ARCHI 2017, Nancy, France — March 6–10, 2017

Hardware Implementation of Cryptography

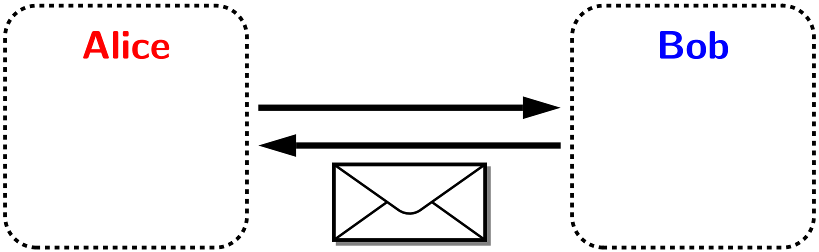
Jérémie Detrey
CARAMBA team, LORIA
INRIA Nancy – Grand Est, France
Jeremie.Detrey@loria.fr

ARCHI 2017, Nancy, France — March 6–10, 2017

Hardware Implementation of (Elliptic Curve) Cryptography

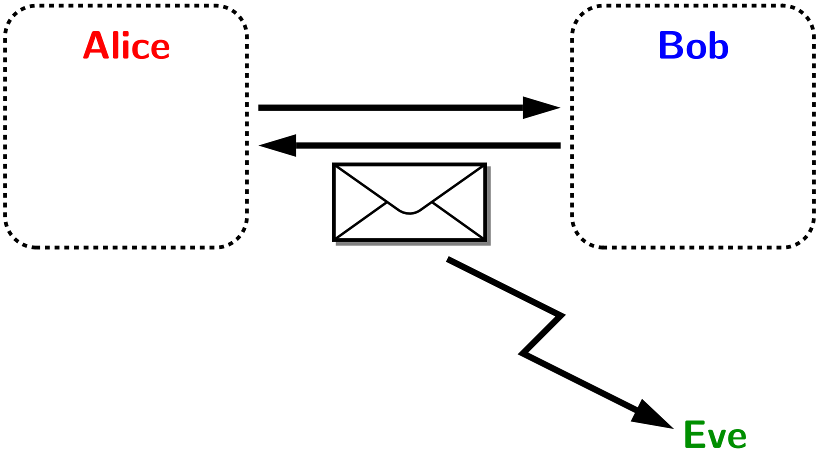
Jérémie Detrey
CARAMBA team, LORIA
INRIA Nancy – Grand Est, France
Jeremie.Detrey@loria.fr

Cryptography



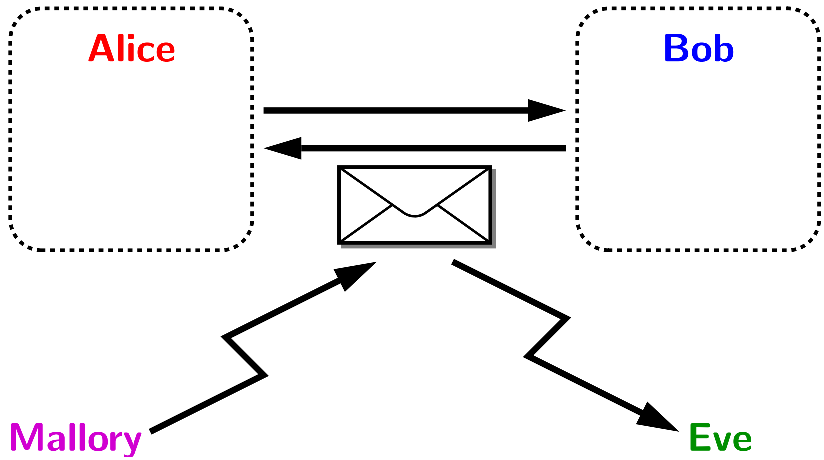
- Alice and Bob want to communicate using a public channel (e.g., Internet)

Cryptography



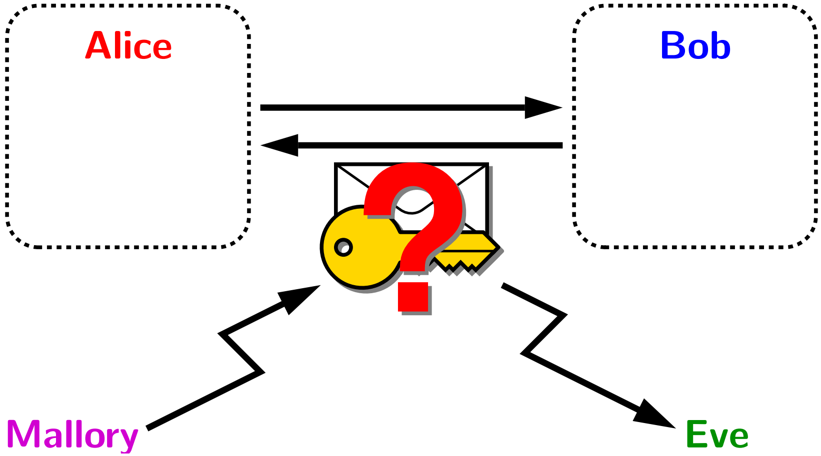
- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)

Cryptography



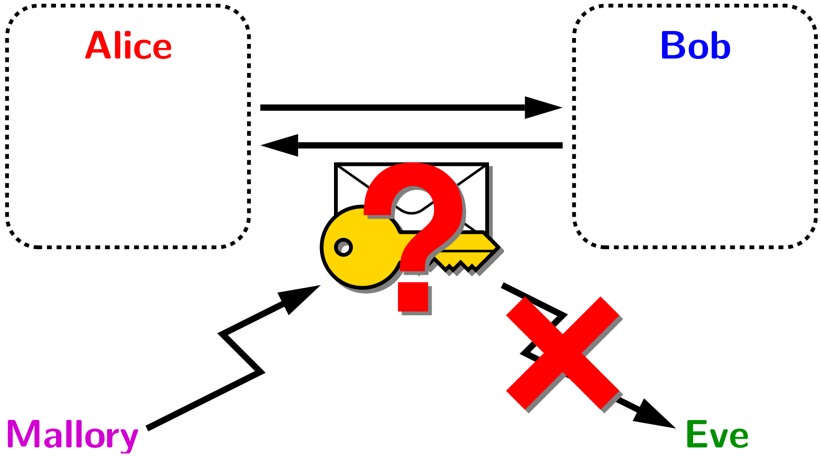
- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)
  - ... and Mallory is interfering (active attack: tampering, forgery, replay, etc.)

Cryptography



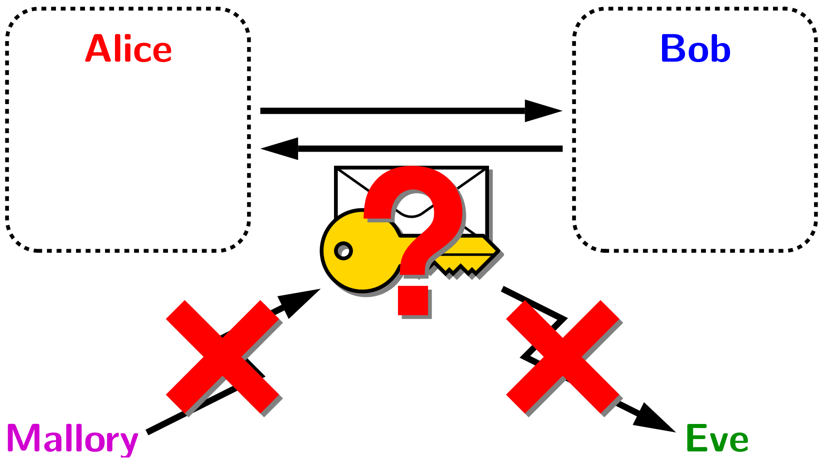
- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)
  - ... and Mallory is interfering (active attack: tampering, forgery, replay, etc.)
- Cryptography: how to prevent such attacks, and ensure

Cryptography



- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - … but Eve is listening (passive attack: eavesdropping)
  - … and Mallory is interfering (active attack: tampering, forgery, replay, etc.)
- Cryptography: how to prevent such attacks, and ensure
  - confidentiality (*Who can read the message?*) → encryption

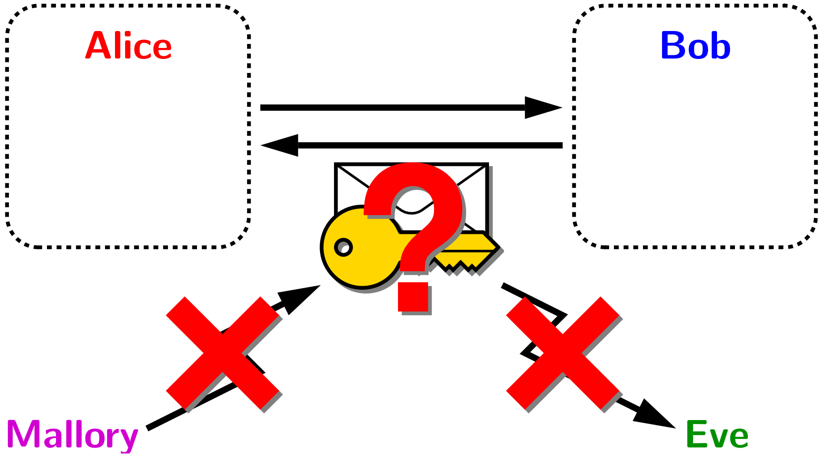
Cryptography



- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)
  - ... and Mallory is interfering (active attack: tampering, forgery, replay, etc.)

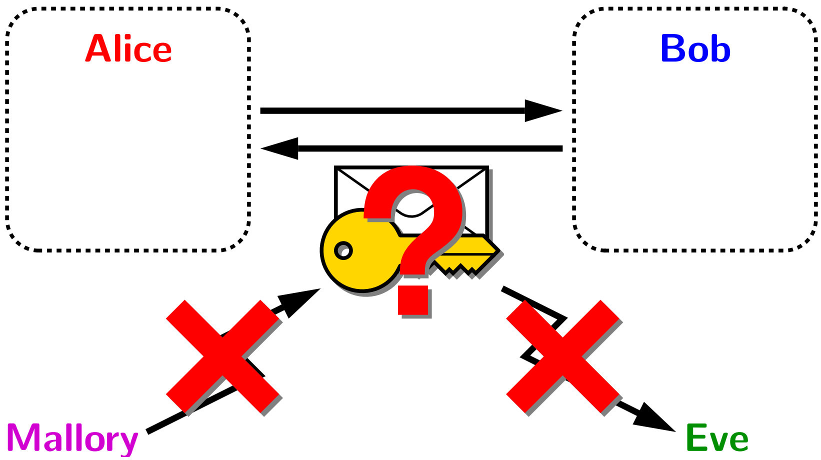
- Cryptography: how to prevent such attacks, and ensure
  - confidentiality (*Who can read the message?*) → encryption
  - integrity (*Was the message modified?*) → cryptographic hash functions

Cryptography



- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)
  - ... and Mallory is interfering (active attack: tampering, forgery, replay, etc.)
- Cryptography: how to prevent such attacks, and ensure
  - confidentiality (*Who can read the message?*) → encryption
  - integrity (*Was the message modified?*) → cryptographic hash functions
  - authenticity (*Who sent the message?*) → message auth. code (MAC), signature

Cryptography



- Alice and Bob want to communicate using a public channel (e.g., Internet)
  - ... but Eve is listening (passive attack: eavesdropping)
  - ... and Mallory is interfering (active attack: tampering, forgery, replay, etc.)
- Cryptography: how to prevent such attacks, and ensure
  - confidentiality (*Who can read the message?*) → encryption
  - integrity (*Was the message modified?*) → cryptographic hash functions
  - authenticity (*Who sent the message?*) → message auth. code (MAC), signature
  - ... and many others: non-repudiation, zero-knowledge proof, secret sharing, etc.

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors
- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - cryptographic primitives (AES, RSA, ECDH, etc.)
  - arithmetic and logic operations (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors
- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements
- All top layers (esp. the blue and green ones) might lead to critical vulnerabilities if poorly implemented!
  $\Rightarrow$ a cryptosystem is no more secure than its weakest link

Cryptographic layers

- A complete cryptosystem implementation relies on many layers:
  - protocol (OpenPGP, TLS, SSH, etc.)
  - cryptographic mechanisms (encryption, hashing, signature, etc.)
  - **cryptographic primitives** (AES, RSA, ECDH, etc.)
  - **arithmetic and logic operations** (CPU / ASIP instruction set)
  - logic circuits (registers, multiplexers, adders, etc.)
  - logic gates (NOT, NAND, etc.) and wires
  - transistors
- When designing a cryptoprocessor, the hardware/software partitioning can be tailored to the application's requirements
- All top layers (esp. the blue and green ones) might lead to critical vulnerabilities if poorly implemented!
  $\Rightarrow$ a cryptosystem is no more secure than its weakest link
- In this lecture, we will mostly focus on the **green layers**

Which target platforms?

- Cryptography should be available everywhere:

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    → 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    → low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    → tiny 8-bit microcontroller (such as Atmel AVRs)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    $\rightarrow$ tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    $\rightarrow$ custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    $\rightarrow$ tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    $\rightarrow$ custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations

- Other possible target platforms, mostly for cryptanalytic computations:
  - clusters of CPUs
  - GPUs (graphics processors)
  - FPGAs (reconfigurable circuits)

Which target platforms?

- Cryptography should be available everywhere:
  - on desktop PCs and laptops
    $\rightarrow$ 64-bit Intel or AMD CPUs with SIMD instructions (SSE / AVX)
  - on smartphones
    $\rightarrow$ low-power 32- or 64-bit ARM CPUs, maybe with SIMD (NEON)
  - on wireless sensors
    $\rightarrow$ tiny 8-bit microcontroller (such as Atmel AVRs)
  - on smart cards and RFID chips
    $\rightarrow$ custom cryptoprocessor (ASIC or ASIP) with dedicated hardware for cryptographic operations
- Other possible target platforms, mostly for cryptanalytic computations:
  - clusters of CPUs
  - GPUs (graphics processors)
  - FPGAs (reconfigurable circuits)

  $\Rightarrow$ In such cases, implementation security is usually less critical

Efficient and secure implementation?

- Many possible meanings for efficiency:

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\to$ low latency or high throughput?
  - small? $\to$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
  - low power?... or low energy?

  ⇒ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?
  - power or electromagnetic analysis?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\rightarrow$ low latency or high throughput?
  - small? $\rightarrow$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?
  - power or electromagnetic analysis?
  - fault attacks? [See A. Tisserand's talk]

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? $\to$ low latency or high throughput?
  - small? $\to$ low memory / code / silicon usage?
  - low power?... or low energy?

  $\Rightarrow$ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?
  - power or electromagnetic analysis?
  - fault attacks? [See A. Tisserand's talk]
  - cache attacks?

Efficient and secure implementation?

- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
  - low power?... or low energy?

  ⇒ Identify constraints according to application and target platform

- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?
  - power or electromagnetic analysis?
  - fault attacks? [See A. Tisserand's talk]
  - cache attacks?
  - branch-prediction attacks?

Efficient and secure implementation?

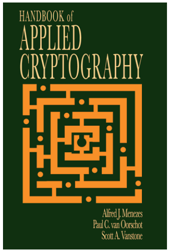
- Many possible meanings for efficiency:
  - fast? → low latency or high throughput?
  - small? → low memory / code / silicon usage?
  - low power?... or low energy?

  ⇒ Identify constraints according to application and target platform

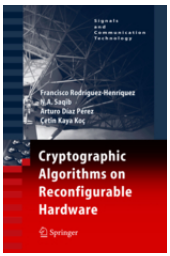
- Secure against which attacks?
  - protocol attacks? (POODLE, FREAK, LogJam, etc.)
  - cryptanalysis? (weak cipher, small keys, etc.)
  - timing attacks?
  - power or electromagnetic analysis?
  - fault attacks? [See A. Tisserand's talk]
  - cache attacks?
  - branch-prediction attacks?
  - etc.

  ⇒ Possible attack scenarios depend on the application

Some references



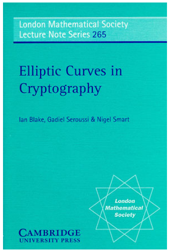
Alfred Menezes, Paul van Oorschot, and Scott Vanstone,
Handbook of Applied Cryptography.
Chapman & Hall / CRC, 1996.
`http://www.cacr.math.uwaterloo.ca/hac/`



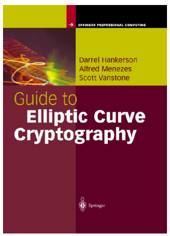
Francisco Rodríguez-Henríquez, Arturo Díaz Pérez, Nazar Abbas Saqib, and Çetin Kaya Koç,
Cryptographic Algorithms on Reconfigurable Hardware.
Springer, 2006.

Proceedings of the CHES workshop and of other crypto conferences.

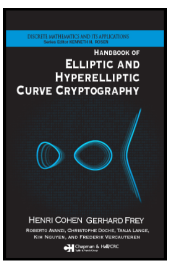
Some references



Elliptic Curves in Cryptography,
Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart.
London Mathematical Society 265,
Cambridge University Press, 1999.



Guide to Elliptic Curve Cryptography,
Darrel Hankerson, Alfred Menezes, and Scott Vanstone.
Springer, 2004.

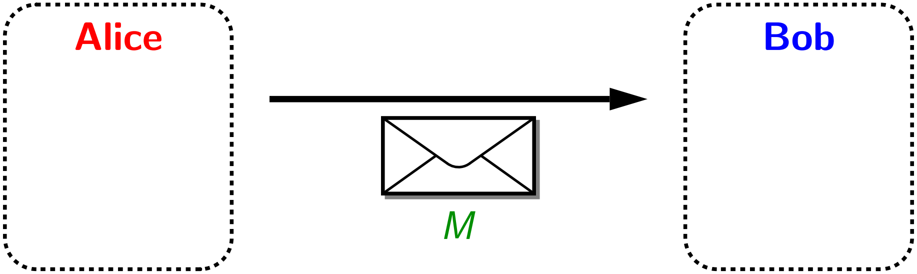


Handbook of Elliptic and Hyperelliptic Curve Cryptography,
Henri Cohen and Gerhard Frey (editors).
Chapman & Hall / CRC, 2005.

Outline

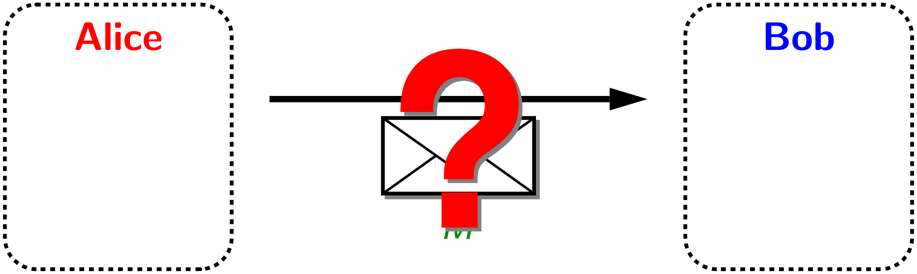
- Some encryption mechanisms
- Elliptic curve cryptography
- Scalar multiplication
- Elliptic curve arithmetic
- Finite field arithmetic

Symmetric encryption



- Alice wants to send a confidential message M to Bob

Symmetric encryption

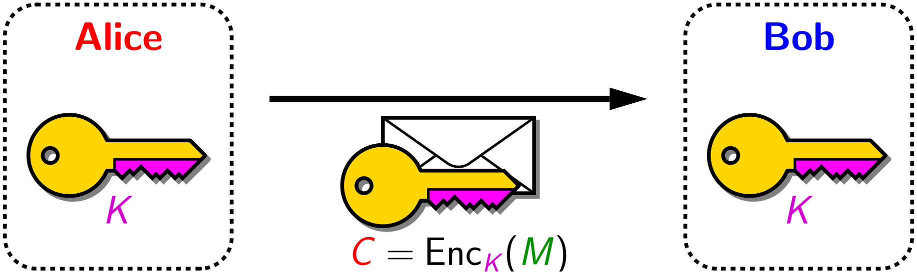


- Alice wants to send a confidential message M to Bob

Symmetric encryption

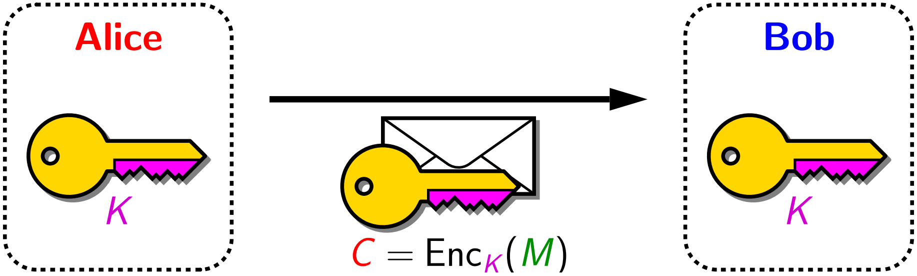
- Alice wants to send a confidential message M to Bob
  - they decide upon a shared secret key K (might be tricky!)

Symmetric encryption



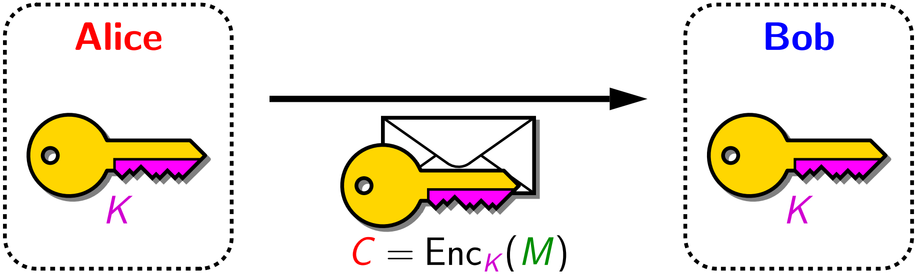
- Alice wants to send a confidential message M to Bob
  - they decide upon a shared secret key K (might be tricky!)
  - encrypt message using shared key: $C = \mathsf{Enc}_K(M)$

Symmetric encryption



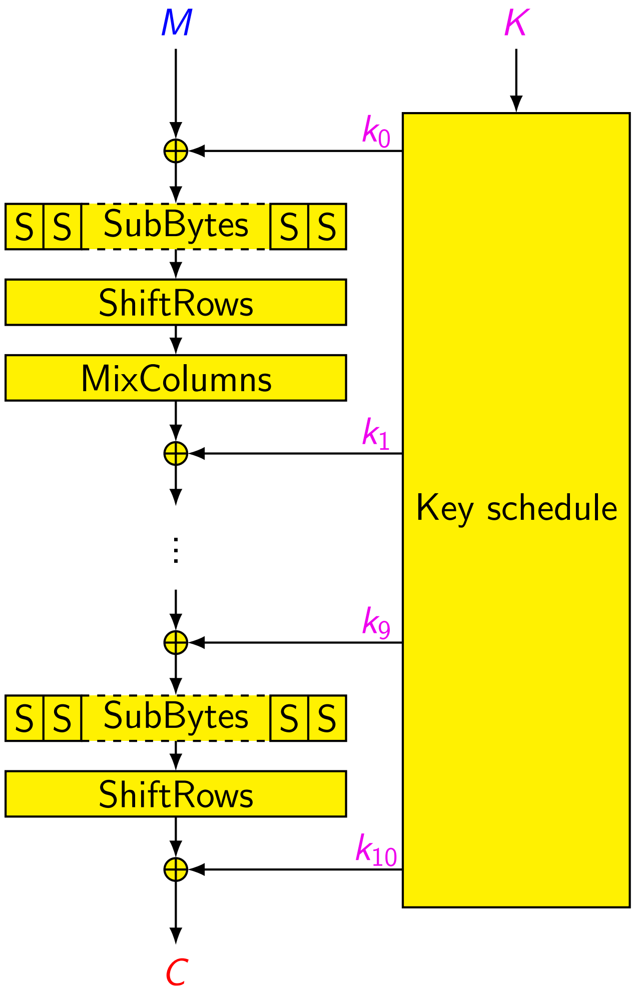
- Alice wants to send a confidential message M to Bob
  - they decide upon a shared secret key K (might be tricky!)
  - encrypt message using shared key: $C = \mathrm{Enc}_K(M)$
  - decrypt ciphertext using shared key: $M = \mathrm{Dec}_K(C)$

