mathbb{F}_p$ mul + 3 $\mathbb{F}_p$ add/sub

- a^{-1}

Extension field arithmetic

Assume $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$, with $i^2 + 1 = 0$

Let $a = (a_0, a_1), b = (b_0, b_1) \in \mathbb{F}_{p^2}$, then:

- $a \pm b = (a_0 \pm b_0, a_1 \pm b_1)$

  cost: 2 $\mathbb{F}_p$ add/sub

- $a \times b = (a_0 \times b_0 - a_1 \times b_1, a_0 \times b_1 + a_1 \times b_0)$

  cost: 4 $\mathbb{F}_p$ mul + 2 $\mathbb{F}_p$ add/sub

  $= (a_0 \times b_0 - a_1 \times b_1, (a_0 + a_1) \times (b_0 + b_1) - a_0 \times b_0 - a_1 \times b_1)$

  cost: 3 $\mathbb{F}_p$ mul + 5 $\mathbb{F}_p$ add/sub *or* 3 mul + 5 add/sub + 2 rdc

- $a^2 = \left((a_0 + a_1) \times (a_0 - a_1), 2a_0 \times a_1\right)$

  cost: 2 $\mathbb{F}_p$ mul + 3 $\mathbb{F}_p$ add/sub

- $a^{-1} = \left(a_0 \times \left(a_0^2 + a_1^2\right)^{-1}, -a_1 \times \left(a_0^2 + a_1^2\right)^{-1}\right)$

  cost: 2 $\mathbb{F}_p$ mul + 2 $\mathbb{F}_p$ sqr + 1 $\mathbb{F}_p$ add + 1 $\mathbb{F}_p$ neg + 1 $\mathbb{F}_p$ inv

Field multiplication

- Two main approaches to implement integer multiplication + reduction:

  separated (non-interleaved) or *integrated (interleaved)*

Field multiplication

- Two main approaches to implement integer multiplication + reduction:

  separated (non-interleaved) or *integrated (interleaved)*

- Separated (integer multiplication and reduction) approach is preferred in most software platforms
  - Enables lazy reduction at $\mathbb{F}_{p^2}$ level

Integer multiplication

- At SIDH/SIKE sizes, multi-level implementation is typically best
  - Karatsuba at highest levels
  - Schoolbook or Comba at lowest levels

Integer multiplication

- At SIDH/SIKE sizes, multi-level implementation is typically best
  - Karatsuba at highest levels
  - Schoolbook or Comba at lowest levels

Some representative cases:

- x64: limited number of registers, availability of carry-preserving instructions (e.g., `mulx`)
  - Can use one-level Karatsuba (top), two-level schoolbook (bottom)

Integer multiplication

- At SIDH/SIKE sizes, multi-level implementation is typically best
  - Karatsuba at highest levels
  - Schoolbook or Comba at lowest levels

Some representative cases:

- x64: limited number of registers, availability of carry-preserving instructions (e.g., `mulx`)
  - Can use one-level Karatsuba (top), two-level schoolbook (bottom)
- 64-bit ARMv8: plenty of registers, relatively expensive `mul`
  - Can use two-level Karatsuba (top), one-level Comba (bottom)

Modular reduction

- SIDH primes are amenable for a simplified Montgomery reduction

Modular reduction

- SIDH primes are amenable for a simplified Montgomery reduction
- Take $p = 2^{216} \cdot 3^{137} - 1$
- Let $R = 2^{448},\ p' = -p^{-1} \bmod R$

Modular reduction

- SIDH primes are amenable for a simplified Montgomery reduction
- Take $p = 2^{216} \cdot 3^{137} - 1$
- Let $R = 2^{448},\ p' = -p^{-1} \bmod R$
- Then, for an input $a < pR$:

$$c = (a + (ap' \bmod R) \cdot p)/R$$
$$c = (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137} - (ap' \bmod 2^{448}))/2^{448}$$
$$c = \left\lfloor (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137})/2^{448} \right\rfloor$$