Symmetric encryption



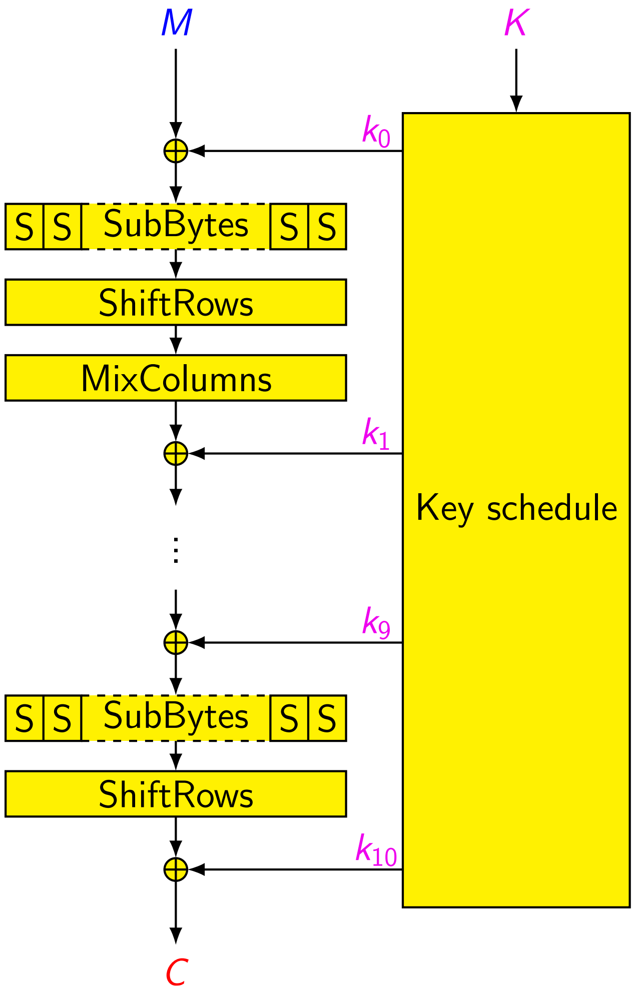
- Alice wants to send a confidential message M to Bob
  - they decide upon a shared secret key K (might be tricky!)
  - encrypt message using shared key: $C = \mathrm{Enc}_K(M)$
  - decrypt ciphertext using shared key: $M = \mathrm{Dec}_K(C)$
- Block cipher:
  - split message M into n-bit blocks (e.g., $n = 128$ bits)
  - encryption/decryption primitive : iterated keyed permutation $\{0,1\}^n \rightarrow \{0,1\}^n$
  - requires a mode of operation to combine the blocks

AES [Daemen & Rijmen, 2001]



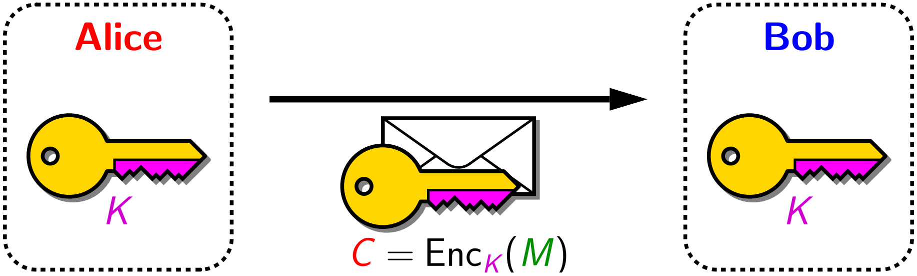
- Advanced Encryption Standard
- Key sizes: 128, 192 or 256 bits
- Block size: 128 bits
- Substitution–permutation network
  - SubBytes: nonlinear subst. on bytes
  - ShiftRows & MixColumns: mainly wires, plus a few XORs
- 10, 12, or 14 rounds (depending on key size)

AES [Daemen & Rijmen, 2001]



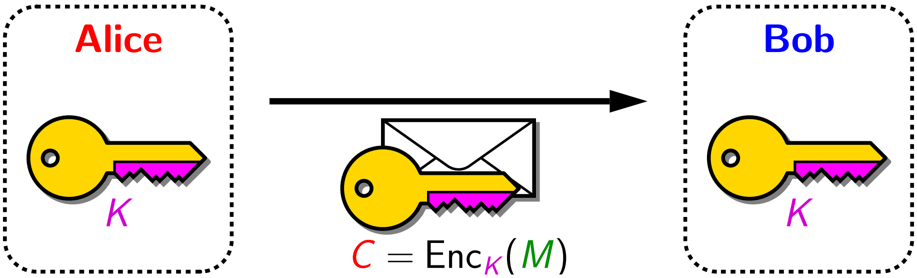
- Advanced Encryption Standard
- Key sizes: 128, 192 or 256 bits
- Block size: 128 bits
- Substitution–permutation network
  - SubBytes: nonlinear subst. on bytes
  - ShiftRows & MixColumns: mainly wires, plus a few XORs
- 10, 12, or 14 rounds (depending on key size)
- Low-area version (1 S-box): 20 cycles / round, 2.5 to 5 kGE
- Parallel version (20 S-boxes): 1 cycle / round, 20 to 35 kGE
- Fully unrolled version (200 S-boxes): 1 cycle / block, at least 200 kGE

Symmetric encryption



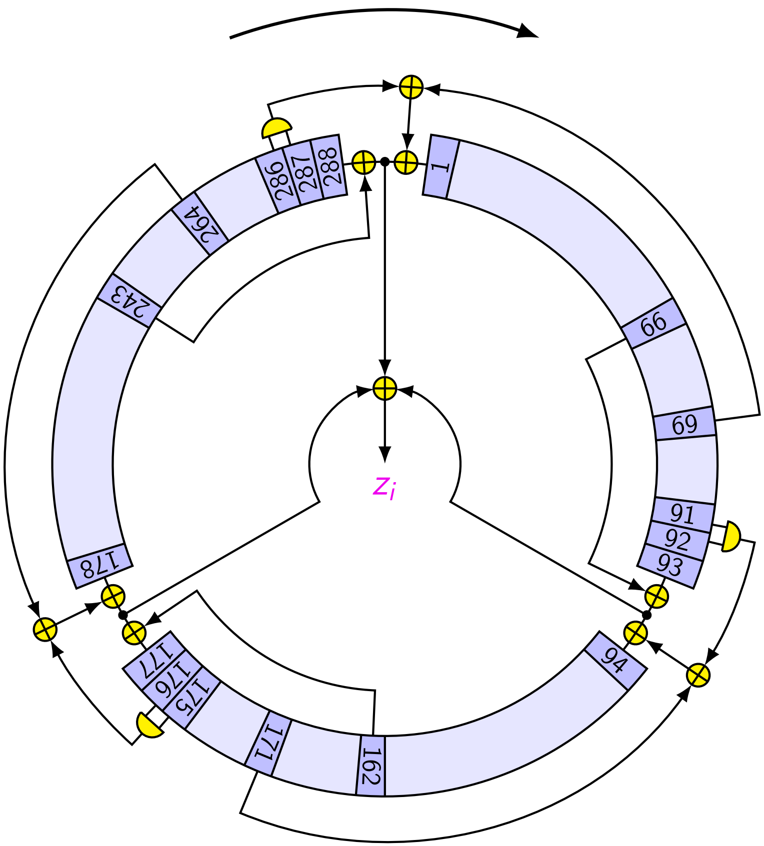
- Alice wants to send a confidential message M to Bob
  - decide upon a shared secret key K
  - encrypt message using shared key: $C = \mathrm{Enc}_K(M)$
  - decrypt ciphertext using shared key: $M = \mathrm{Dec}_K(C)$
- Block cipher:
  - split message M into n-bit blocks (e.g., $n = 128$ bits)
  - encryption/decryption primitive : keyed permutation $\{0,1\}^n \to \{0,1\}^n$
  - requires a mode of operation to combine the blocks

Symmetric encryption



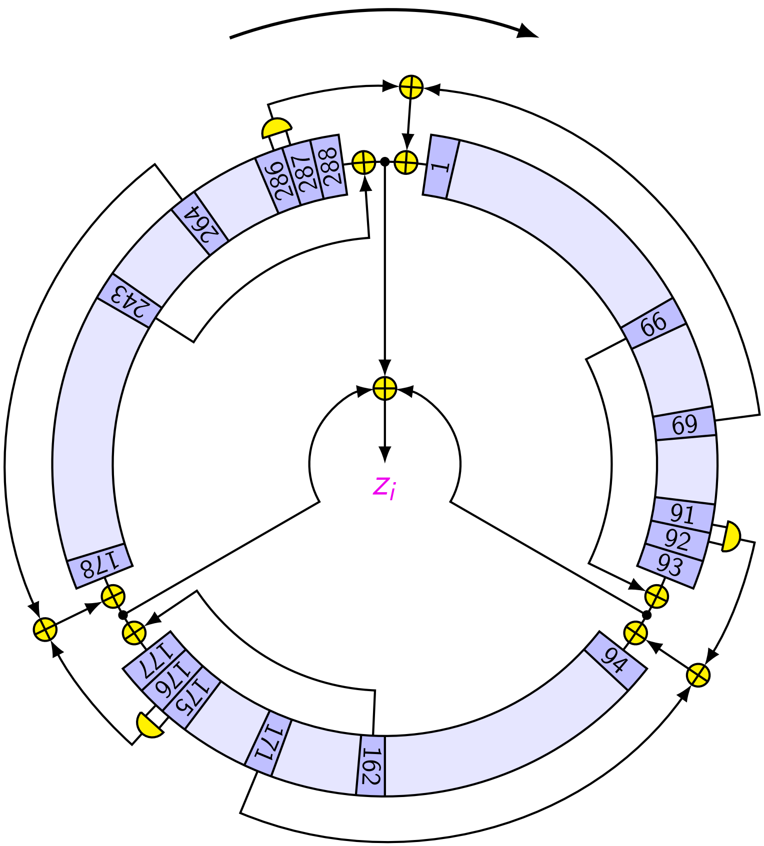
- Alice wants to send a confidential message M to Bob
  - decide upon a shared secret key K
  - encrypt message using shared key: $C = \mathrm{Enc}_K(M)$
  - decrypt ciphertext using shared key: $M = \mathrm{Dec}_K(C)$
- Block cipher:
  - split message M into n-bit blocks (e.g., $n = 128$ bits)
  - encryption/decryption primitive : keyed permutation $\{0,1\}^n \to \{0,1\}^n$
  - requires a mode of operation to combine the blocks
- Stream cipher:
  - generate a pseudorandom keystream Z using a PRNG initialized by the key K and a random initialization vector (IV)
  - use Z to mask the message: $C = M \oplus Z$ and $M = C \oplus Z$ ($\oplus$ is XOR)

Trivium [De Cannière & Preneel, 2005]



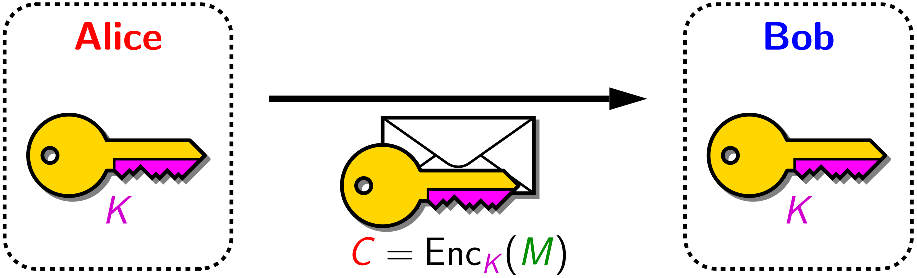
- Part of the eSTREAM portfolio (low-area hardware ciphers)
- Key size: 80 bits
- IV size: 80 bits
- 288-bit circular shift register, plus a few XOR and AND gates

Trivium [De Cannière & Preneel, 2005]

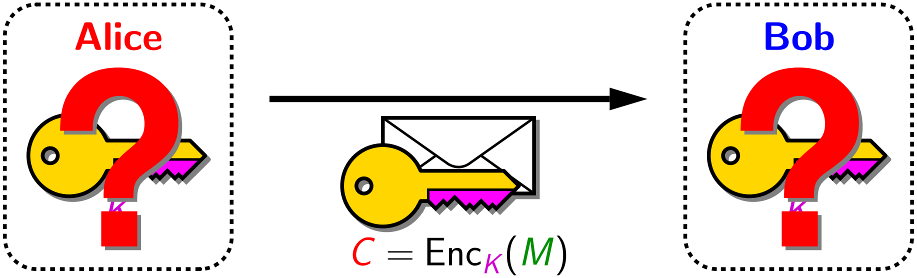


- Part of the eSTREAM portfolio (low-area hardware ciphers)
- Key size: 80 bits
- IV size: 80 bits
- 288-bit circular shift register, plus a few XOR and AND gates
- Serial version:
  - 1 keystream bit / clock cycle
  - 2.6 kGE
- Parallel version:
  - up to 64 bits / clock cycle
  - 4.9 kGE

Public-key encryption



Public-key encryption



- Agreeing on a shared secret key K over a public channel is difficult

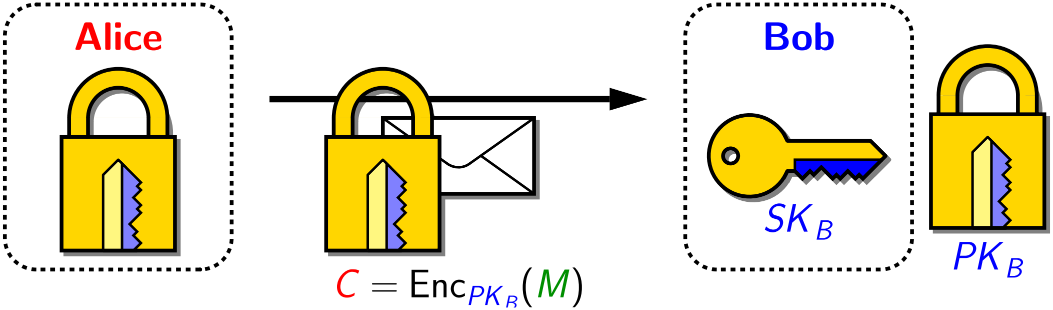
Public-key encryption

- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)

Public-key encryption

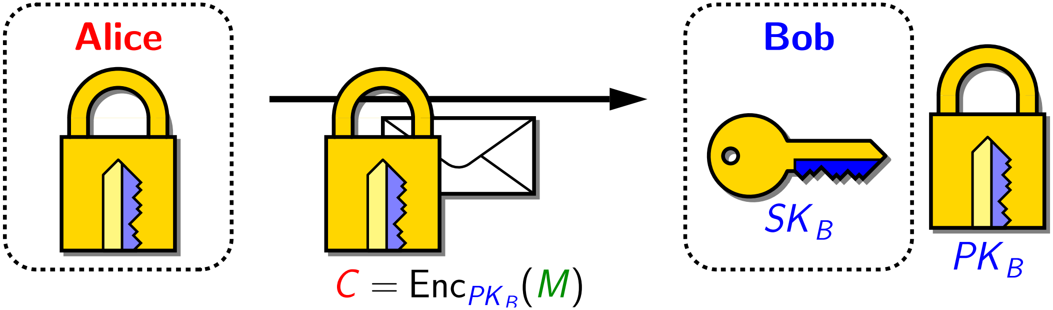
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B

Public-key encryption



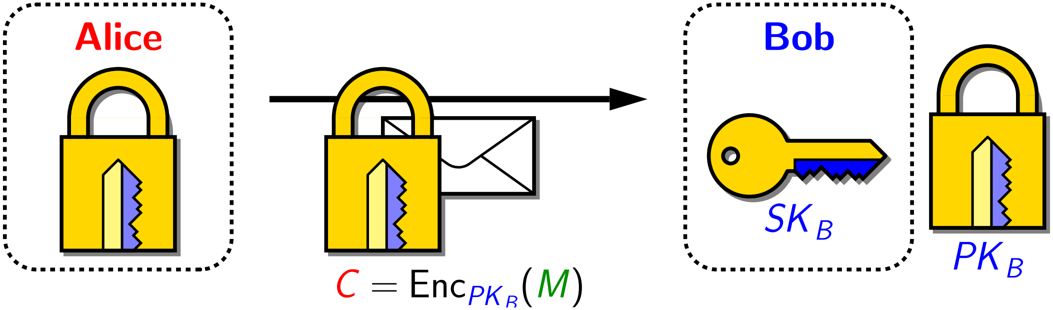
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$

Public-key encryption



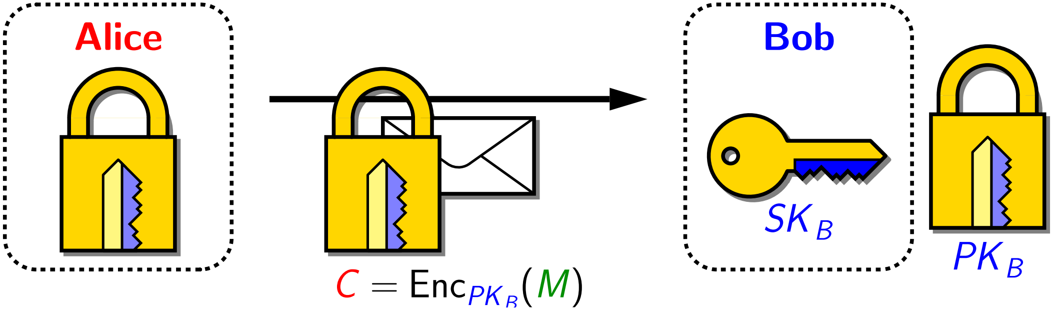
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$

Public-key encryption



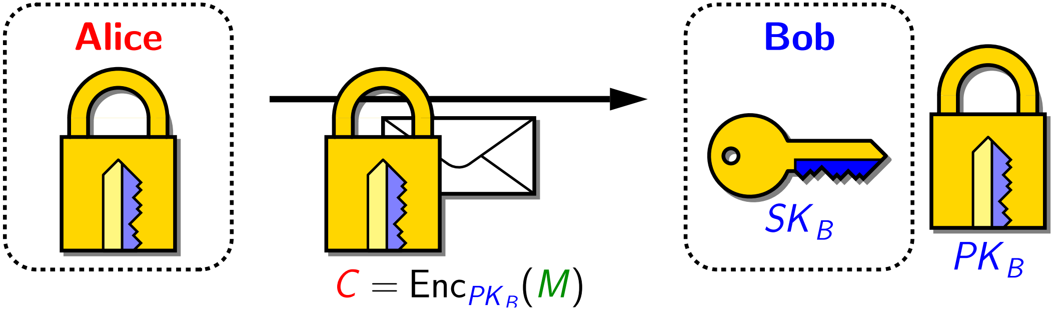
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \text{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \text{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult

Public-key encryption



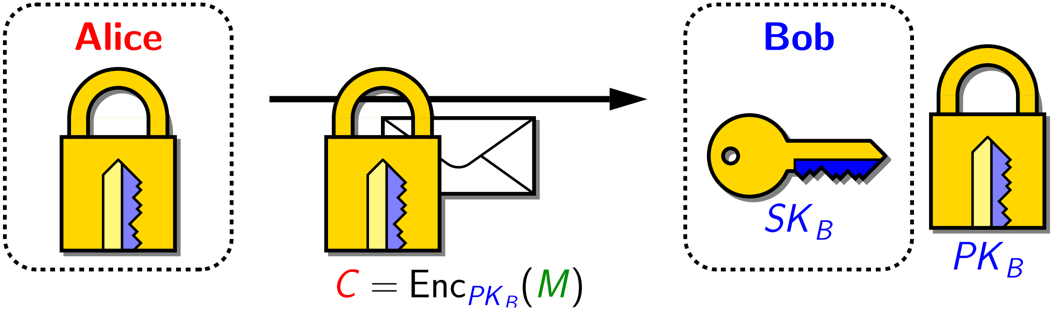
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems

Public-key encryption



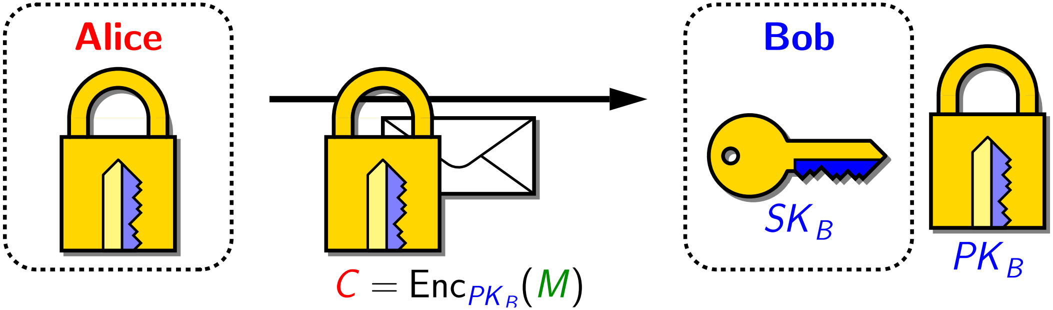
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems
  - integer factorization (RSA)

Public-key encryption



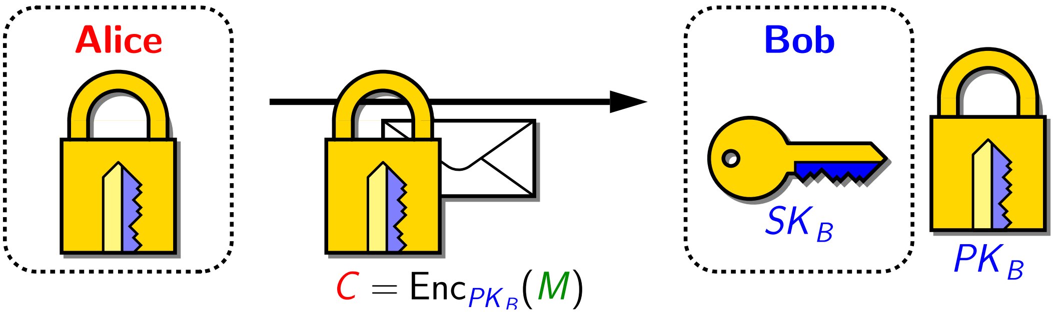
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems
  - integer factorization (RSA)
  - discrete logarithm problem in finite fields (ElGamal, DSA, etc.)

Public-key encryption



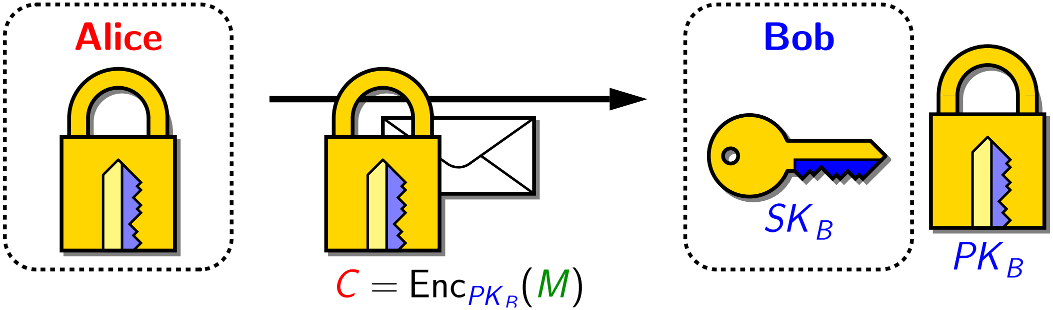
- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems
  - integer factorization (RSA)
  - discrete logarithm problem in finite fields (ElGamal, DSA, etc.)
  - discrete logarithm problem in elliptic curves (elliptic curve cryptography)

Public-key encryption



- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems
  - integer factorization (RSA)
  - discrete logarithm problem in finite fields (ElGamal, DSA, etc.)
  - discrete logarithm problem in elliptic curves (elliptic curve cryptography)
    $\rightarrow$ harder problem, and thus requires smaller keys (256 vs. 3072 bits)

Public-key encryption



- Agreeing on a shared secret key K over a public channel is difficult
- Use public-key cryptography:
  - Bob generates a public/secret key-pair (PK_B, SK_B)
  - Alice retrieves Bob's public key PK_B
  - encryption only uses the public key: $C = \mathrm{Enc}_{PK_B}(M)$
  - but decryption requires the secret key: $M = \mathrm{Dec}_{SK_B}(C)$
- Security: computing SK_B from PK_B should be difficult
  $\Rightarrow$ rely on (supposedly) "hard" number-theoretic problems
  - integer factorization (RSA)
  - discrete logarithm problem in finite fields (ElGamal, DSA, etc.)
  - **discrete logarithm problem in elliptic curves** (elliptic curve cryptography)
    $\rightarrow$ harder problem, and thus requires smaller keys (256 vs. 3072 bits)

Outline

- Some encryption mechanisms
- Elliptic curve cryptography
- Scalar multiplication
- Elliptic curve arithmetic
- Finite field arithmetic

A primer on elliptic curves

- Let us consider a field K (e.g., $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, etc.)

A primer on elliptic curves

- Let us consider a field K (e.g., $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, etc.)
- An elliptic curve E defined over K is given by an equation of the form

$$E : y^2 = x^3 + Ax + B, \qquad \text{with parameters } A, B \in K$$

A primer on elliptic curves

- Let us consider a field K (e.g., $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, etc.)
- An elliptic curve E defined over K is given by an equation of the form

$$E : y^2 = x^3 + Ax + B, \qquad \text{with parameters } A, B \in K$$

- The set of K-rational points of E is defined as

$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfy } E\}$$

A primer on elliptic curves

- Let us consider a field K (e.g., $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, etc.)
- An elliptic curve E defined over K is given by an equation of the form

$$E : y^2 = x^3 + Ax + B, \qquad \text{with parameters } A, B \in K$$

- The set of K-rational points of E is defined as

$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfy } E\} \cup \{\mathcal{O}\}$$

  $\mathcal{O}$ is called the "point at infinity"

A primer on elliptic curves

- Let us consider a field K (e.g., $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{F}_p$, etc.)
- An elliptic curve E defined over K is given by an equation of the form

$$E : y^2 = x^3 + Ax + B, \qquad \text{with parameters } A, B \in K$$

- The set of K-rational points of E is defined as

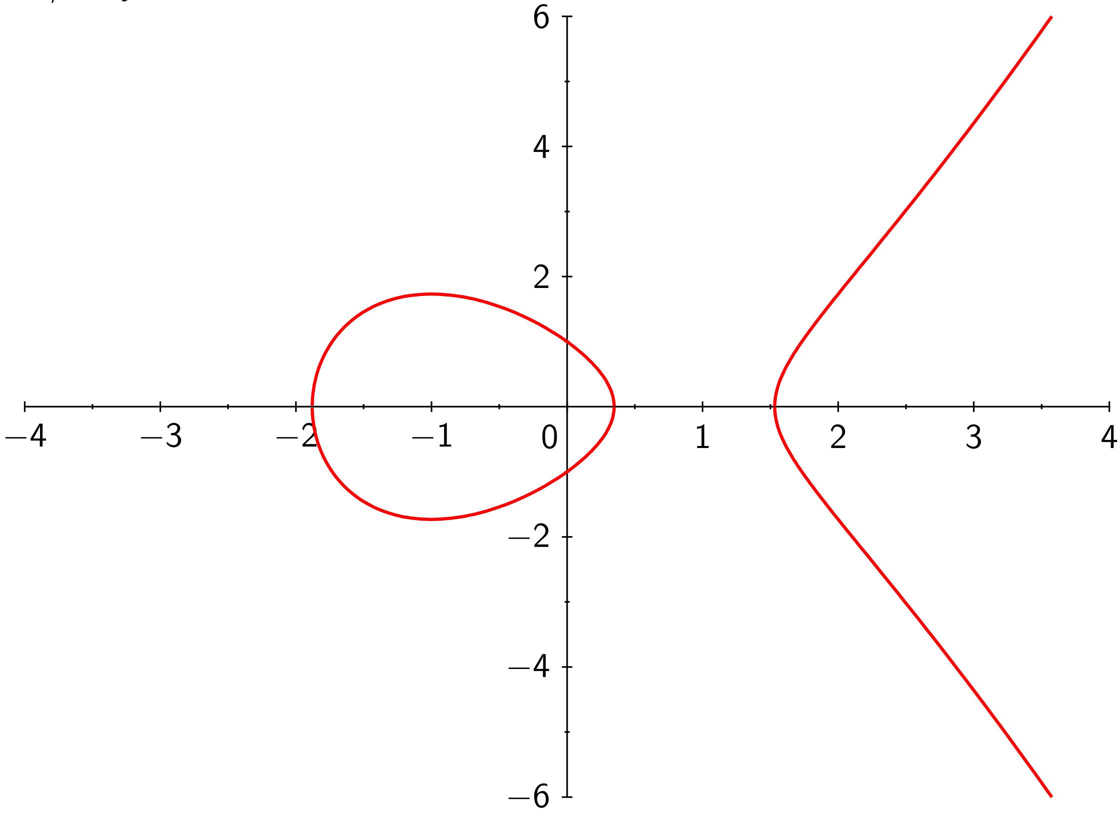
$$E(K) = \{(x, y) \in K \times K \mid (x, y) \text{ satisfy } E\} \cup \{\mathcal{O}\}$$

  $\mathcal{O}$ is called the "point at infinity"

- Additive group law: $E(K)$ is an abelian group
  - addition via the "chord and tangent" method
  - $\mathcal{O}$ is the neutral element

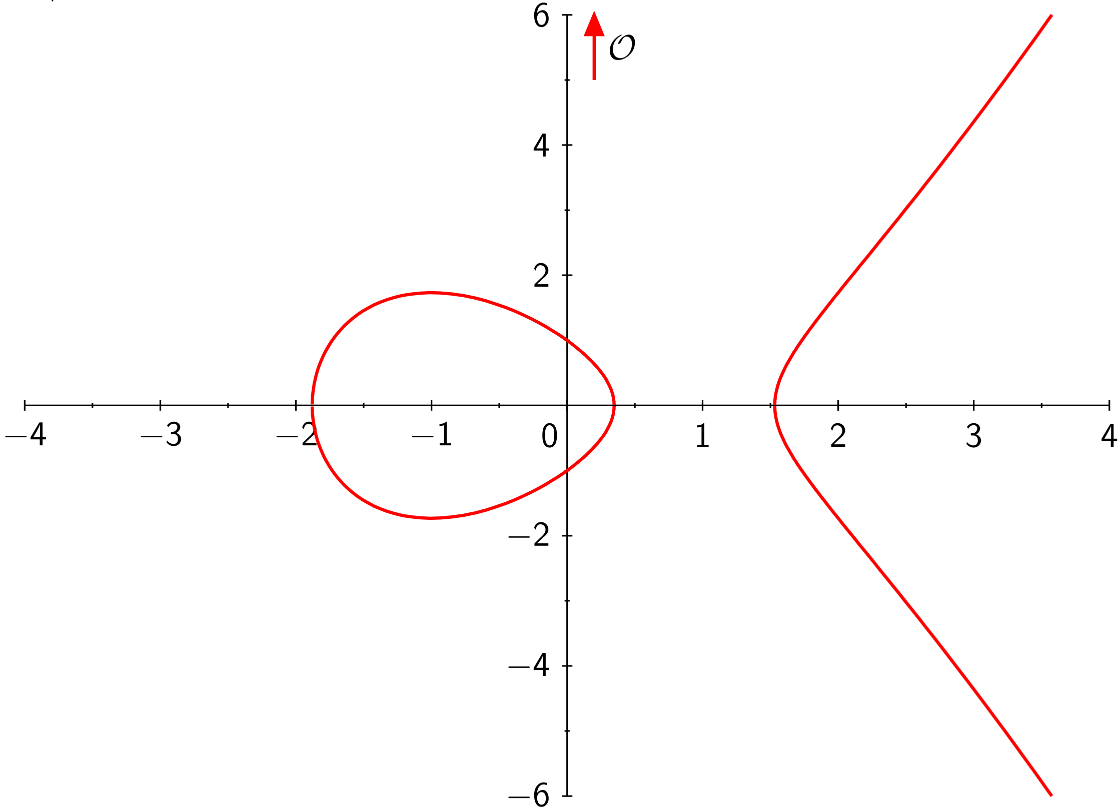
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



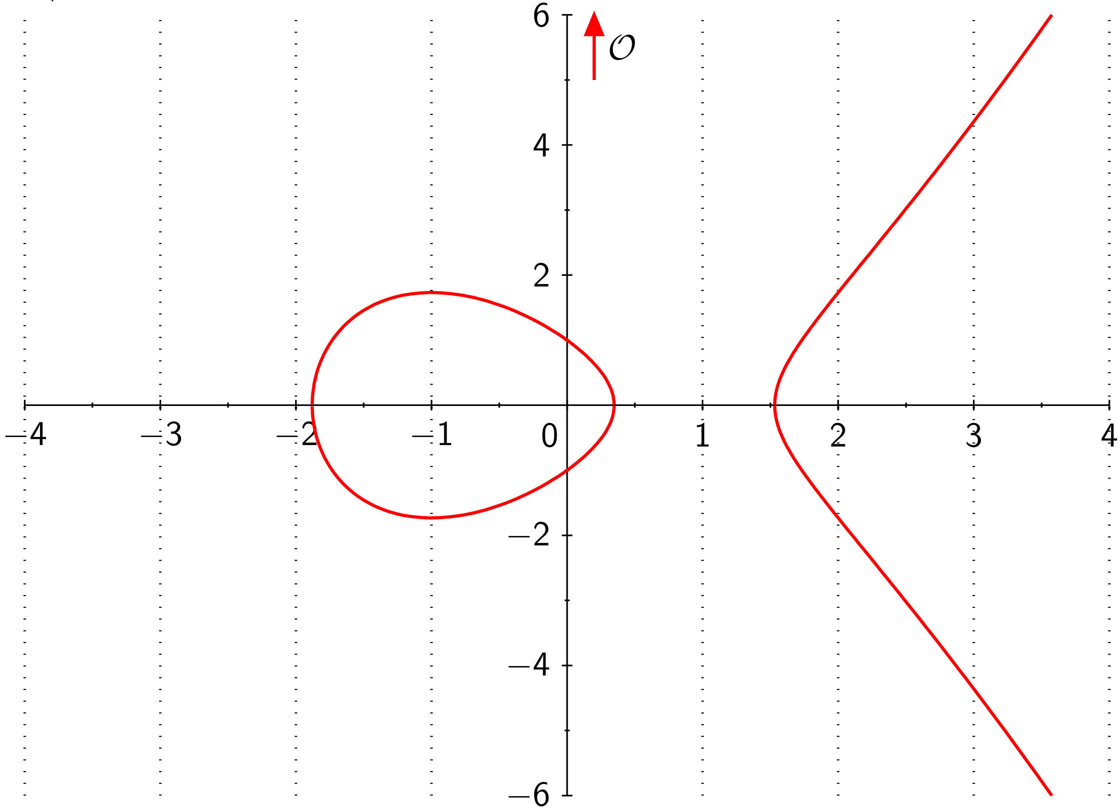
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



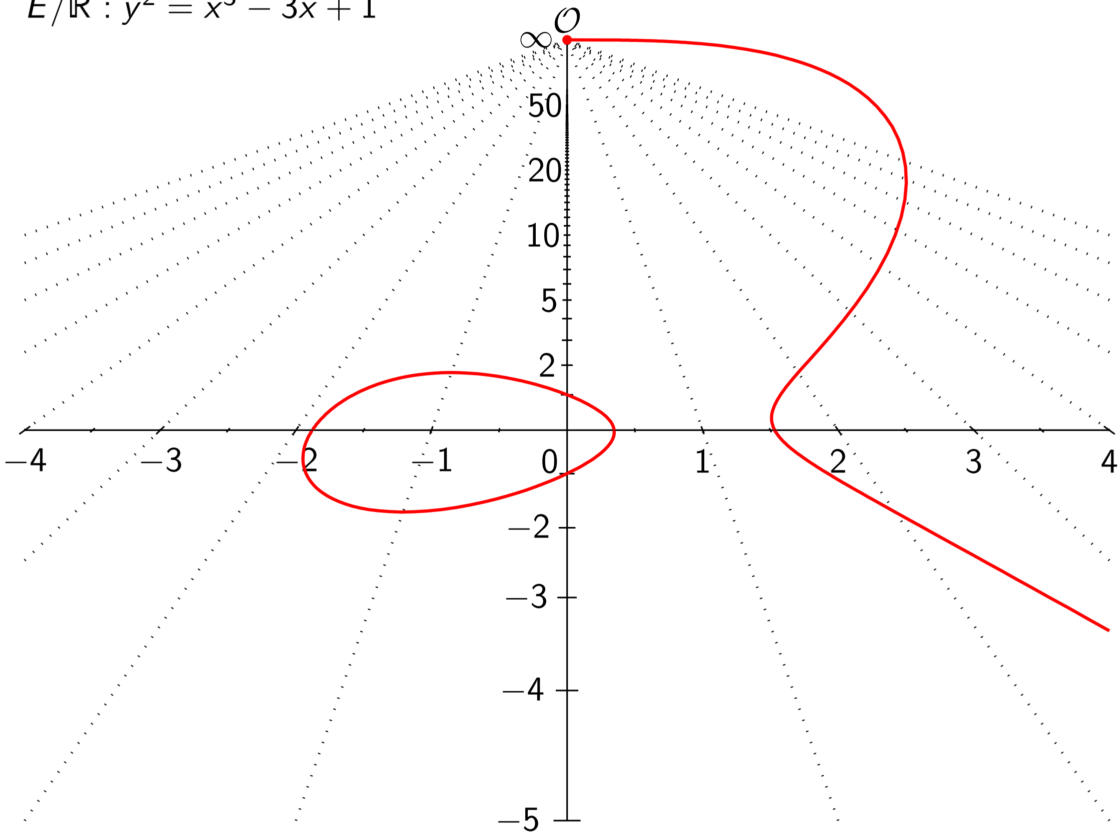
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



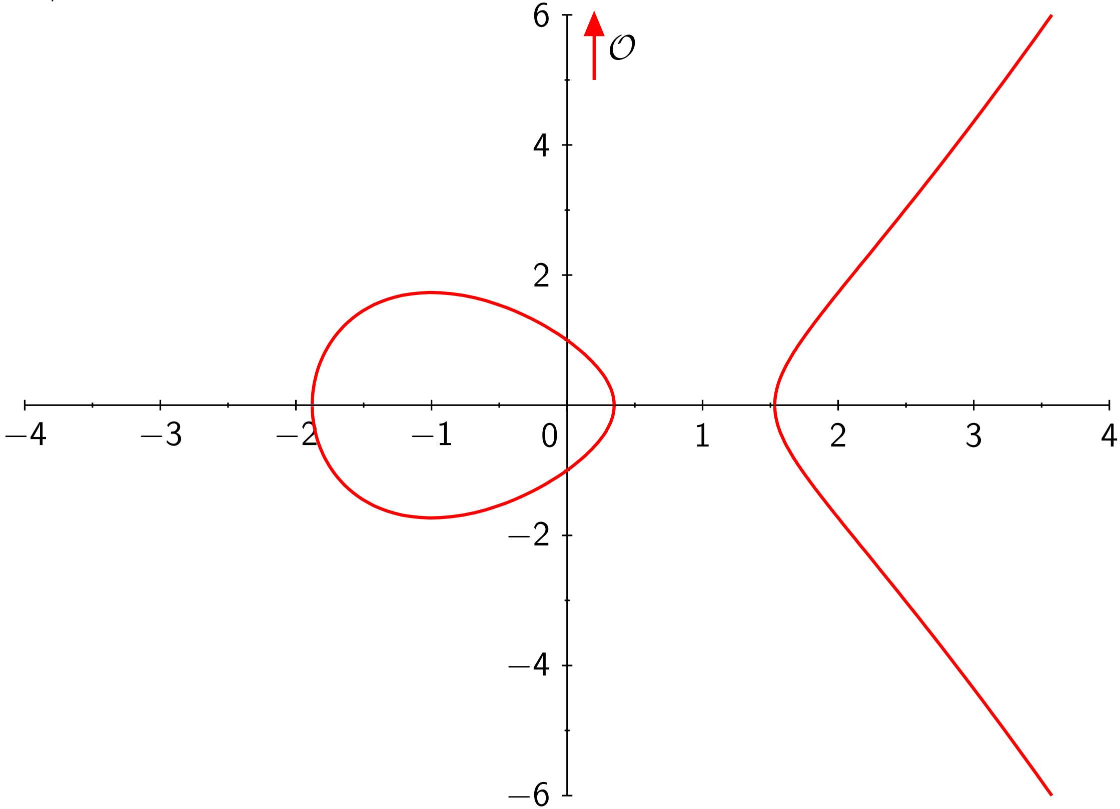
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



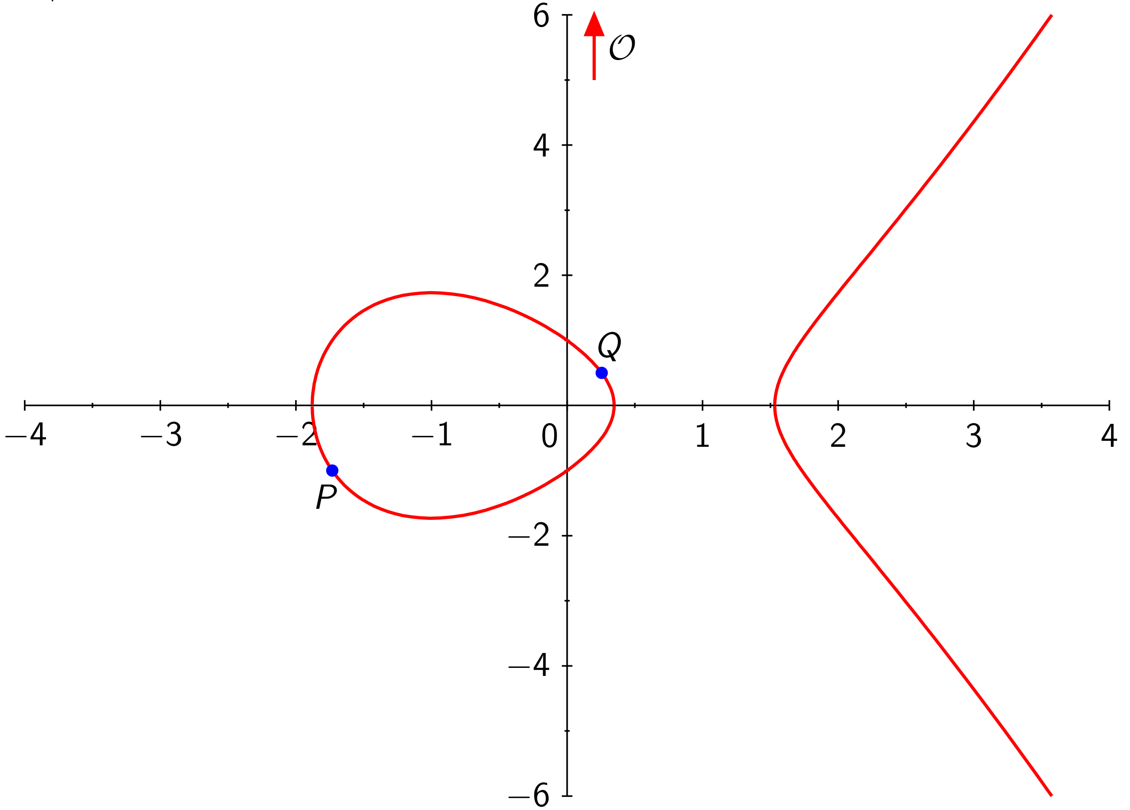
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



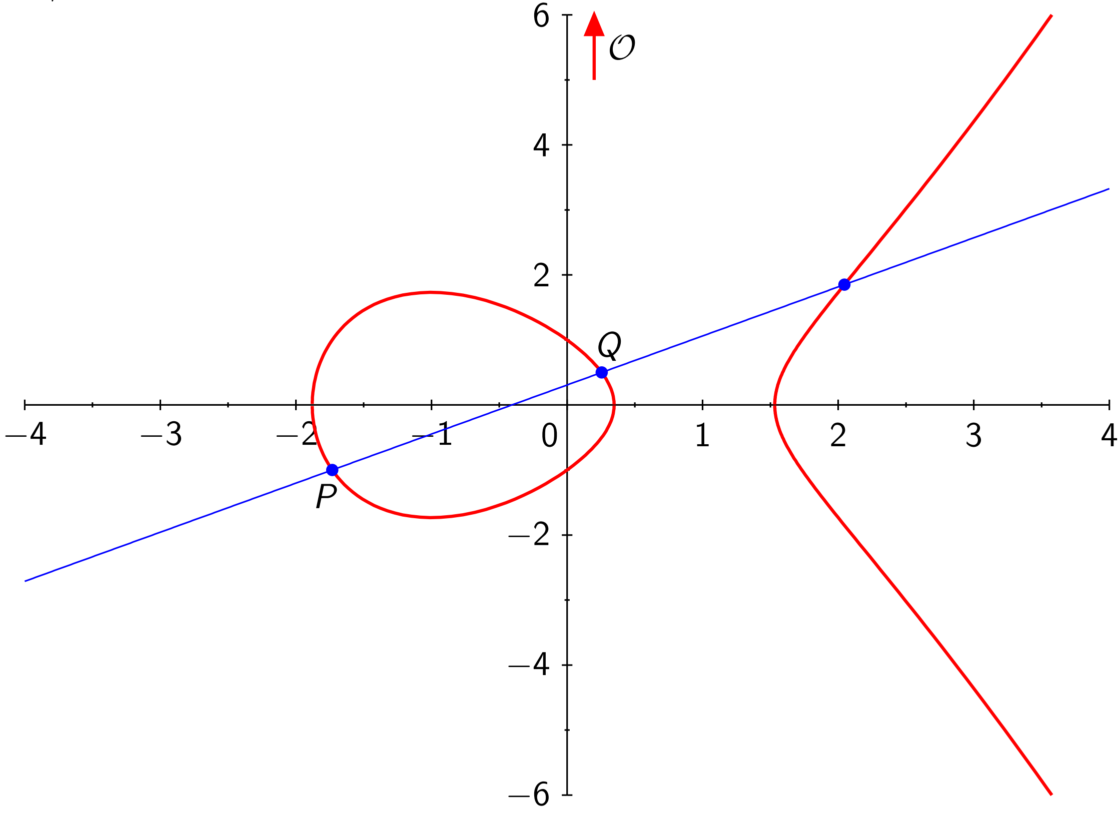
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