Modular reduction

- SIDH primes are amenable for a simplified Montgomery reduction
- Take $p = 2^{216} \cdot 3^{137} - 1$
- Let $R = 2^{448},\ p' = -p^{-1} \bmod R$
- Then, for an input $a < pR$:

$$c = (a + (ap' \bmod R) \cdot p)/R$$
$$c = (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137} - (ap' \bmod 2^{448}))/2^{448}$$
$$c = \left\lfloor (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137})/2^{448} \right\rfloor$$

Modular reduction

- SIDH primes are amenable for a simplified Montgomery reduction
- Take $p = 2^{216} \cdot 3^{137} - 1$
- Let $R = 2^{448},\ p' = -p^{-1} \bmod R$
- Then, for an input $a < pR$:

$$c = (a + (ap' \bmod R) \cdot p)/R$$
$$c = (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137} - (ap' \bmod 2^{448}))/2^{448}$$
$$c = \left\lfloor (a + (ap' \bmod 2^{448}) \cdot 2^{216} \cdot 3^{137})/2^{448} \right\rfloor$$

Also: $p' \bmod 2^w \equiv 1$ for $w = 32, 64$

Security and parameters

SIDH security

Setting: supersingular curves $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, a large prime p, and isogeny $\phi: E_1 \to E_2$ with fixed, smooth, public degree.

> ***Supersingular isogeny problem:*** given $P, Q \in E_1$ and $\phi(P), \phi(Q) \in E_2$, compute ϕ.

SIDH security

Setting: supersingular curves $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, a large prime p, and isogeny $\phi: E_1 \to E_2$ with fixed, smooth, public degree.

> ***Supersingular isogeny problem:*** given $P, Q \in E_1$ and $\phi(P), \phi(Q) \in E_2$, compute ϕ.

- Adj–Cervantes-Vázquez–Chi-Domínguez–Menezes–Rodríguez-Henríquez (2018): *best classical attack* is **van Oorschot–Wiener (vOW) collision finding algorithm.**

SIDH security

Setting: supersingular curves $E_1/\mathbb{F}_{p^2}$ and $E_2/\mathbb{F}_{p^2}$, a large prime p, and isogeny $\phi: E_1 \to E_2$ with fixed, smooth, public degree.

> ***Supersingular isogeny problem:*** given $P, Q \in E_1$ and $\phi(P), \phi(Q) \in E_2$, compute ϕ.

- Adj–Cervantes-Vázquez–Chi-Domínguez–Menezes–Rodríguez-Henríquez (2018): *best classical attack* is **van Oorschot–Wiener (vOW) collision finding algorithm.**

  For SIDH/SIKE: number of order-$\ell^{e/2}$ subgroups of $E[\ell^e] = |S| \approx p^{1/4}$

  Assume storage $w \approx 2^{80}$

$$\mathcal{O}\left(\frac{|S|^{\frac{3}{2}}}{\sqrt{w}}\right)$$

SIKE parameters (round 2 and 3)

- Eight parameter sets submitted to NIST:
  - SIKEp434, SIKEp503, SIKEp610, SIKEp751, and their corresponding compressed variants

Scheme (SIKEp + $\lceil \log_2 p \rceil$)	(e_A, e_B)	Security level	Standard (bytes)		Compressed (bytes)	
			pk	ct	pk	ct
SIKEp434	(216,137)	AES-128 (level 1)	330	346	196	209
SIKEp503	(250,159)	SHA3-256 (level 2)	378	402	224	248
SIKEp610	(305,192)	AES-192 (level 3)	462	486	273	297
SIKEp751	(372,239)	AES-256 (level 5)	564	596	331	363

Starting curve $E_0/\mathbb{F}_{p^2}$: $y^2 = x^3 + 6x^2 + x$, where $p = 2^{e_A}3^{e_B} - 1$.