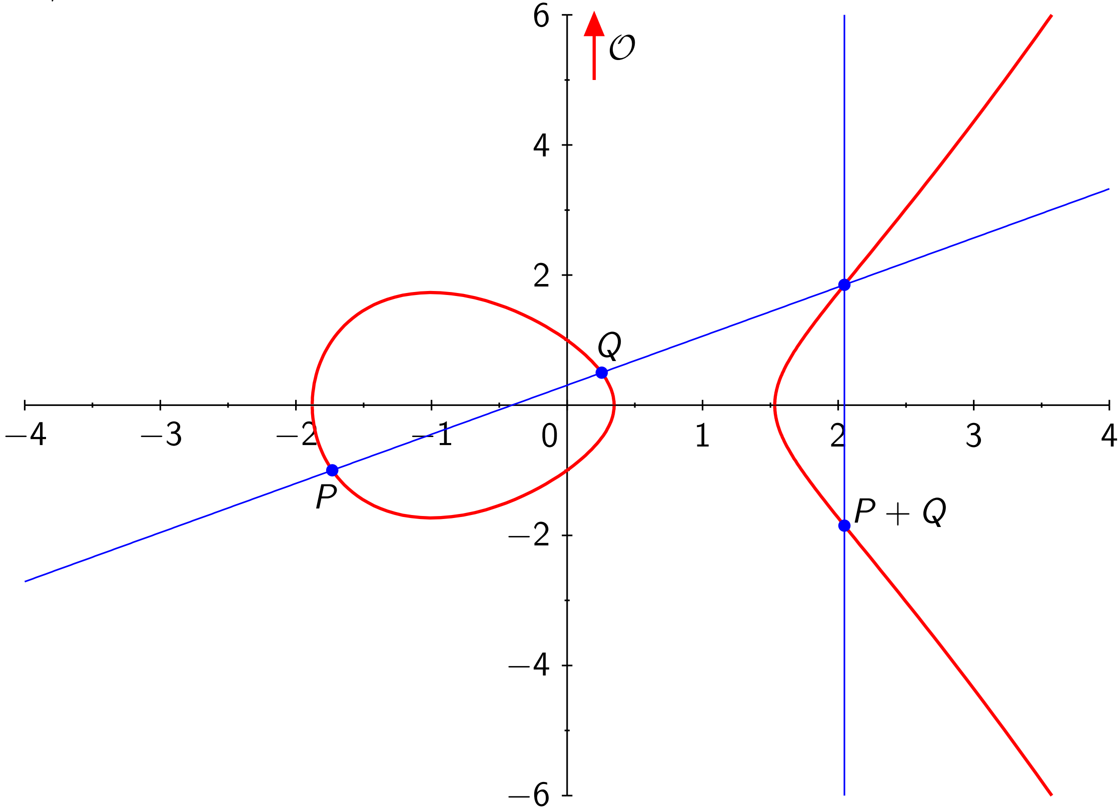
Elliptic curves and group law

$E/\mathbb{R} : y^2 = x^3 - 3x + 1$

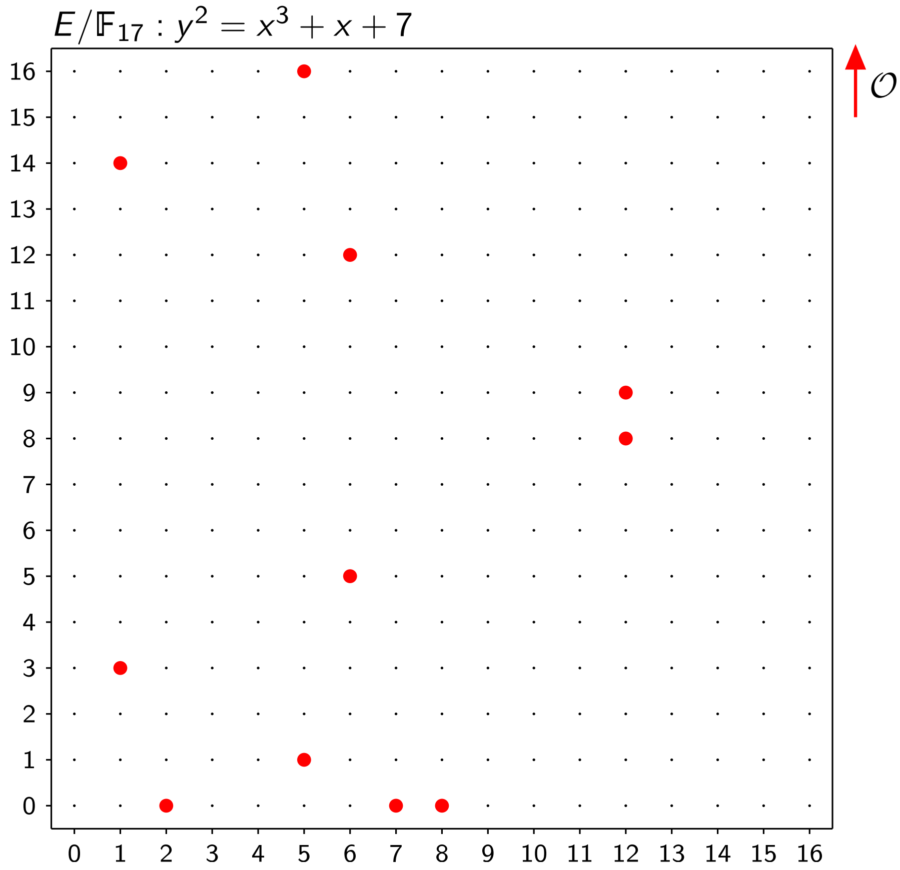


Elliptic curves and group law

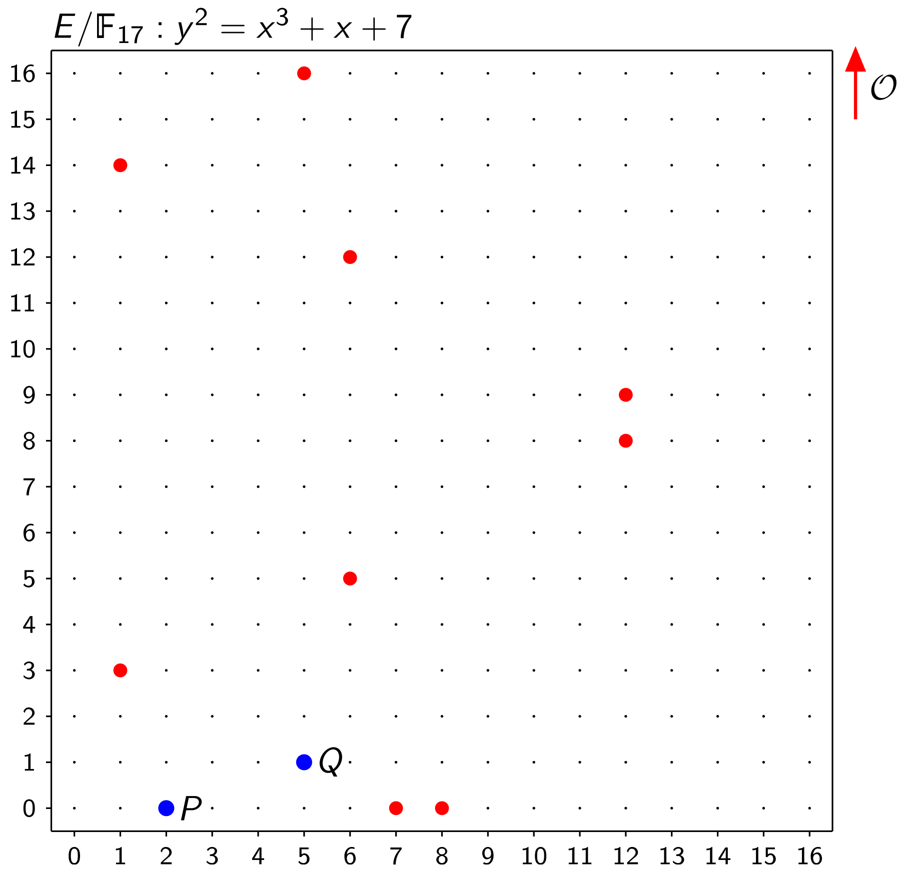
$E/\mathbb{R} : y^2 = x^3 - 3x + 1$



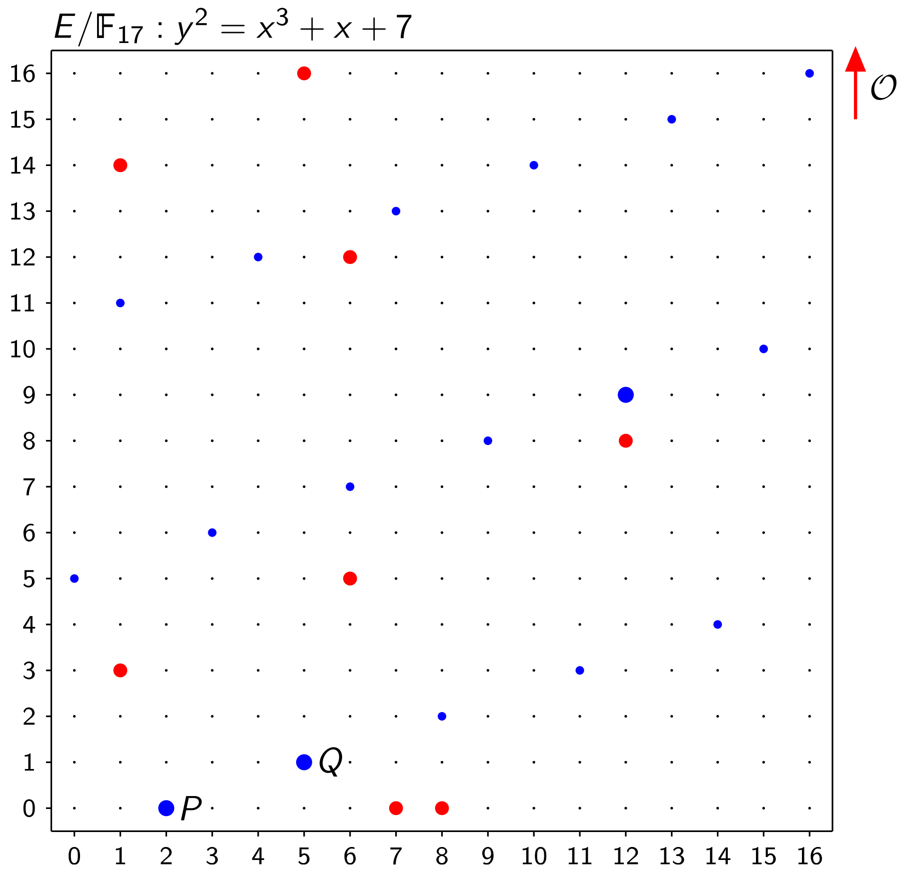
Elliptic curves and group law



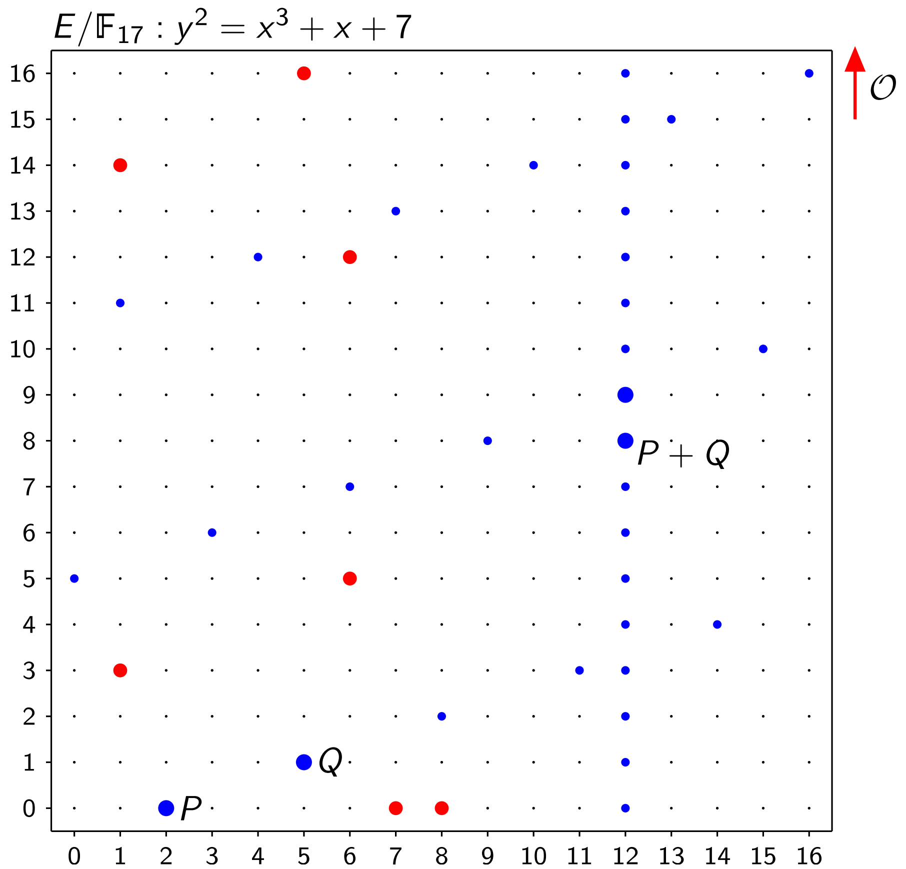
Elliptic curves and group law



Elliptic curves and group law



Elliptic curves and group law



Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.
  - since $E(\mathbb{F}_q)$ is finite, take the smallest $\ell > 0$ such that $\ell P = \mathcal{O}$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.
  - since $E(\mathbb{F}_q)$ is finite, take the smallest $\ell > 0$ such that $\ell P = \mathcal{O}$
  - define $\mathbb{G}$ as

$$\mathbb{G} = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.
  - since $E(\mathbb{F}_q)$ is finite, take the smallest $\ell > 0$ such that $\ell P = \mathcal{O}$
  - define $\mathbb{G}$ as

$$\mathbb{G} = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

  - $\mathbb{G}$ is a cyclic subgroup of $E(\mathbb{F}_q)$, of order ℓ, and P is a generator of $\mathbb{G}$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.
  - since $E(\mathbb{F}_q)$ is finite, take the smallest $\ell > 0$ such that $\ell P = \mathcal{O}$
  - define $\mathbb{G}$ as

$$\mathbb{G} = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$

  - $\mathbb{G}$ is a cyclic subgroup of $E(\mathbb{F}_q)$, of order ℓ, and P is a generator of $\mathbb{G}$

- The scalar multiplication in base P gives an isomorphism between $\mathbb{Z}/\ell\mathbb{Z}$ and $\mathbb{G}$:

$$\begin{array}{rccl} \exp_P : & \mathbb{Z}/\ell\mathbb{Z} & \longrightarrow & \mathbb{G} \\ & k & \longmapsto & kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \end{array}$$

Scalar multiplication and discrete logarithm

$$E/K : y^2 = x^3 + Ax + B$$

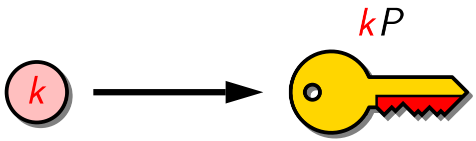
- If K is a finite field $\mathbb{F}_q$, then $E(K)$ is a finite abelian group
  - let $P \in E(\mathbb{F}_q)$, with $P \neq \mathcal{O}$
  - consider $2P = P + P$, then $3P = P + P + P$, etc.
  - since $E(\mathbb{F}_q)$ is finite, take the smallest $\ell > 0$ such that $\ell P = \mathcal{O}$
  - define $\mathbb{G}$ as
    $$\mathbb{G} = \{\mathcal{O}, P, 2P, 3P, \ldots, (\ell - 1)P\}$$
  - $\mathbb{G}$ is a cyclic subgroup of $E(\mathbb{F}_q)$, of order ℓ, and P is a generator of $\mathbb{G}$
- The scalar multiplication in base P gives an isomorphism between $\mathbb{Z}/\ell\mathbb{Z}$ and $\mathbb{G}$:
  $$\begin{array}{rcccl} \exp_P & : & \mathbb{Z}/\ell\mathbb{Z} & \longrightarrow & \mathbb{G} \\ & & k & \longmapsto & kP = \underbrace{P + P + \ldots + P}_{k \text{ times}} \end{array}$$
- The inverse map is the so-called discrete logarithm (in base P):
  $$\begin{array}{rcccll} \mathrm{dlog}_P = \exp_P^{-1} & : & \mathbb{G} & \longrightarrow & \mathbb{Z}/\ell\mathbb{Z} & \\ & & Q & \longmapsto & k & \text{such that } Q = kP \end{array}$$

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

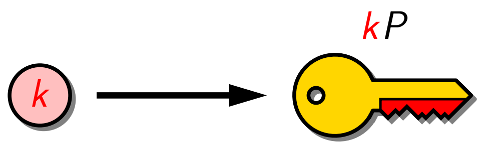
Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

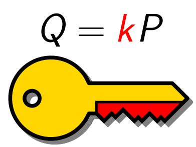


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

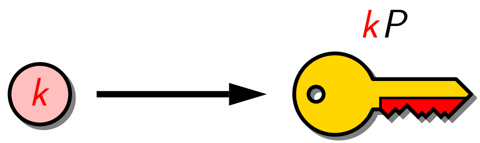


- Under a few conditions, the discrete logarithm can only be computed in exponential time (as far as we know):

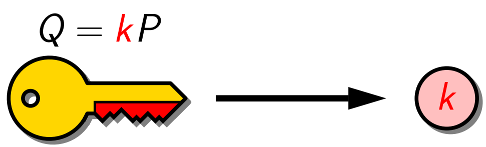


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:

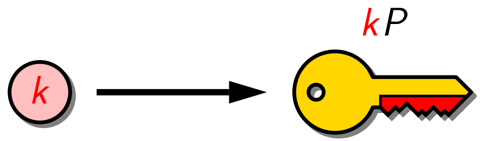


- Under a few conditions, the discrete logarithm can only be computed in exponential time (as far as we know):

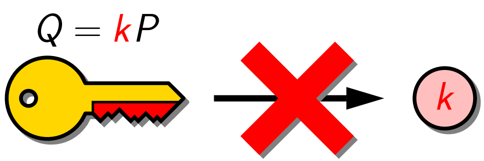


Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



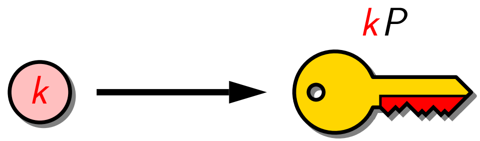
- Under a few conditions, the discrete logarithm can only be computed in exponential time (as far as we know):



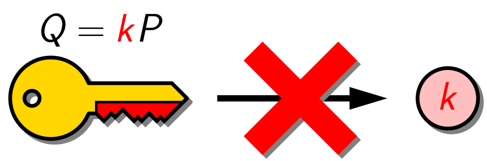
- That's a one-way function

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



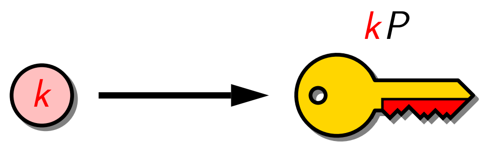
- Under a few conditions, the discrete logarithm can only be computed in exponential time (as far as we know):



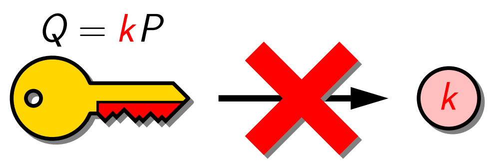
- That's a one-way function $\Rightarrow$ public-key cryptography!

Towards elliptic curve cryptography

- Scalar multiplication can be computed in polynomial time:



- Under a few conditions, the discrete logarithm can only be computed in exponential time (as far as we know):



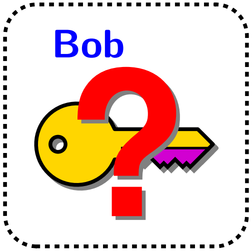
- That's a one-way function $\Rightarrow$ public-key cryptography!
  - secret key: an integer k in $\mathbb{Z}/\ell\mathbb{Z}$
  - public key: the point kP in $\mathbb{G} \subseteq E(\mathbb{F}_q)$

Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel

Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?



Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

Example protocol: EC Diffie–Hellman key exchange

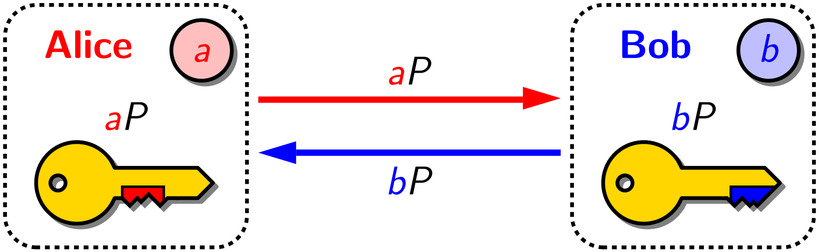
- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?

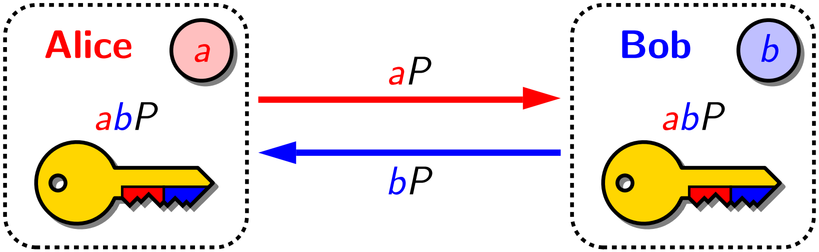
Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?



Example protocol: EC Diffie–Hellman key exchange

- Alice and Bob want to establish a secure communication channel
- How can they decide upon a shared secret key over a public channel?



Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)
- etc.

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)
- etc.
- Other important operations might be required, such as pairings

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)
- etc.
- Other important operations might be required, such as pairings
- Several algorithmic and arithmetic layers:

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)
- etc.
- Other important operations might be required, such as pairings
- Several algorithmic and arithmetic layers:
  - scalar multiplication

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)
- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)
- etc.
- Other important operations might be required, such as pairings
- Several algorithmic and arithmetic layers:
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)

Central operation: the scalar multiplication

- Elliptic curve Diffie–Hellman (ECDH):
  - Alice: $Q_A \leftarrow aP$ and $K \leftarrow aQ_B$ (2 scalar mults)
  - Bob: $Q_B \leftarrow bP$ and $K \leftarrow bQ_A$ (2 scalar mults)

- Elliptic curve Digital Signature Algorithm (ECDSA):
  - Alice (KeyGen): $Q_A \leftarrow aP$ (1 scalar mult)
  - Alice (Sign): $R \leftarrow kP$ (1 scalar mult)
  - Bob (Verify): $R' \leftarrow uP + vQ_A$ (2 scalar mults)

- etc.

- Other important operations might be required, such as pairings

- Several algorithmic and arithmetic layers:
  - scalar multiplication
  - elliptic curve arithmetic (point addition, point doubling, etc.)
  - finite field arithmetic (addition, multiplication, inversion, etc.)

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

- Available open-source hardware implementations of ECC:

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

- Available open-source hardware implementations of ECC:
  - implementation of NaCl's `crypto_box` (Ed25519 + Salsa20 + Poly1305) in 29.3 to 32.6 kGE [Hutter *et al.*, 2015]

Available implementations

- There already exist several free-software, open-source implementations of ECC (or of useful layers thereof):
  - at the protocol level:
    GnuPG, OpenSSL, GnuTLS, OpenSSH, cryptlib, etc.
  - at the cryptographic primitive level:
    RELIC, NaCl (Ed25519), crypto++, etc.
  - at the curve arithmetic level: PARI, Sage (not for crypto!)
  - at the field arithmetic level: MPFQ, GF2X, NTL, GMP, etc.

- Available open-source hardware implementations of ECC:
  - implementation of NaCl's `crypto_box` (Ed25519 + Salsa20 + Poly1305) in 29.3 to 32.6 kGE [Hutter *et al.*, 2015]
  - PAVOIS project: ECC cryptoprocessor designed to evaluate algorithmic and arithmetic protections against side-channel attacks [See A. Tisserand's talk]

Outline

- Some encryption mechanisms
- Elliptic curve cryptography
- Scalar multiplication
- Elliptic curve arithmetic
- Finite field arithmetic

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

- Size of ℓ (and k) for crypto applications: from 250 to 500 bits

Scalar multiplication

- Given k in $\mathbb{Z}/\ell\mathbb{Z}$ and P in $\mathbb{G} \subseteq E(\mathbb{F}_q)$, we want to compute

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

- Size of ℓ (and k) for crypto applications: from 250 to 500 bits
- Repeated addition, in $O(k)$ complexity, is out of the question!

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k
  - start from the most significant bit of k
  - double current result at each step
  - add P if the corresponding bit of k is 1

Double-and-add algorithm

- Available operations on $E(\mathbb{F}_q)$:
  - point addition: $(Q, R) \mapsto Q + R$
  - point doubling: $Q \mapsto 2Q = Q + Q$
- Idea: iterative algorithm based on the binary expansion of k
  - start from the most significant bit of k
  - double current result at each step
  - add P if the corresponding bit of k is 1
  - same principle as binary exponentiation

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
$T \leftarrow 2T$
if $k_i = 1$**:**
$T \leftarrow T + P$
return T

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n - 1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

$$T = \qquad = \quad \mathcal{O}$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (\underline{1}10101111)_2$

$T = \quad P \quad\quad = \quad P$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
$T \leftarrow 2T$
if $k_i = 1$**:**
$T \leftarrow T + P$
return T

- Example: $k = 431 = (1\underline{1}0101111)_2$

$$T = \quad P \cdot 2 \quad = \quad 2P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (1\underline{1}0101111)_2$

$$T = \quad P \cdot 2 + P \quad = \quad 3P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
    $T \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ **downto** 0**:**
        $T \leftarrow 2T$
        if $k_i = 1$**:**
            $T \leftarrow T + P$
    return T

- Example: $k = 431 = (11\underline{0}101111)_2$

$$T = \quad (P \cdot 2 + P) \cdot 2 \quad = \quad 6P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110\underline{1}01111)_2$

$$T = \quad (P \cdot 2 + P) \cdot 2^2 \qquad = \quad 12P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110\underline{1}01111)_2$

$$T = \quad (P \cdot 2 + P) \cdot 2^2 + P \quad = \quad 13P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110\underline{1}01111)_2$

$$T = ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2 = 26P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

$$T = ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 = 52P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

$$
\begin{array}{l}
\textbf{function } \text{scalar-mult}(k, P)\textbf{:} \\
\quad T \leftarrow \mathcal{O} \\
\quad \textbf{for } i \leftarrow n-1 \textbf{ downto } 0\textbf{:} \\
\quad\quad T \leftarrow 2T \\
\quad\quad \textbf{if } k_i = 1\textbf{:} \\
\quad\quad\quad T \leftarrow T + P \\
\quad \textbf{return } T
\end{array}
$$

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = \quad ((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P \qquad = \quad 53P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 \qquad = 106P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**

$T \leftarrow \mathcal{O}$

for $i \leftarrow n-1$ **downto** 0**:**

$T \leftarrow 2T$

if $k_i = 1$**:**

$T \leftarrow T + P$

return T

- Example: $k = 431 = (110101\underline{1}11)_2$

$$T = (((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P \qquad = 107P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101\underline{1}1)_2$

$$T = ((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 \qquad = 214P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (11010111\underline{1}1)_2$

$$T = \; ((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P \qquad = 215P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (11010111\underline{1})_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 \qquad = 430P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (11010111\underline{1})_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \dots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

```
function scalar-mult(k, P):
    T ← O
    for i ← n − 1 downto 0:
        T ← 2T
        if k_i = 1:
            T ← T + P
    return T
```

- Example: $k = 431 = (110101111)_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

Double-and-add algorithm

- Denoting by $(k_{n-1} \ldots k_1 k_0)_2$, with $n = \lceil \log_2 \ell \rceil$, the binary expansion of k:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- Example: $k = 431 = (110101111)_2$

$$T = (((((P \cdot 2 + P) \cdot 2^2 + P) \cdot 2^2 + P) \cdot 2 + P) \cdot 2 + P) \cdot 2 + P = 431P$$

- Complexity in $O(n) = O(\log_2 \ell)$ operations over $E(\mathbb{F}_q)$:
  - $n-1$ doublings, and
  - $n/2$ additions on average

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, $\ldots$, $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, $\ldots$, $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = \qquad = \quad \mathcal{O}$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (\underline{110}\,101\,111)_2 = (\underline{6}57)_{2^3}$

$$T = 6P \qquad = \quad 6P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,\underline{101}\,111)_2 = (6\underline{5}7)_{2^3}$

$$T = 6P \cdot 2^3 = 48P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,\underline{101}\,111)_2 = (6\underline{5}7)_{2^3}$

$$T = 6P \cdot 2^3 + 5P \qquad = 53P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,\underline{111})_2 = (65\underline{7})_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 \qquad = 424P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,\underline{111})_2 = (65\underline{7})_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \ldots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - $n - w$ doublings, and
  - $(1 - 2^{-w})n/w$ additions on average

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P, 3P, \dots, (2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - $n - w$ doublings, and
  - $(1 - 2^{-w})n/w$ additions on average
- Select w carefully so that precomputation cost does not become predominant

Windowed method

- Consider 2^w-ary expansion of k: i.e., split k into w-bit chunks
- Precompute $2P$, $3P$, ..., $(2^w - 1)P$:
  - $2^{w-1} - 1$ doublings, and
  - $2^{w-1} - 1$ additions
- Example with $w = 3$: $k = 431 = (110\,101\,111)_2 = (657)_{2^3}$

$$T = (6P \cdot 2^3 + 5P) \cdot 2^3 + 7P = 431P$$

- Complexity:
  - $n - w$ doublings, and
  - $(1 - 2^{-w})n/w$ additions on average
- Select w carefully so that precomputation cost does not become predominant
- Sliding window variant: half as many precomputations

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

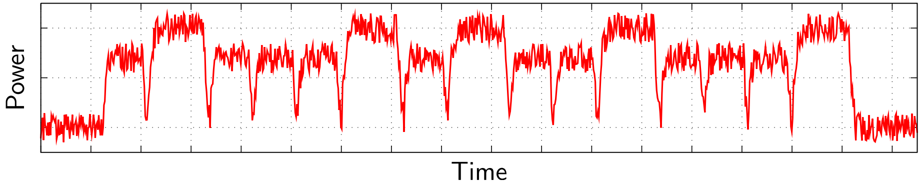
- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - simple power analysis (SPA) will leak bits of k

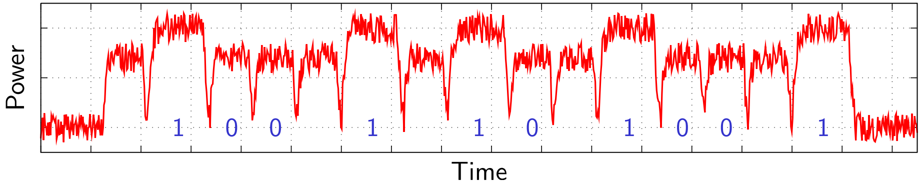


Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - simple power analysis (SPA) will leak bits of k

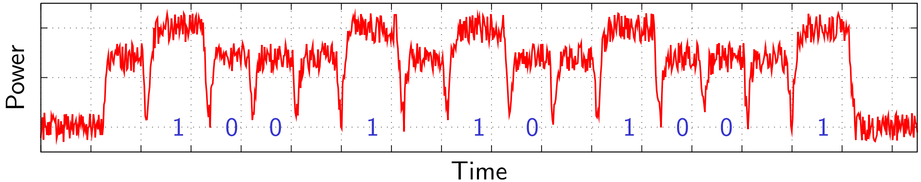


Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
    else:
      $Z \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - simple power analysis (SPA) will leak bits of k



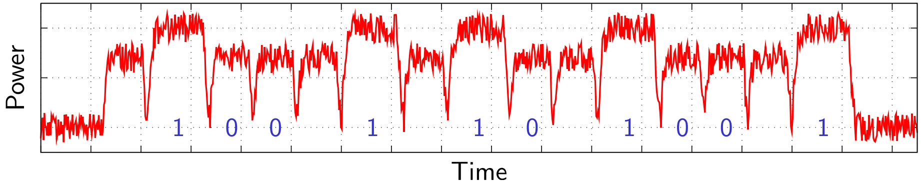
- Use double-and-add-always algorithm?

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
$T \leftarrow \mathcal{O}$
for $i \leftarrow n-1$ **downto** 0**:**
$T \leftarrow 2T$
if $k_i = 1$**:**
$T \leftarrow T + P$
else:
$Z \leftarrow T + P$
return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - simple power analysis (SPA) will leak bits of k



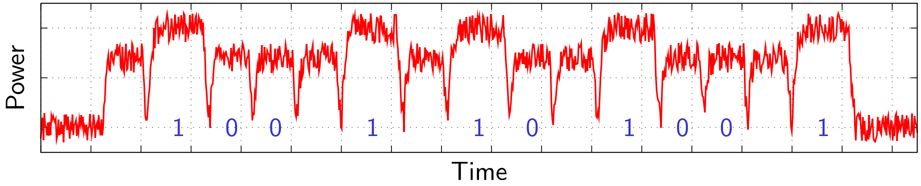
- Use double-and-add-always algorithm?
  - the result of the point addition is used if and only if $k_i = 1$

Security issues

- Back to the double-and-add algorithm:

function scalar-mult(k, P)**:**
  $T \leftarrow \mathcal{O}$
  for $i \leftarrow n-1$ **downto** 0**:**
    $T \leftarrow 2T$
    if $k_i = 1$**:**
      $T \leftarrow T + P$
    else:
      $Z \leftarrow T + P$
  return T

- At step i, point addition $T \leftarrow T + P$ is computed if and only if $k_i = 1$
  - careful timing analysis will reveal Hamming weight of secret k
  - simple power analysis (SPA) will leak bits of k



- Use double-and-add-always algorithm?
  - the result of the point addition is used if and only if $k_i = 1$

  $\Rightarrow$ vulnerable to fault attacks [See A. Tisserand's lecture]

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

  function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
  $T_0 \leftarrow T_0 + T_1$
  $T_1 \leftarrow 2T_1$
  else:
  $T_1 \leftarrow T_0 + T_1$
  $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2 T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2 T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

$$T_0 = \quad = \mathcal{O}$$
$$T_1 = P \quad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

  function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
  $T_0 \leftarrow T_0 + T_1$
  $T_1 \leftarrow 2T_1$
  else:
  $T_1 \leftarrow T_0 + T_1$
  $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = \quad = \mathcal{O}$$
$$T_1 = P \quad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = P \qquad = P$$
$$T_1 = P \qquad = P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (\underline{1}0011)_2$

$$T_0 = P \qquad = P$$
$$T_1 = P \cdot 2 \qquad = 2P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**

$T_0 \leftarrow \mathcal{O}$

$T_1 \leftarrow P$

for $i \leftarrow n-1$ **downto** 0**:**

if $k_i = 1$**:**

$T_0 \leftarrow T_0 + T_1$

$T_1 \leftarrow 2T_1$

else:

$T_1 \leftarrow T_0 + T_1$

$T_0 \leftarrow 2T_0$

return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$

- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \qquad = P$$

$$T_1 = P \cdot 2 \qquad = 2P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \qquad = P$$
$$T_1 = P \cdot 2 + P \qquad = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1\underline{0}011)_2$

$$T_0 = P \cdot 2 = 2P$$
$$T_1 = P \cdot 2 + P = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2 = 2P$$
$$T_1 = P \cdot 2 + P = 3P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

  function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
  $T_0 \leftarrow T_0 + T_1$
  $T_1 \leftarrow 2T_1$
  else:
  $T_1 \leftarrow T_0 + T_1$
  $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2 = 2P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10\underline{0}11)_2$

$$T_0 = P \cdot 2^2 = 4P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (100\underline{1}1)_2$

$$T_0 = P \cdot 2^2 = 4P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (100\underline{1}1)_2$

$$T_0 = P \cdot 2^2 + 5P = 9P$$
$$T_1 = P \cdot 2 + P + 2P = 5P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
  if $k_i = 1$**:**
    $T_0 \leftarrow T_0 + T_1$
    $T_1 \leftarrow 2T_1$
  else:
    $T_1 \leftarrow T_0 + T_1$
    $T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (100\underline{1}1)_2$

$$T_0 = P \cdot 2^2 + 5P = 9P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2 = 10P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
    $T_0 \leftarrow \mathcal{O}$
    $T_1 \leftarrow P$
    for $i \leftarrow n-1$ **downto** 0**:**
        if $k_i = 1$**:**
            $T_0 \leftarrow T_0 + T_1$
            $T_1 \leftarrow 2T_1$
        else:
            $T_1 \leftarrow T_0 + T_1$
            $T_0 \leftarrow 2T_0$
    return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$\begin{aligned} T_0 &= P \cdot 2^2 + 5P &&= 9P \\ T_1 &= (P \cdot 2 + P + 2P) \cdot 2 &&= 10P \end{aligned}$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
  $T_0 \leftarrow \mathcal{O}$
  $T_1 \leftarrow P$
  for $i \leftarrow n-1$ **downto** 0**:**
    if $k_i = 1$**:**
      $T_0 \leftarrow T_0 + T_1$
      $T_1 \leftarrow 2T_1$
    else:
      $T_1 \leftarrow T_0 + T_1$
      $T_0 \leftarrow 2T_0$
  return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2 = 10P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

function scalar-mult(k, P)**:**
$T_0 \leftarrow \mathcal{O}$
$T_1 \leftarrow P$
for $i \leftarrow n-1$ **downto** 0**:**
if $k_i = 1$**:**
$T_0 \leftarrow T_0 + T_1$
$T_1 \leftarrow 2T_1$
else:
$T_1 \leftarrow T_0 + T_1$
$T_0 \leftarrow 2T_0$
return T_0

- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (1001\underline{1})_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P$$

The Montgomery ladder

- Algorithm proposed by Montgomery in 1987:

```
function scalar-mult(k, P):
    T_0 ← O
    T_1 ← P
    for i ← n − 1 downto 0:
        if k_i = 1:
            T_0 ← T_0 + T_1
            T_1 ← 2T_1
        else:
            T_1 ← T_0 + T_1
            T_0 ← 2T_0
    return T_0
```

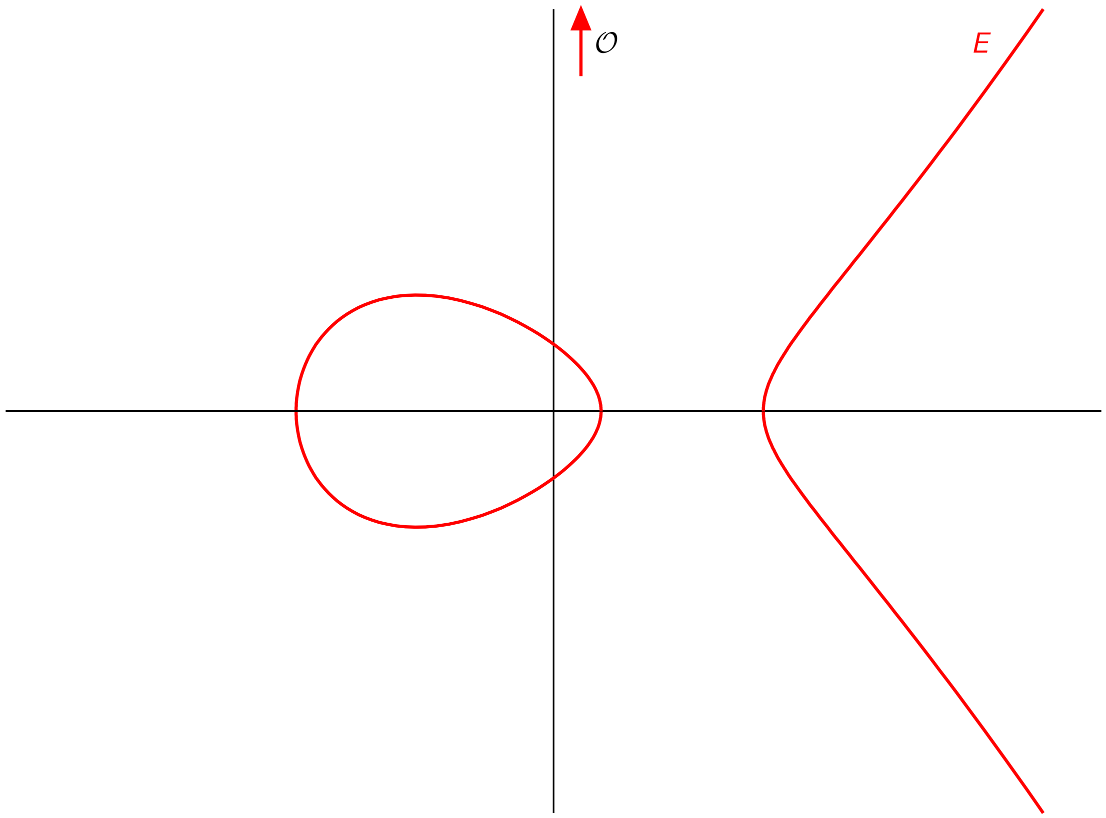
- Properties:
  - perform one addition and one doubling at each step
  - ensure that both results are used in the next step
  - loop invariant: $T_1 = T_0 + P$
- Example: $k = 19 = (10011)_2$

$$T_0 = P \cdot 2^2 + 5P + 10P = 19P$$
$$T_1 = (P \cdot 2 + P + 2P) \cdot 2^2 = 20P$$

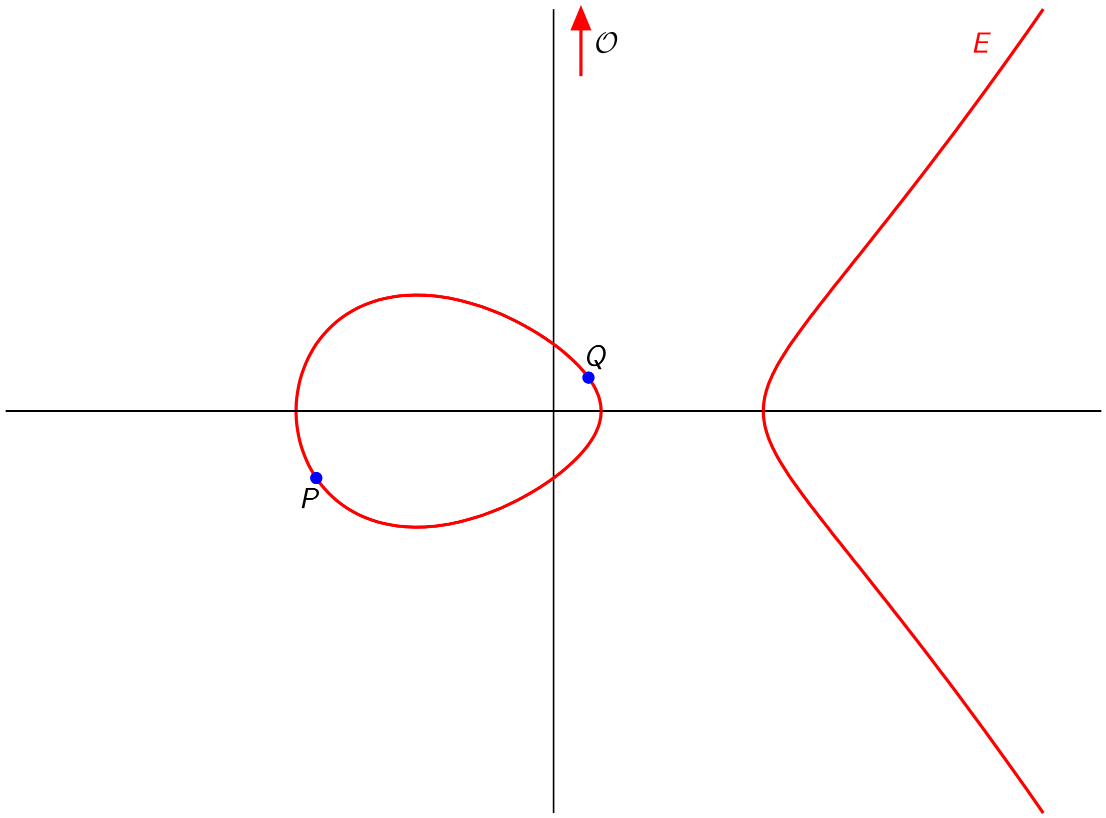
Outline

- Some encryption mechanisms
- Elliptic curve cryptography
- Scalar multiplication
- Elliptic curve arithmetic
- Finite field arithmetic