Implementation results

SIDH Library

- Current release: **version 3.3**

  https://github.com/Microsoft/PQCrypto-SIDH

- Implements **SIDH** and **SIKE** with the *four standard parameter sets*: SIDH/SIKE{p434, p503, p610, p751} and their *four compressed* counterparts

SIDH Library

- Current release: **version 3.3**

  https://github.com/Microsoft/PQCrypto-SIDH

- Implements **SIDH** and **SIKE** with the *four standard parameter sets*: SIDH/SIKE{p434, p503, p610, p751} and their *four compressed* counterparts
- Included implementations:
  - Portable C
  - High-performance 64-bit CPUs
    - With high-speed x64 assembly for the field arithmetic
    - With high-speed ARMv8-A assembly for the field arithmetic

SIDH Library

- Current release: **version 3.3**

  https://github.com/Microsoft/PQCrypto-SIDH

- Implements **SIDH** and **SIKE** with the *four standard parameter sets*: SIDH/SIKE{p434, p503, p610, p751} and their *four compressed* counterparts
- Included implementations:
  - Portable C
  - High-performance 64-bit CPUs
    - With high-speed x64 assembly for the field arithmetic
    - With high-speed ARMv8-A assembly for the field arithmetic
- No secret branches, no secret memory accesses: protected against cache and timing attacks

Performance on x64 (SIDH v3.3)

- Total time: Encapsulation + Decapsulation

Primitive	Standard		Compressed	
	Cycles ($\times 10^6$)	Time	Cycles ($\times 10^6$)	Time
SIKEp434	20.0	5.9 ms	26.2	7.7 ms
SIKEp503	27.9	8.2 ms	36.3	10.7 ms
SIKEp610	54.7	16.1 ms	66.9	19.7 ms
SIKEp751	84.6	24.9 ms	110.1	32.4 ms

(*) Obtained on a 3.4GHz Intel Core i7-6700 (Skylake) processor.

Performance on 64-bit ARM (SIDH v3.3)

Primitive	NIST sec level	Cycles ($\times 10^6$)	Time @1.992GHz
Passively secure key-exchange			
SIDHp434	1	60.8	30.5 ms
SIDHp503	2	88.4	44.4 ms
IND-CCA secure KEMs			
SIKEp434	1	59.4	29.8 ms
SIKEp503	2	82.7	41.5 ms

(*) Obtained on a 1.992GHz ARM Cortex-A72 (ARMv8-A) processor.

Performance on x64

Primitive	PQ security	Problem	Speed	Comm.
Classical				
RSA 3072	~0 bits	factoring	4.6 ms	0.8 KB
ECDH NIST P-256	~0 bits	EC dlog	1.4 ms	0.1 KB
Passively secure key-exchange				
SIDHp434	128 bits	isogenies	6.0 ms	0.6 KB
IND-CCA secure KEMs				
Kyber	100 bits	M-LWE	0.03 ms	0.7 KB
FrodoKEM	108 bits	LWE	1.1 ms	9.5 KB
SIKEp434	128 bits	isogenies	5.9–7.7 ms	0.2–0.3 KB

(*) Obtained on 3.4GHz Intel Haswell (Kyber) or Skylake (FrodoKEM and SIKE).

Recent research

The Cost to Break SIKE: A Comparative Hardware-Based Analysis with AES and SHA-3, L-Wang-Szefer, 2020. https://eprint.iacr.org/2020/1457

- Uses a budget-based cost model to estimate cryptanalysis cost of vOW versus brute forcing AES.

Recent research

The Cost to Break SIKE: A Comparative Hardware-Based Analysis with AES and SHA-3, L-Wang-Szefer, 2020. https://eprint.iacr.org/2020/1457

- Uses a budget-based cost model to estimate cryptanalysis cost of vOW versus brute forcing AES.
- E.g., suggests SIKEp377 to match AES128 security (level 1).
  - ~1.4x faster than SIKEp434, ~13% smaller public keys

SIKE in the NIST post-quantum “competition”

- SIKE website: http://sike.org/
- SIKE specification: http://sike.org/files/SIDH-spec.pdf
- SIDH Library: https://github.com/Microsoft/PQCrypto-SIDH
- Package (protocol specification and implementations) submitted to NIST:

https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/SIKE-Round3.zip

Software Implementation of (Post-Quantum) Public-Key Cryptography

Patrick Longa

Microsoft Research

https://microsoft.com/en-us/research/people/plonga
http://patricklonga.com
Twitter: @PatrickLonga

SPACE 2020

Virtual Conference