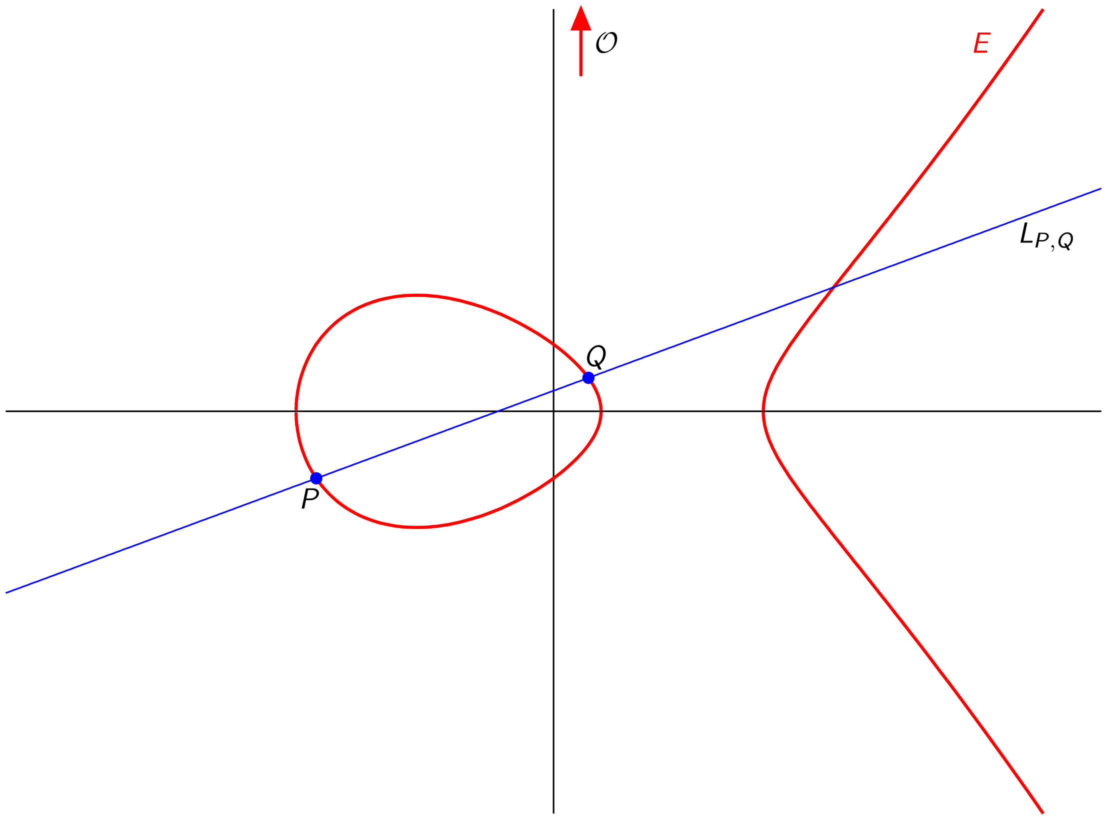
Addition and doubling



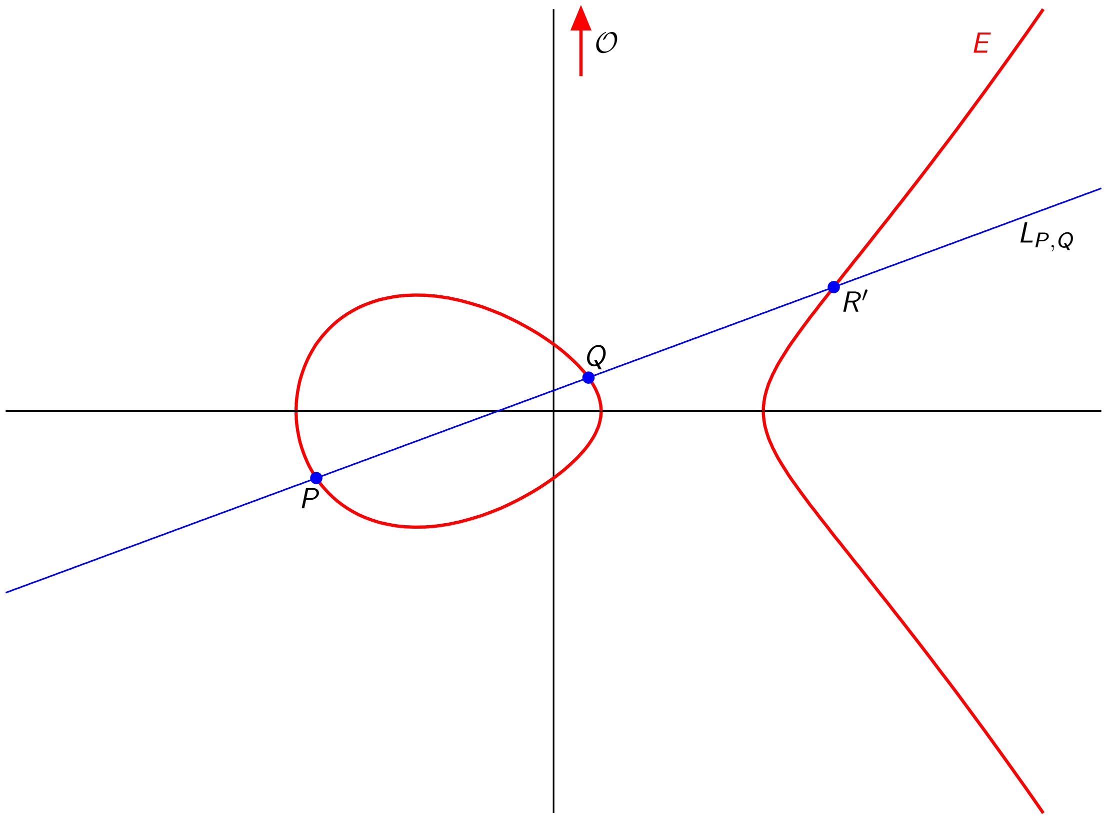
Addition and doubling



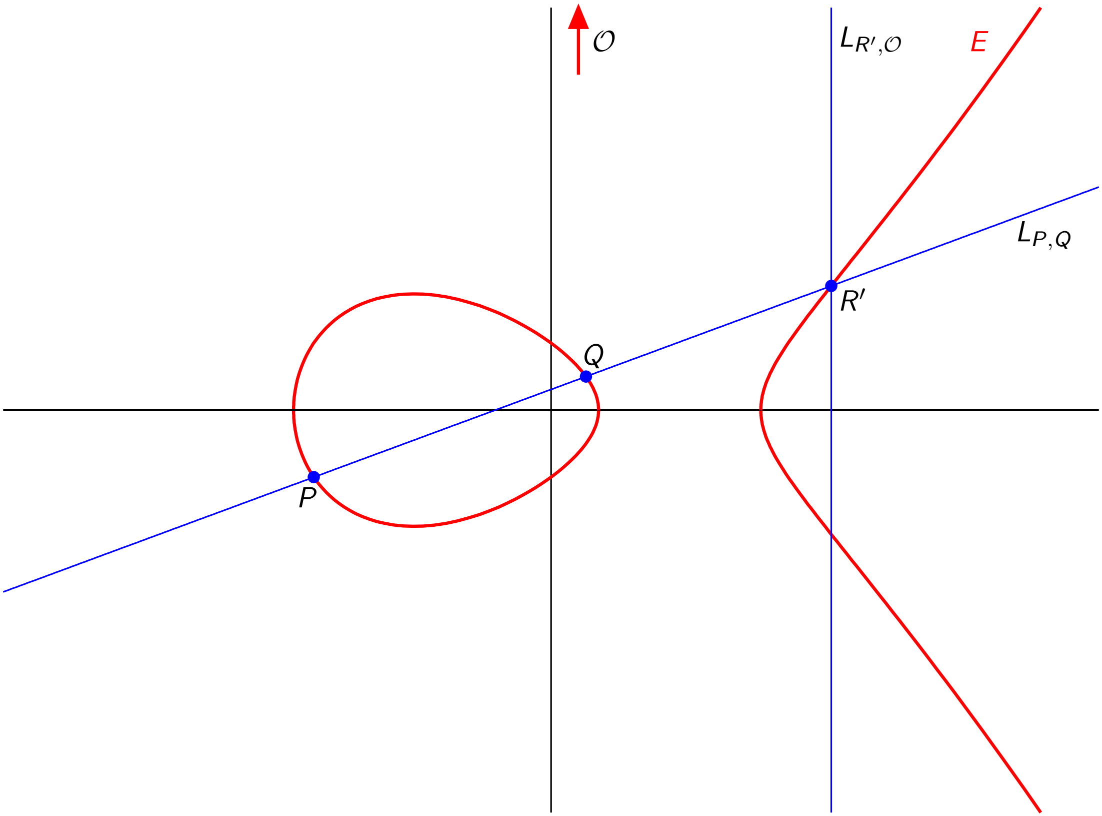
Addition and doubling



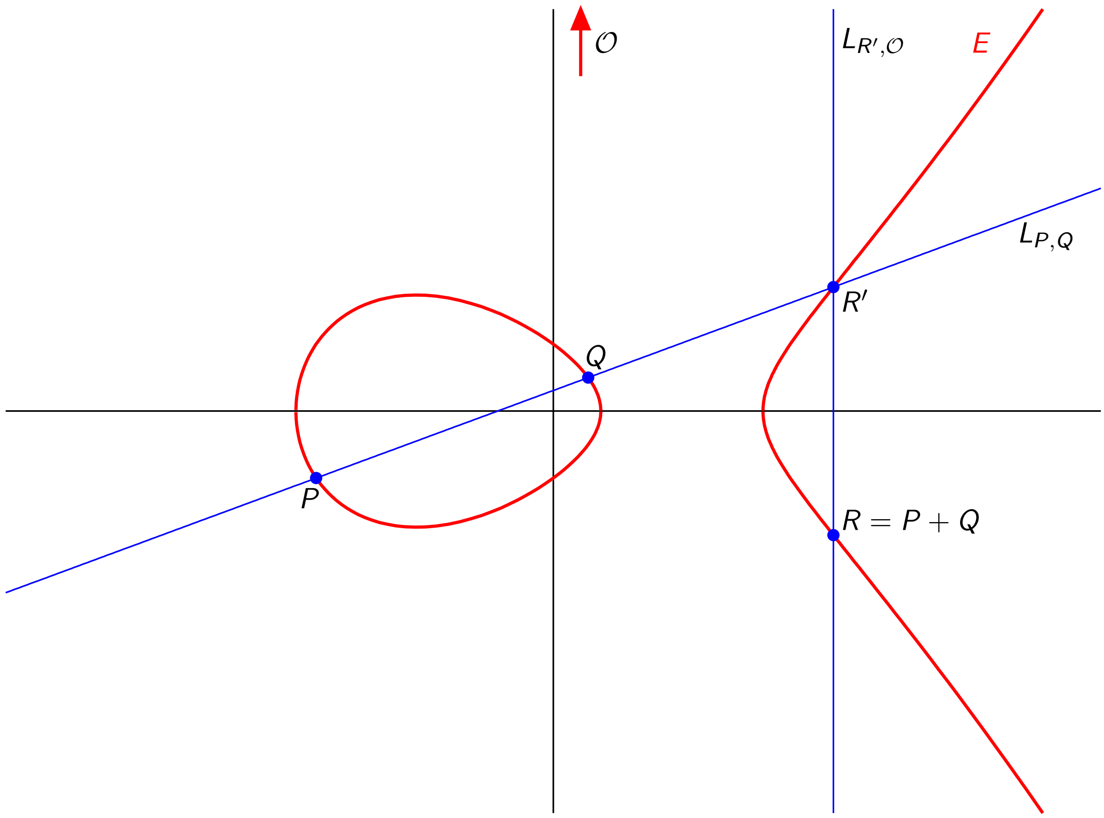
Addition and doubling



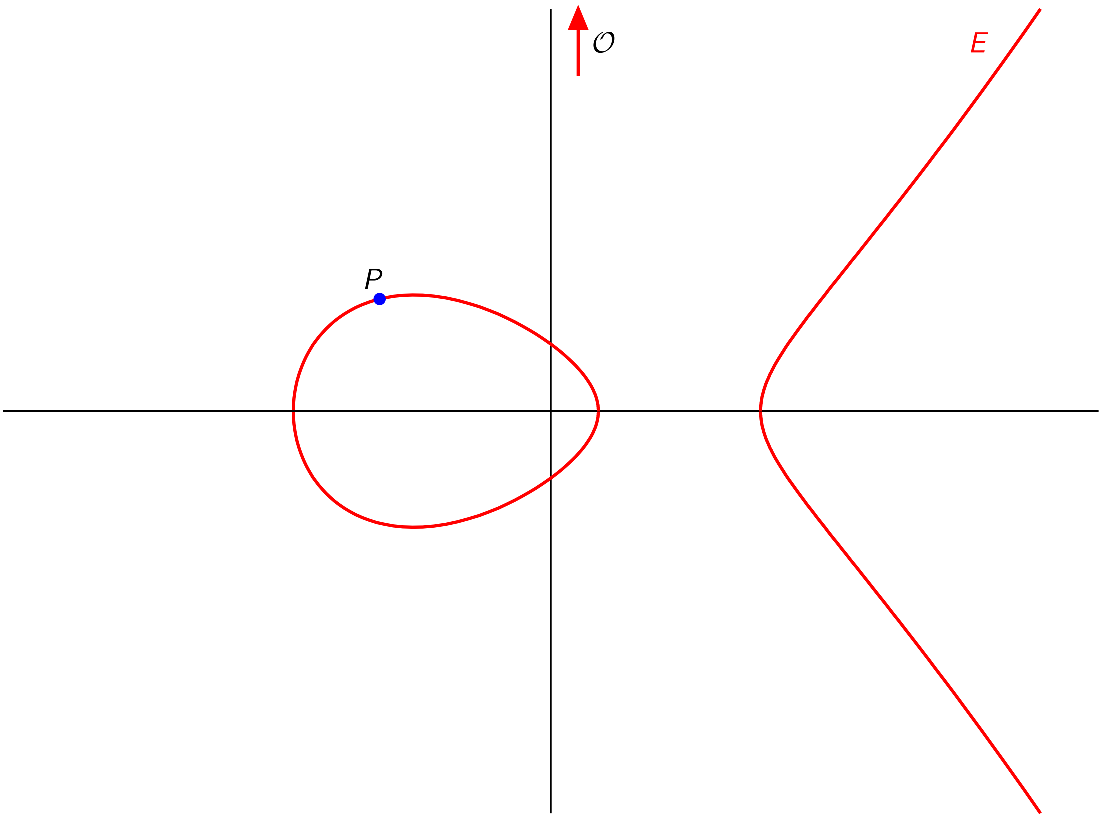
Addition and doubling



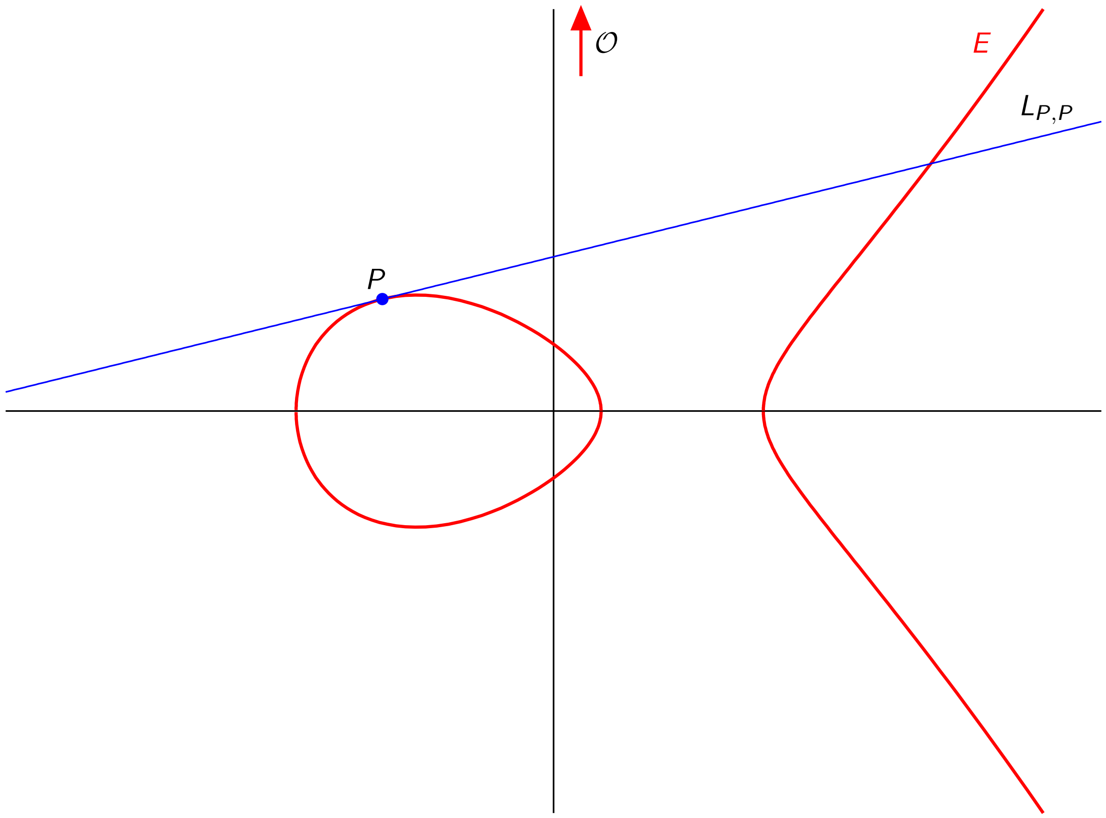
Addition and doubling



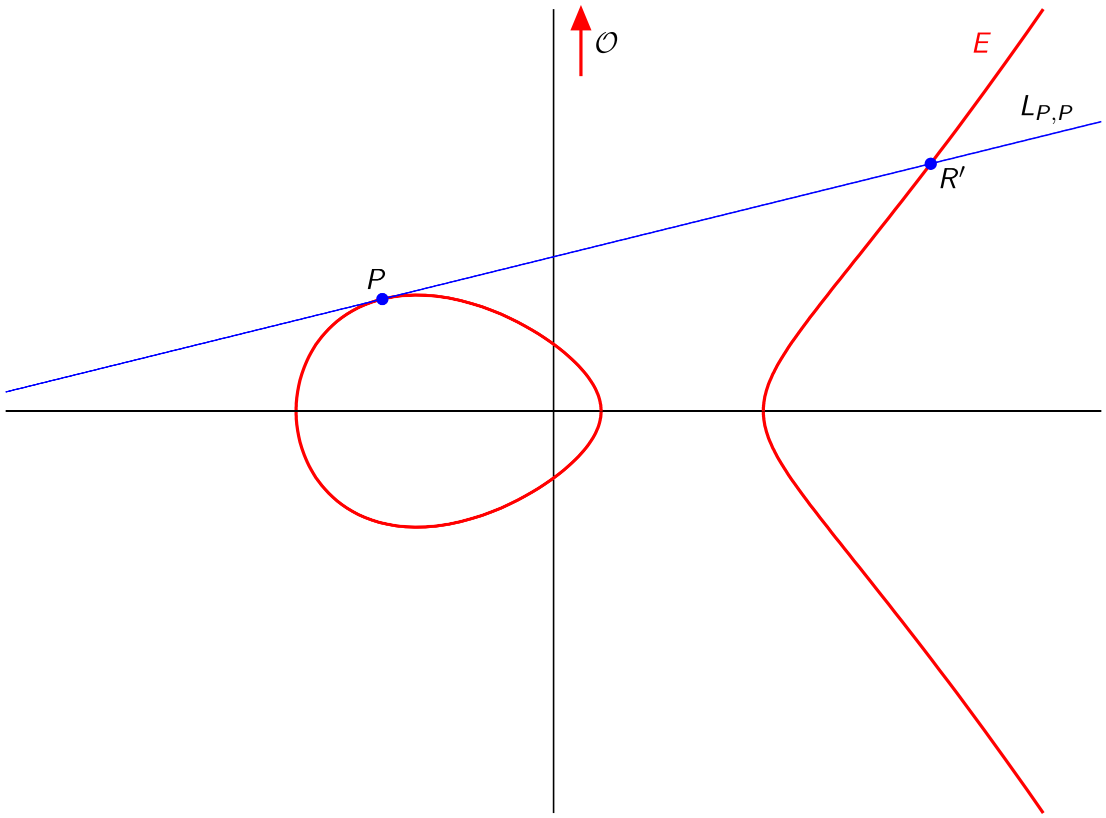
Addition and doubling



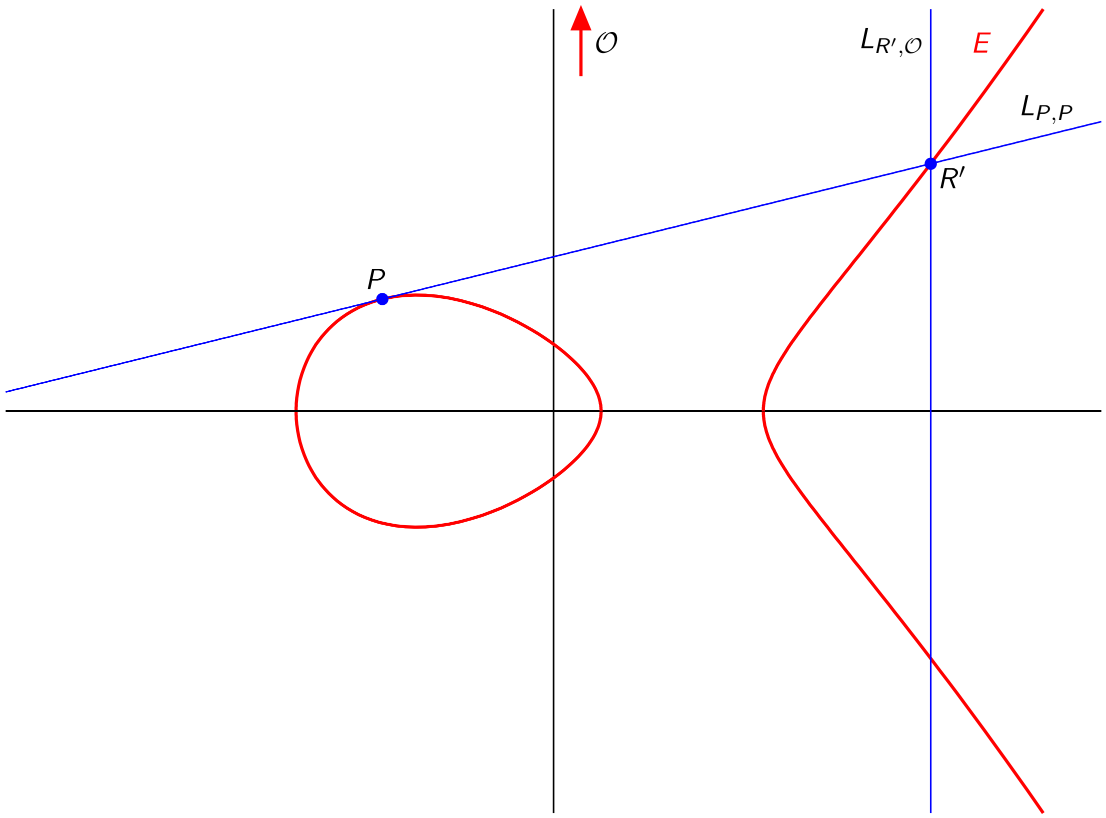
Addition and doubling



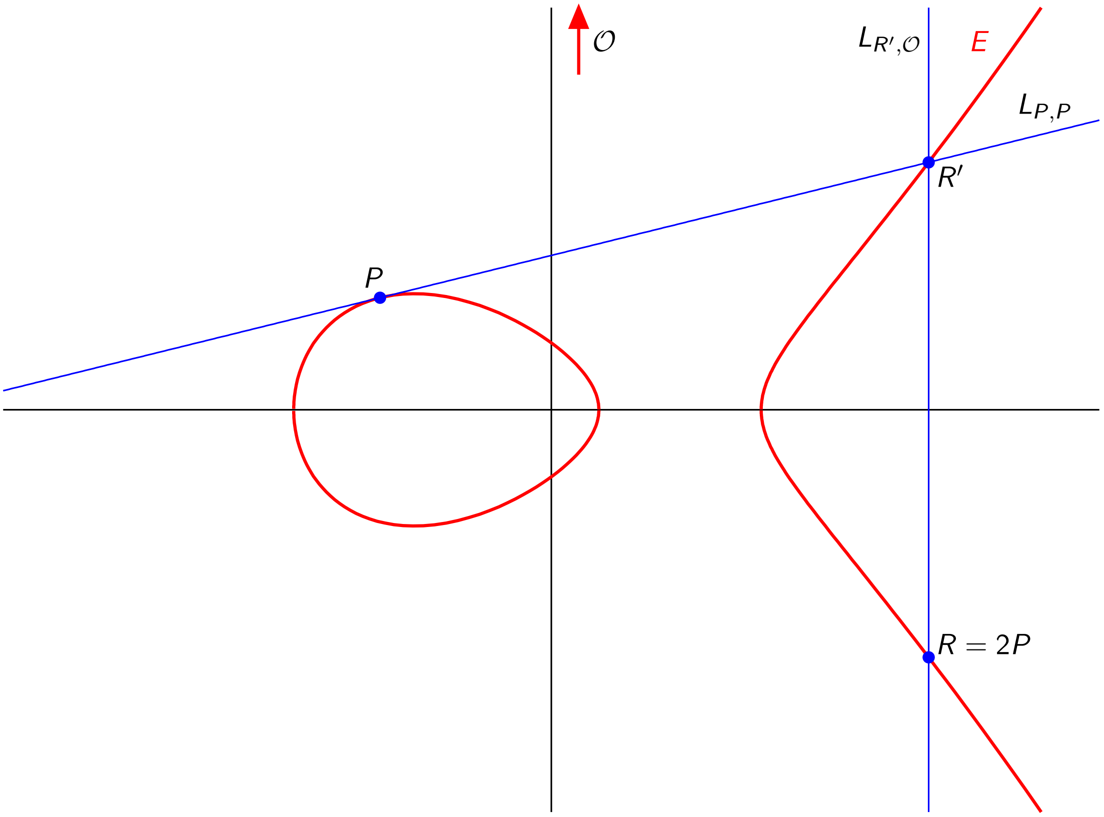
Addition and doubling



Addition and doubling



Addition and doubling



Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (addition), or} \\ \dfrac{3x_P^2 + A}{2y_P} & \text{if } P = Q \text{ (doubling)} \end{cases}$$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q) \backslash \{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (addition), or} \\ \dfrac{3x_P^2 + A}{2y_P} & \text{if } P = Q \text{ (doubling)} \end{cases}$$

- Cost (number of multiplications, squarings, and inversions in $\mathbb{F}_q$):

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

  where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (addition), or} \\ \dfrac{3x_P^2 + A}{2y_P} & \text{if } P = Q \text{ (doubling)} \end{cases}$$

- Cost (number of multiplications, squarings, and inversions in $\mathbb{F}_q$):
  - addition: $2\text{M} + 1\text{S} + 1\text{I}$

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (addition), or} \\ \dfrac{3x_P^2 + A}{2y_P} & \text{if } P = Q \text{ (doubling)} \end{cases}$$

- Cost (number of multiplications, squarings, and inversions in $\mathbb{F}_q$):
  - addition: 2M + 1S + 1I
  - doubling: 2M + 2S + 1I

Addition and doubling formulae

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q) \in E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ (affine coordinates)
- The opposite of P is $-P = (x_P, -y_P)$
- If $P \neq -Q$, then $P + Q = R = (x_R, y_R)$ with

$$x_R = \lambda^2 - x_P - x_Q \qquad \text{and} \qquad y_R = \lambda(x_P - x_R) - y_P$$

  where

$$\lambda = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P} & \text{if } P \neq Q \text{ (addition), or} \\ \dfrac{3x_P^2 + A}{2y_P} & \text{if } P = Q \text{ (doubling)} \end{cases}$$

- Cost (number of multiplications, squarings, and inversions in $\mathbb{F}_q$):
  - addition: 2M + 1S + 1I
  - doubling: 2M + 2S + 1I

  $\Rightarrow$ field inversion is expensive!

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2 Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S
- And many others: modified jacobian coordinates, López–Dahab (over $\mathbb{F}_{2^n}$), etc.

Other coordinate systems

$$E/\mathbb{F}_q : y^2 = x^3 + Ax + B$$

- One can use other coordinate systems which provide more efficient formulae
- Projective coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z, Y/Z)$

$$E/\mathbb{F}_q : Y^2Z = X^3 + AXZ^2 + BZ^3$$

  - idea: get rid of the inversion over $\mathbb{F}_q$ by using Z as the denominator
  - addition: 12M + 2S
  - doubling: 7M + 5S
- Jacobian coordinates: points $(X : Y : Z)$ with $(x, y) = (X/Z^2, Y/Z^3)$

$$E/\mathbb{F}_q : Y^2 = X^3 + AXZ^4 + BZ^6$$

  - addition: 12M + 4S
  - doubling: 4M + 6S
- And many others: modified jacobian coordinates, López–Dahab (over $\mathbb{F}_{2^n}$), etc.
- Explicit-Formula Database (by Bernstein and Lange):

`http://hyperelliptic.org/EFD/`

Outline

- Some encryption mechanisms
- Elliptic curve cryptography
- Scalar multiplication
- Elliptic curve arithmetic
- Finite field arithmetic

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with p an n-bit prime and n between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with n prime and between 250 and 500

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with p an n-bit prime and n between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with n prime and between 250 and 500 (... still secure?)

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$
- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with p an n-bit prime and n between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with n prime and between 250 and 500 (... still secure?)
- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$

- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with p an n-bit prime and n between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with n prime and between 250 and 500 (... still secure?)

- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)

- ... on w-bit words:
  - $w = 32$ or 64 on CPUs
  - $w = 8$ or 16 bits on microcontrollers
  - a bit more flexibility in hardware
    (but integer arithmetic with $w > 64$ bits is hard!)

Implementing finite field arithmetic

- Group law over $E(\mathbb{F}_q)$ requires:
  - additions / subtractions over $\mathbb{F}_q$
  - multiplications / squarings over $\mathbb{F}_q$
  - a few inversions over $\mathbb{F}_q$
- Typical finite fields $\mathbb{F}_q$:
  - prime field $\mathbb{F}_p$, with p an n-bit prime and n between 250 and 500 bits
  - binary field $\mathbb{F}_{2^n}$, with n prime and between 250 and 500 (... still secure?)
- What we have at our disposal:
  - basic integer arithmetic (addition, multiplication)
  - left and right shifts
  - bitwise logic operations (bitwise NOT, AND, etc.)
- ... on w-bit words:
  - $w = 32$ or 64 on CPUs
  - $w = 8$ or 16 bits on microcontrollers
  - a bit more flexibility in hardware
    (but integer arithmetic with $w > 64$ bits is hard!)

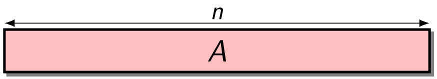
  $\Rightarrow$ elements of $\mathbb{F}_q$ represented using several words

Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime

Multiprecision representation

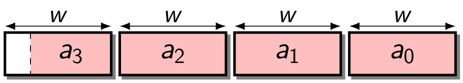
- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P



Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

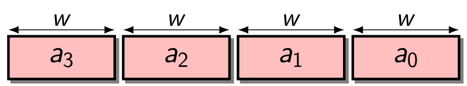
$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$



Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

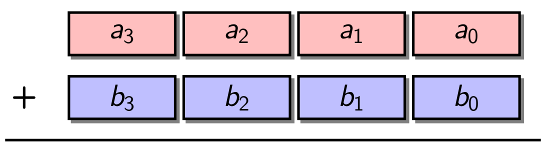


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:

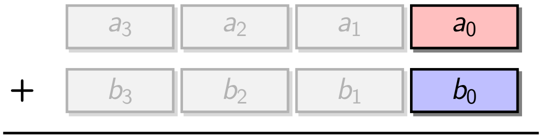


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition

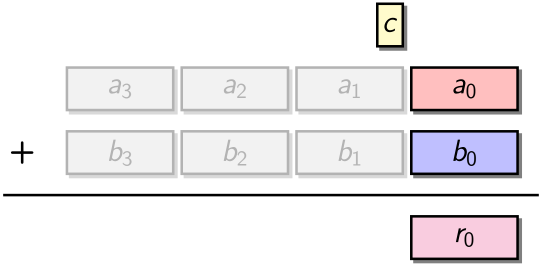


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1} 2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

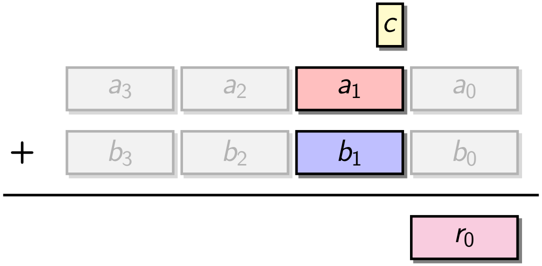


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

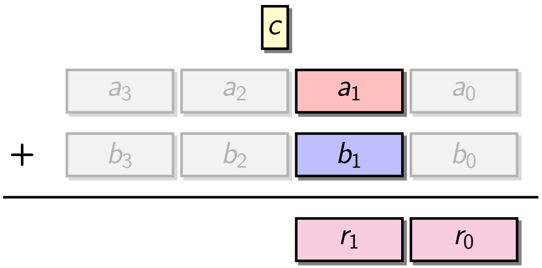


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

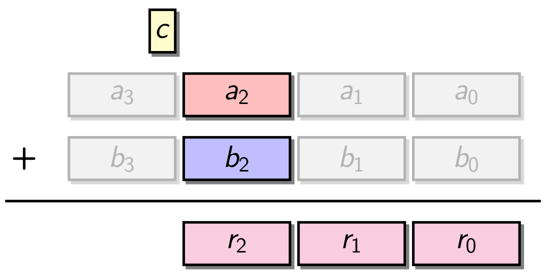


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

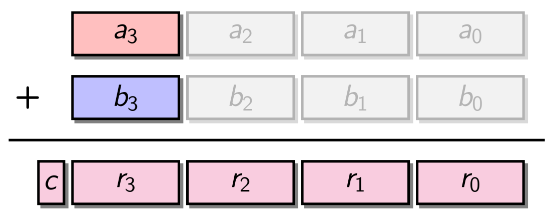


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

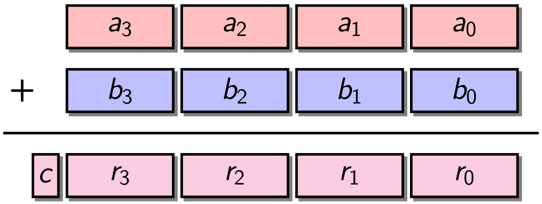


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry

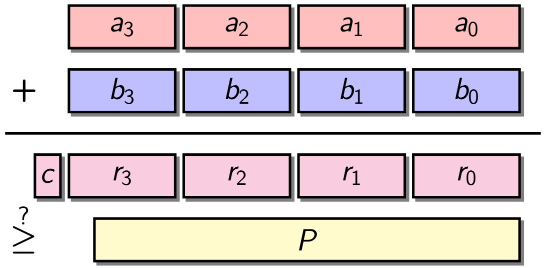


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

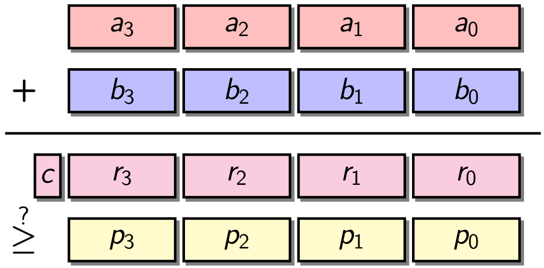


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

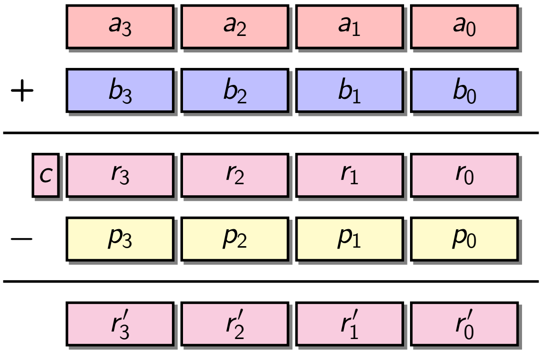


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)

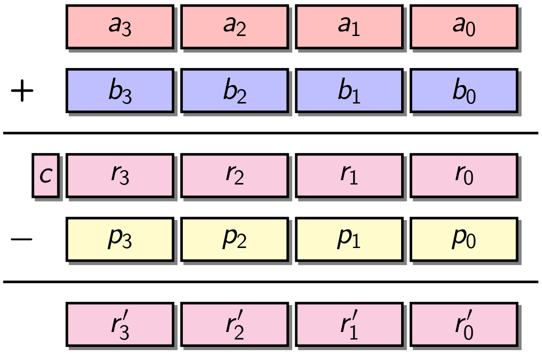


Multiprecision representation

- Consider $A \in \mathbb{F}_P$, with P an n-bit prime
  - represent A as an integer modulo P
  - split A into $k = \lceil n/w \rceil$ w-bit words (or limbs), a_{k-1}, ..., a_1, a_0:

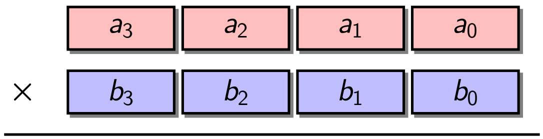
$$A = a_{k-1}2^{(k-1)w} + \cdots + a_1 2^w + a_0$$

- Addition of A and $B \in \mathbb{F}_P$:
  - right-to-left word-wise addition
  - need to propagate carry
  - might need reduction modulo P: compare then subtract (in constant time!)
  - lazy reduction: if $kw > n$, do not reduce after each addition



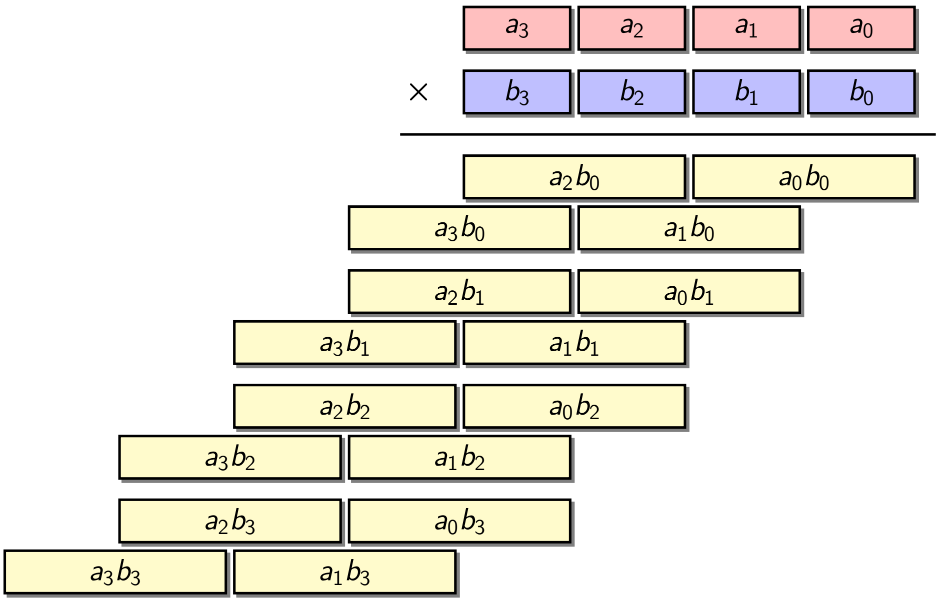
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:



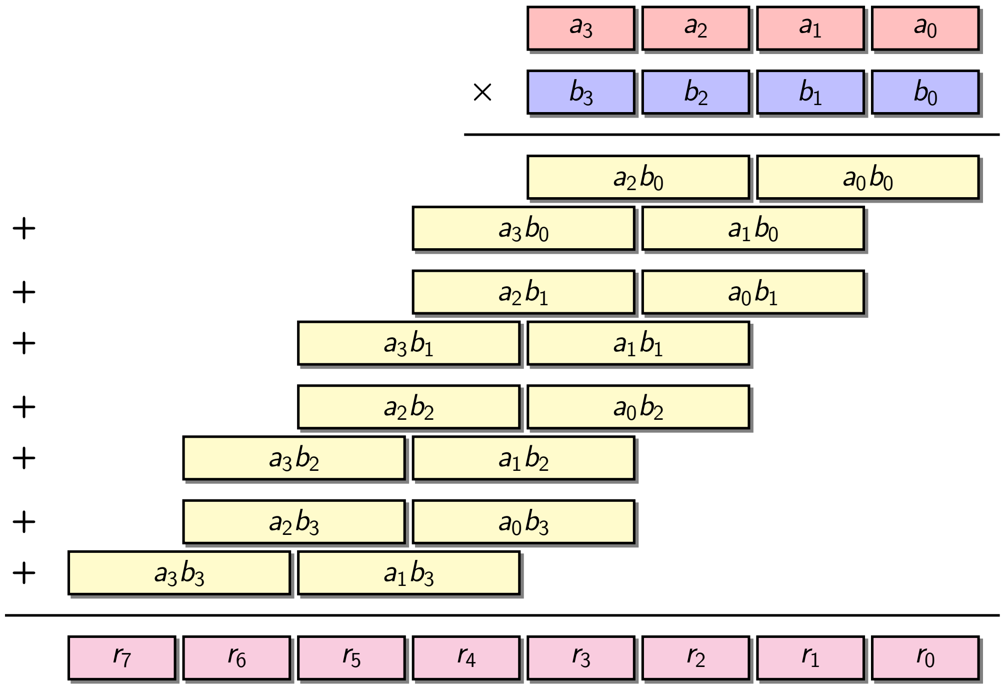
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:
  - schoolbook method: k^2 w-by-w-bit products



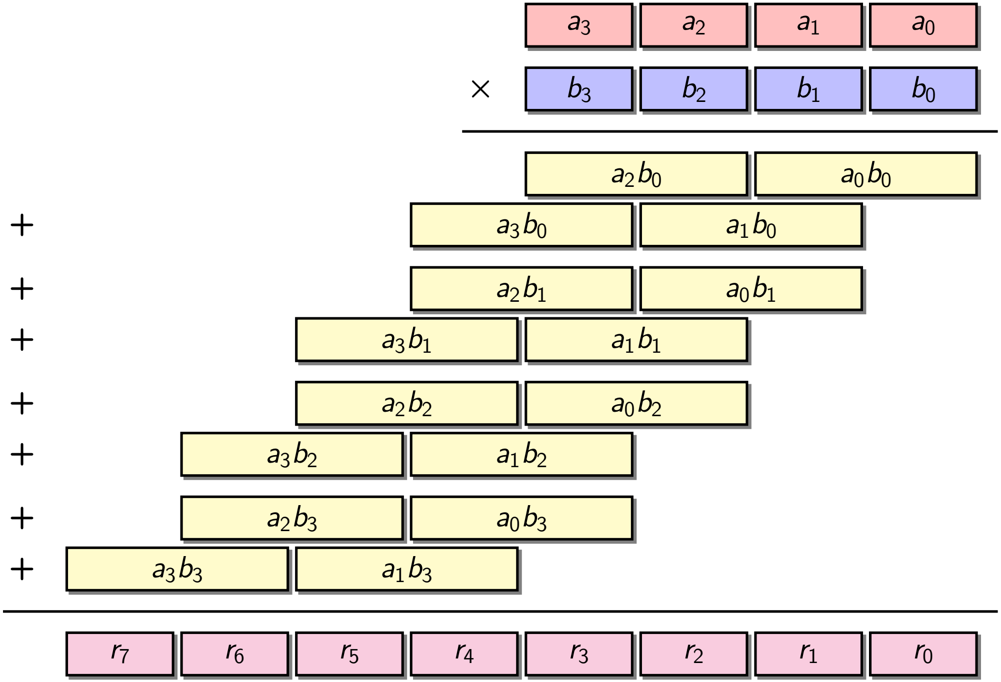
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:
  - schoolbook method: k^2 w-by-w-bit products



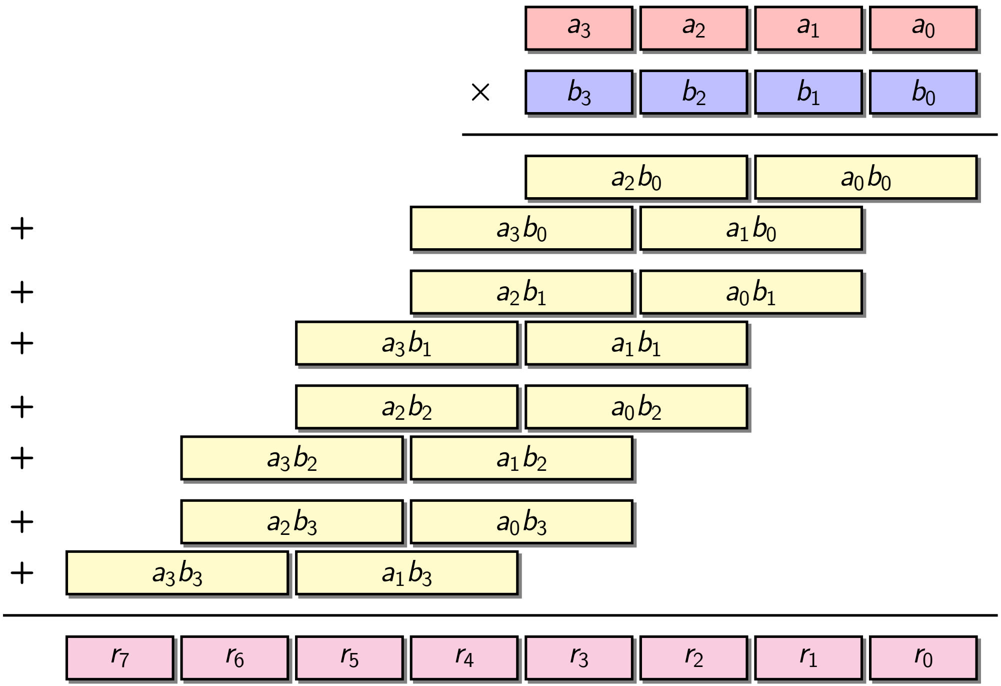
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:
  - schoolbook method: k^2 w-by-w-bit products
  - subquadratic algorithms (e.g., Karatsuba) when k is large



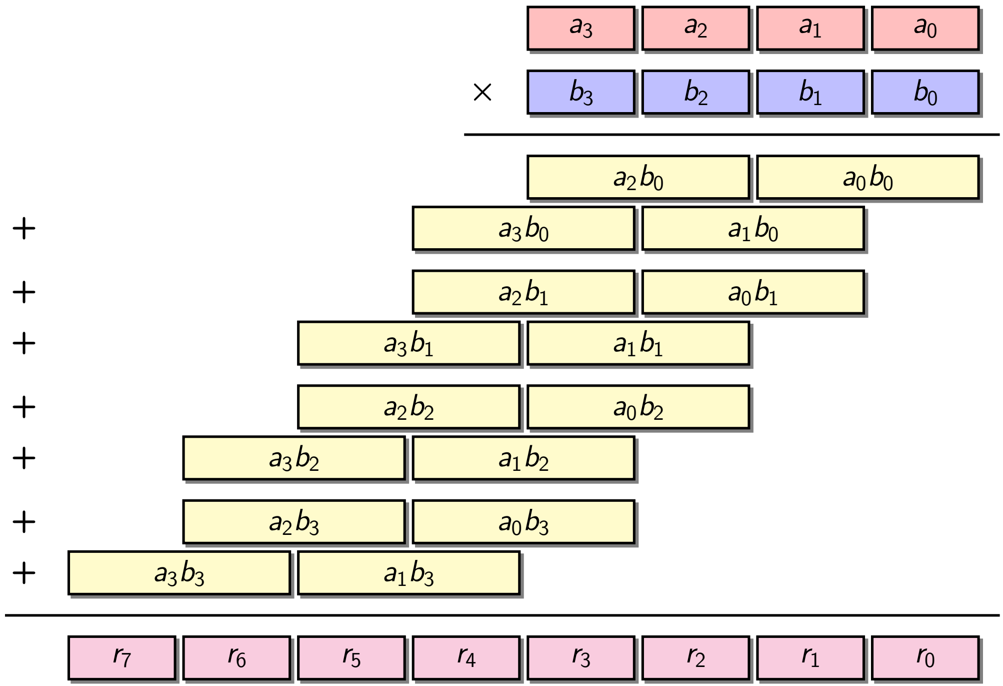
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:
  - schoolbook method: k^2 w-by-w-bit products
  - subquadratic algorithms (e.g., Karatsuba) when k is large
  - final product fits into $2k$ words $\rightarrow$ requires reduction modulo P (see later)



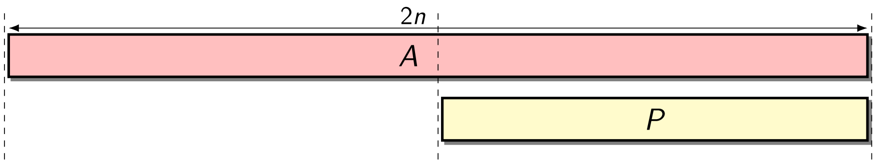
MP multiplication

- Multiplication of A and $B \in \mathbb{F}_P$:
  - schoolbook method: k^2 w-by-w-bit products
  - subquadratic algorithms (e.g., Karatsuba) when k is large
  - final product fits into $2k$ words $\rightarrow$ requires reduction modulo P (see later)
  - should run in constant time (for fixed P)!



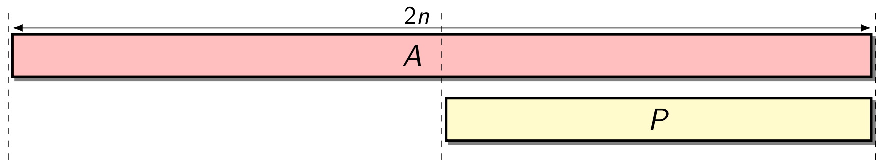
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$



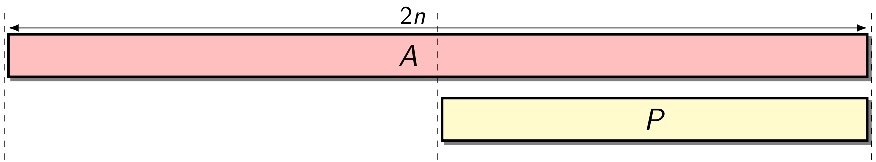
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)



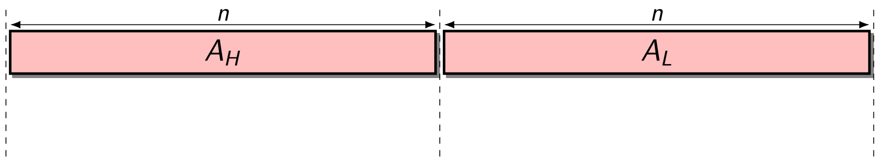
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$



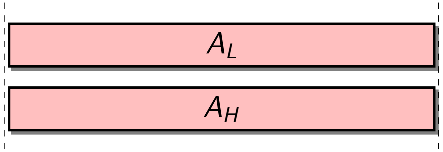
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$



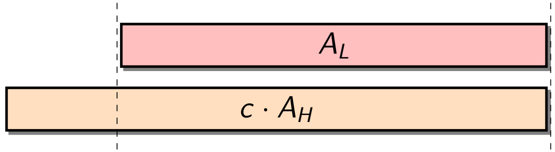
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)



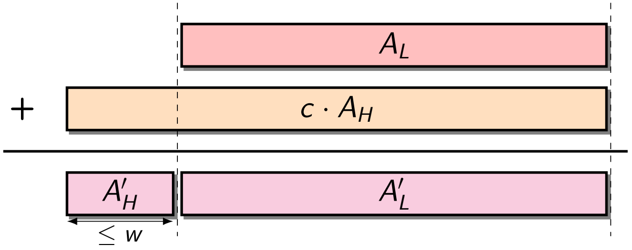
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)



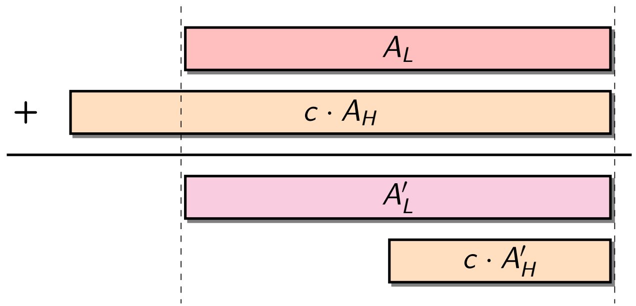
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



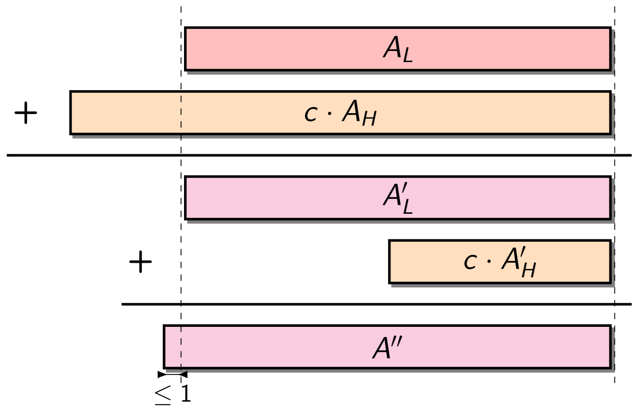
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



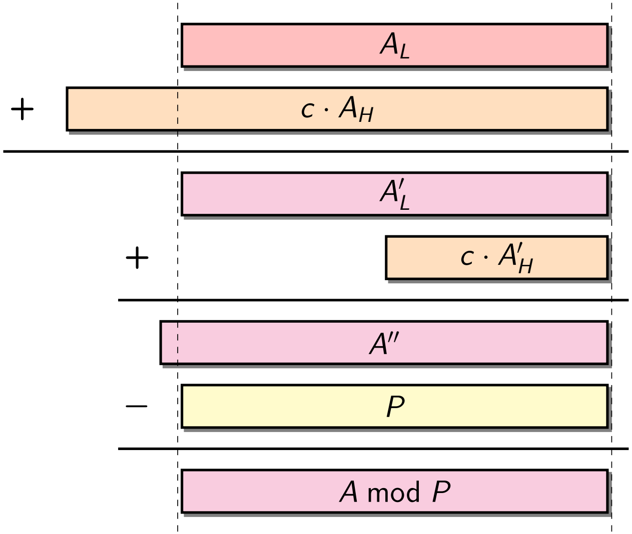
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)



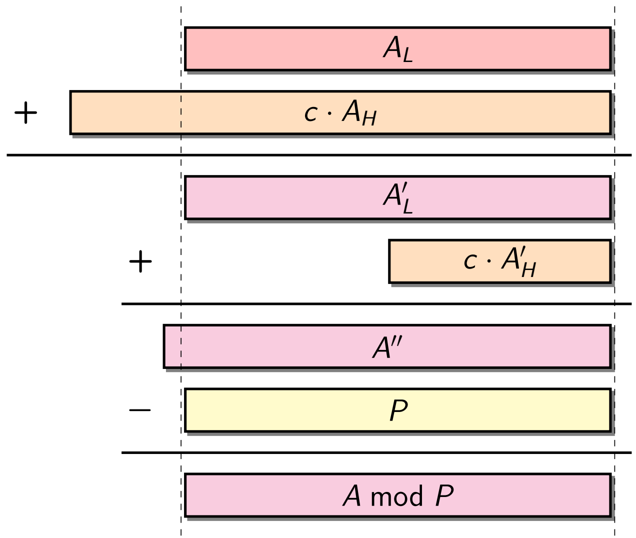
MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)
  - final subtraction might be necessary



MP modular reduction

- Given an integer $A < P^2$ (on $2k$ words), compute $R = A \bmod P$
- Easy case: P is a pseudo-Mersenne prime $P = 2^n - c$ with c "small" (e.g., $< 2^w$)
  - then $2^n \equiv c \pmod{P}$
  - split A wrt. 2^n: $A = A_H 2^n + A_L$
  - compute $A' \leftarrow c \cdot A_H + A_L$ (one $1 \times k$-word multiplication)
  - rinse & repeat (one 1×1-word multiplication)
  - final subtraction might be necessary
- Examples: $P = 2^{255} - 19$ (Curve25519) or $P = 2^{448} - 2^{224} - 1$ (Ed448-Goldilocks)



MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$

MP modular reduction: general case

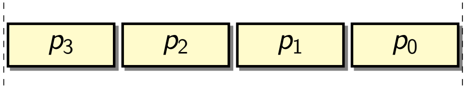
- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!

MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision

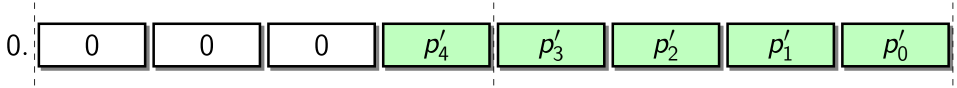
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:



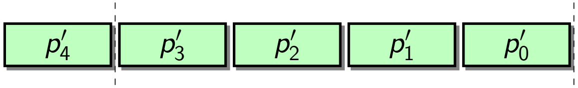
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)



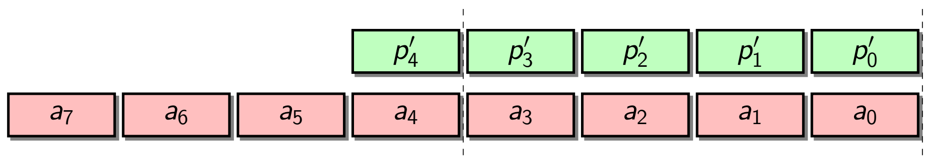
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k + 1$ words)



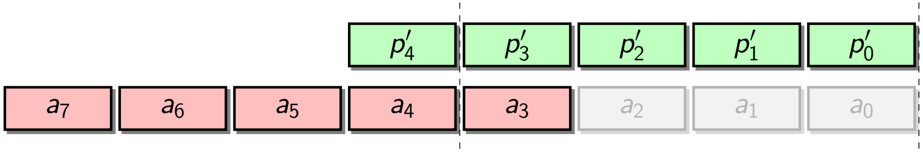
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



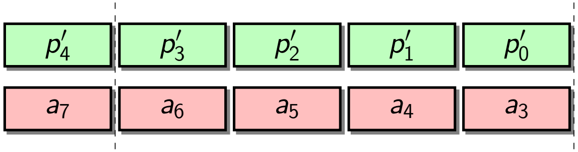
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



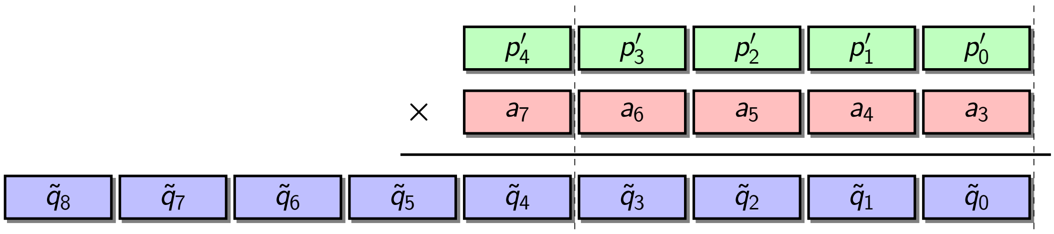
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k + 1$ words)
  - given $A < P^2$, get the $k + 1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$



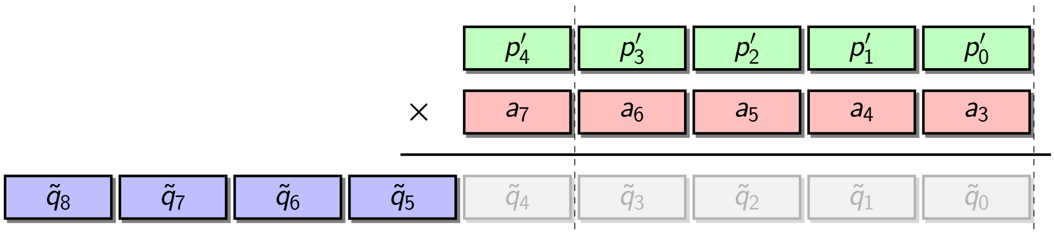
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)



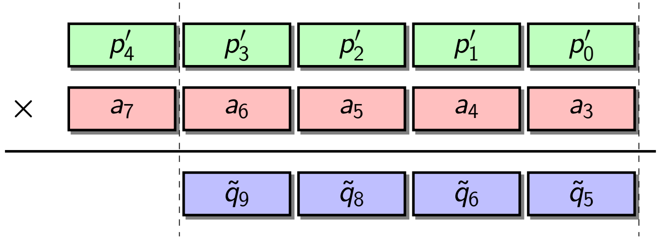
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)



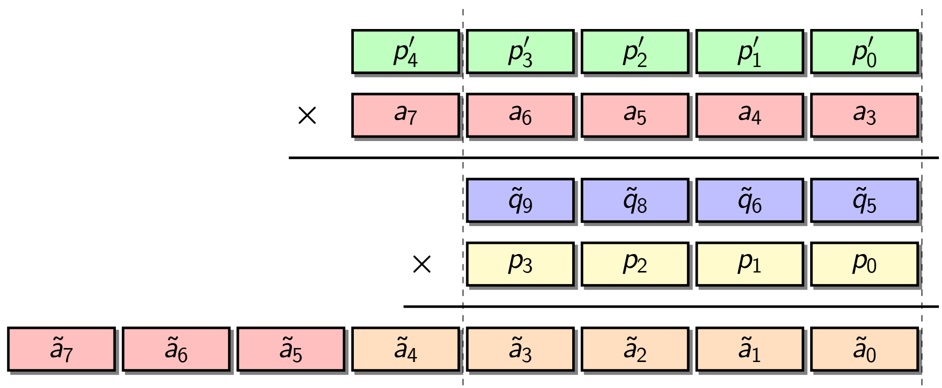
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)



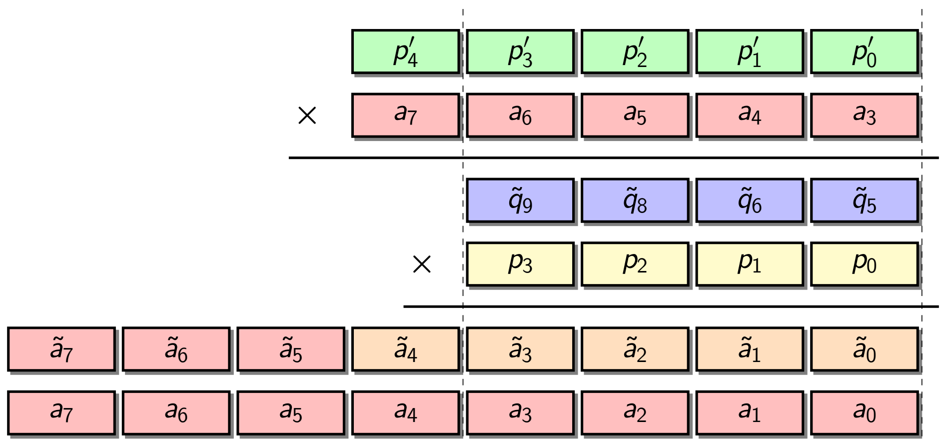
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)



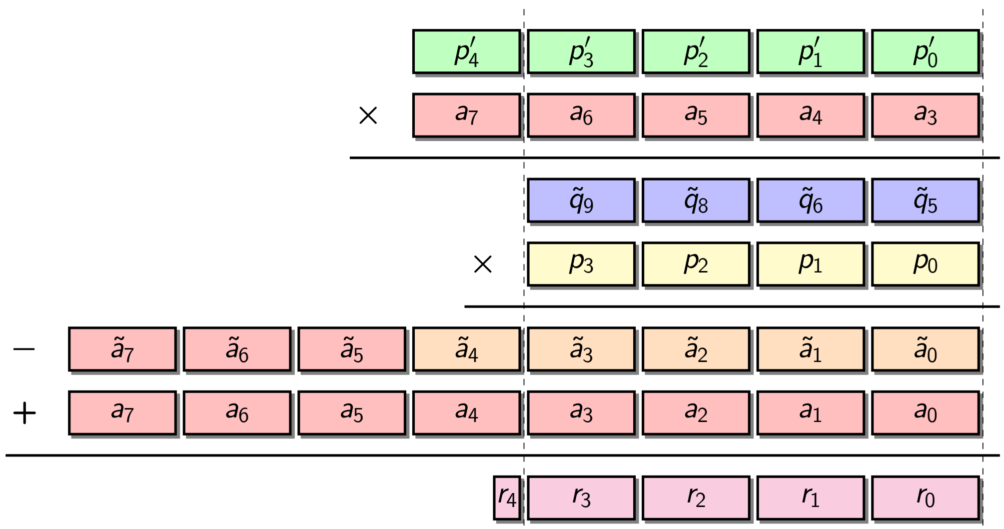
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)



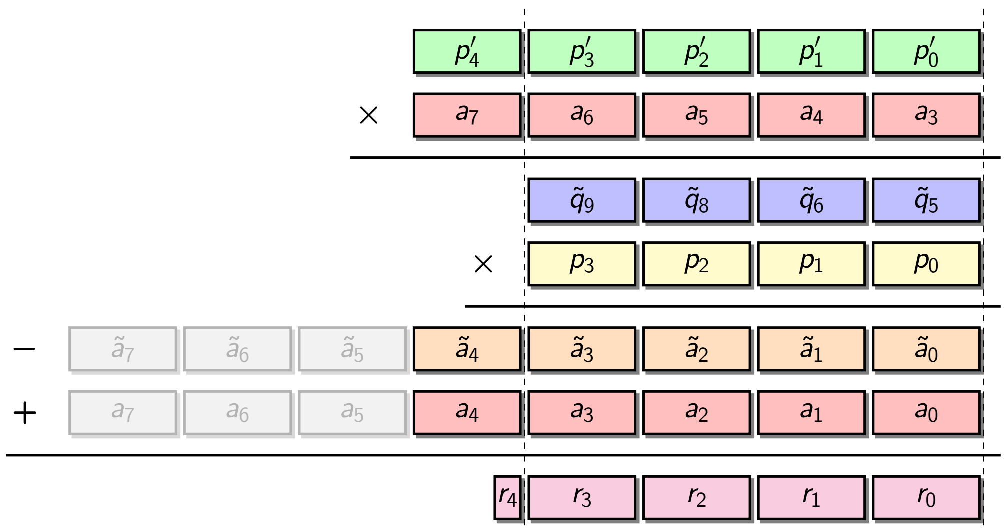
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)
  - compute $\tilde{A} \leftarrow \tilde{Q} \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A - \tilde{A}$



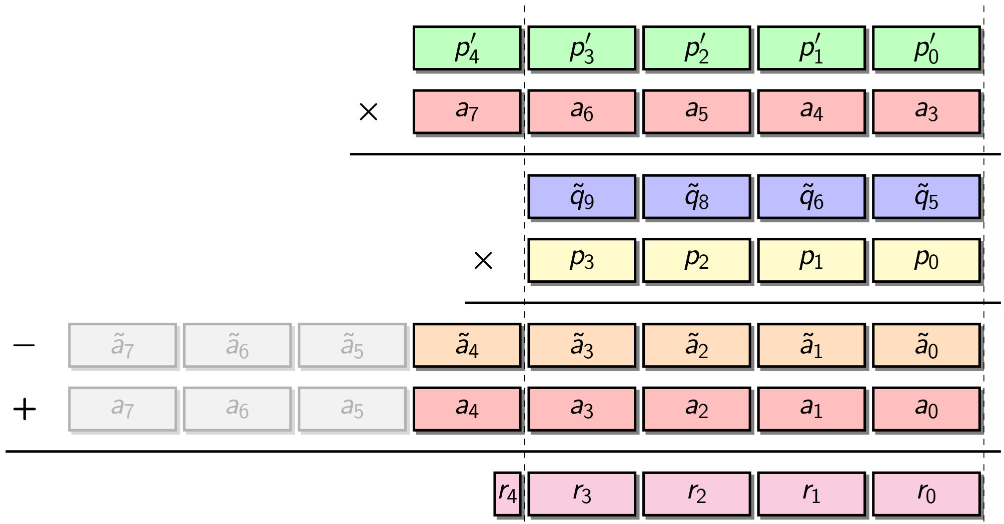
MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)
  - compute $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \bmod 2^{(k+1)w}$ (one $k \times k$-word short multiplication)
  - compute remainder $R \leftarrow (A - \tilde{A}) \bmod 2^{(k+1)w}$



MP modular reduction: general case

- Idea: find quotient $Q = \lfloor A/P \rfloor$, then take remainder as $A - QP$
  - Euclidean division is way too expensive!
  - since P is fixed, precompute $1/P$ with enough precision
- Barrett reduction:
  - precompute $P' = \lfloor 2^{2kw}/P \rfloor$ ($k+1$ words)
  - given $A < P^2$, get the $k+1$ most significant words $A_H \leftarrow \lfloor A/2^{(k-1)w} \rfloor$
  - compute $\tilde{Q} \leftarrow \lfloor A_H \cdot P'/2^{(k+1)w} \rfloor$ (one $(k+1) \times (k+1)$-word multiplication)
  - compute $\tilde{A} \leftarrow (\tilde{Q} \cdot P) \bmod 2^{(k+1)w}$ (one $k \times k$-word short multiplication)
  - compute remainder $R \leftarrow (A - \tilde{A}) \bmod 2^{(k+1)w}$
  - since $Q - 2 \leq \tilde{Q} \leq Q$, at most two final subtractions

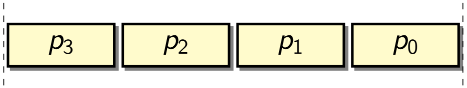


MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words

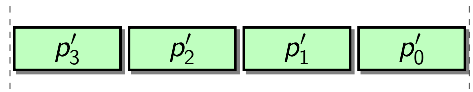
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$



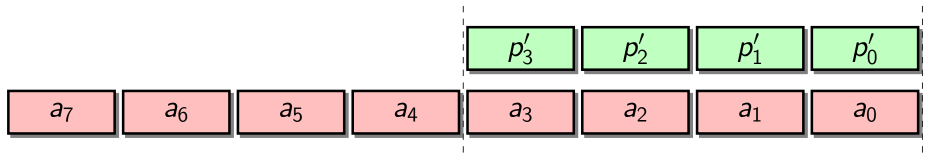
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)



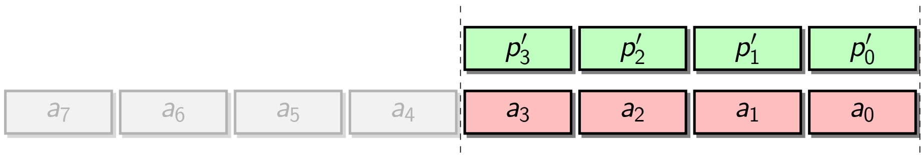
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



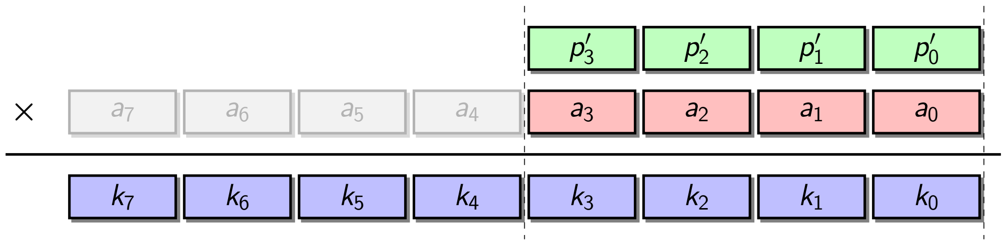
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



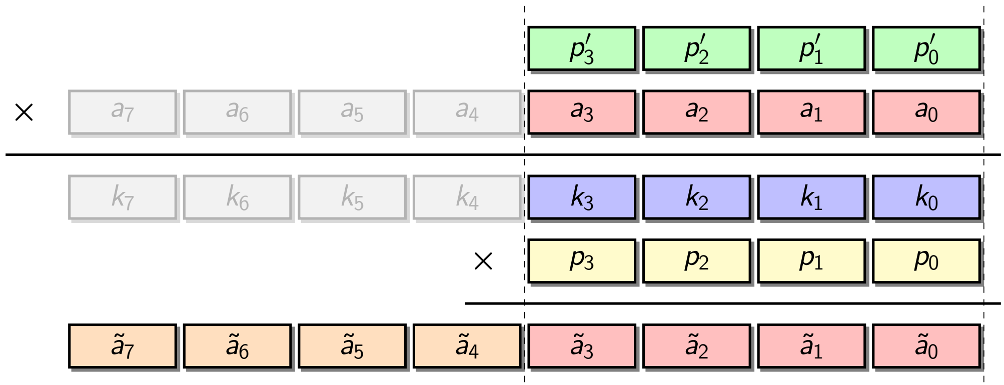
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)



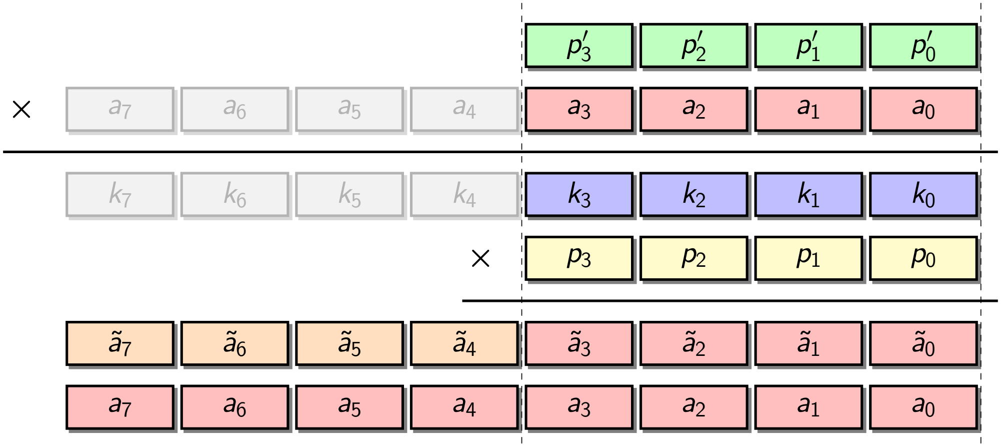
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)



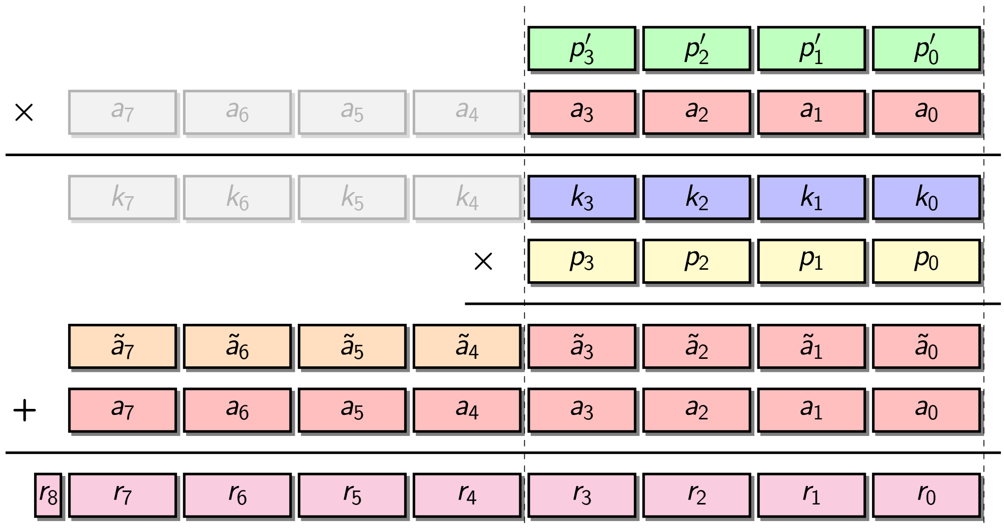
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow \ A + \tilde{A}$



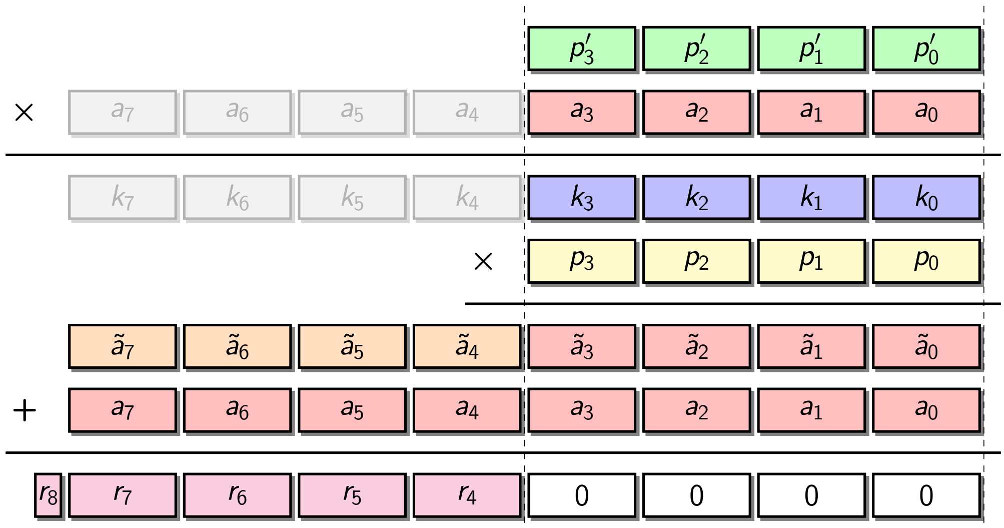
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$



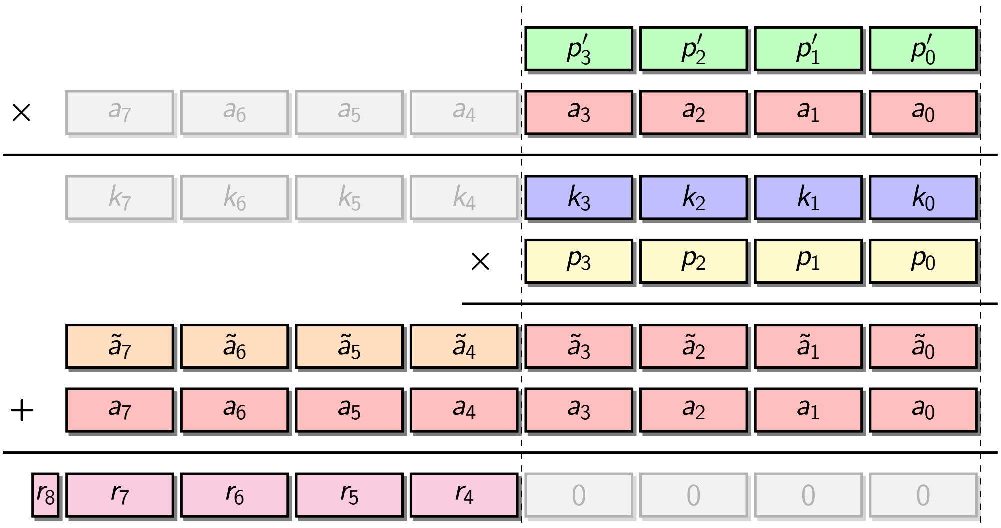
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$



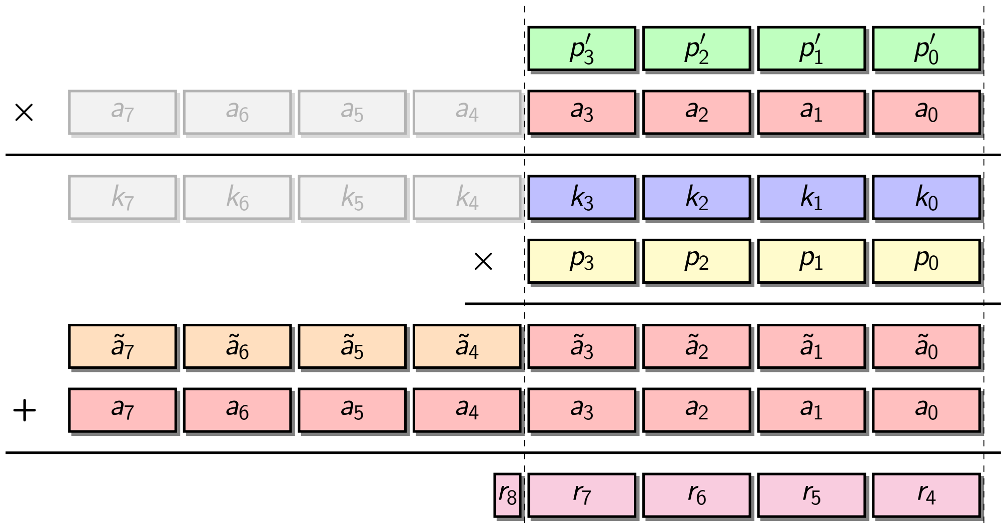
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow A + \tilde{A}$



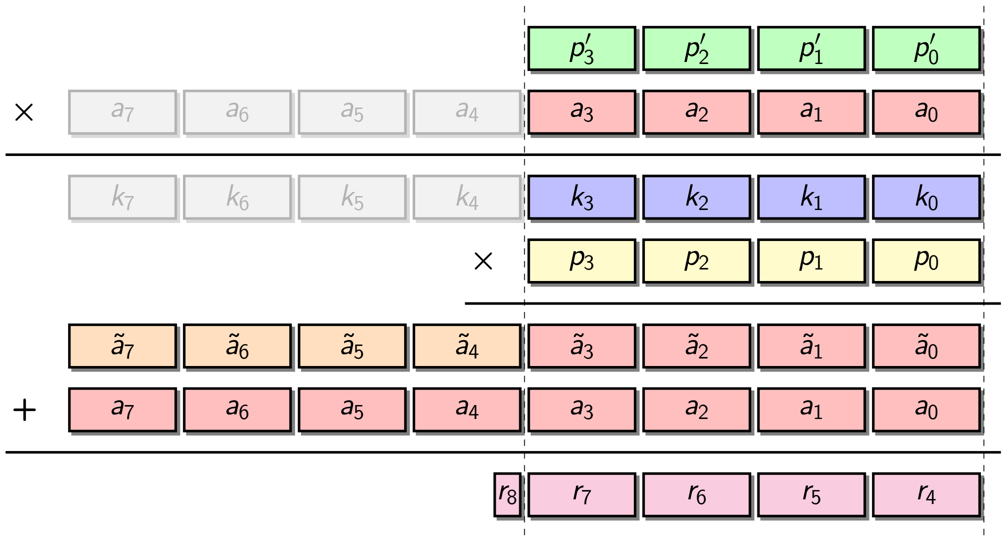
MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$



MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction



MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- REDC(A) returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- REDC(A) returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

$\rightarrow$ that's the so-called Montgomery multiplication

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- REDC(A) returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  → that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \text{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \text{REDC}(\hat{X}, 1)$$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  $\rightarrow$ that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\mathrm{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\mathrm{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

  → that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \mathrm{REDC}(X, 2^{2kw} \bmod P) \qquad \text{and} \qquad X = \mathrm{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

  ⇒ do all computations in Montgomery repr. instead of converting back and forth

MP modular reduction: general case

- Montgomery reduction (REDC): like Barrett, but on the least significant words
  - requires P odd (on k words) and $A < 2^{kw}P$
  - precompute $P' \leftarrow (-P^{-1}) \bmod 2^{kw}$ (on k words)
  - given A, compute $K \leftarrow (A \cdot P') \bmod 2^{kw}$ (one $k \times k$-word short multiplication)
  - compute $\tilde{A} \leftarrow K \cdot P$ (one $k \times k$-word multiplication)
  - compute remainder $R \leftarrow (A + \tilde{A})/2^{kw}$
  - at most one extra subtraction
- $\text{REDC}(A)$ returns $R = (A \cdot 2^{-kw}) \bmod P$, not $A \bmod P$!
  - represent $X \in \mathbb{F}_P$ in Montgomery representation: $\hat{X} = (X \cdot 2^{kw}) \bmod P$
  - if $Z = (X \cdot Y) \bmod P$, then

$$\text{REDC}(\hat{X} \cdot \hat{Y}) = (X \cdot Y \cdot 2^{kw}) \bmod P = \hat{Z}$$

    $\rightarrow$ that's the so-called Montgomery multiplication
  - conversions:

$$\hat{X} = \text{REDC}(X, 2^{2kw} \bmod P) \quad \text{and} \quad X = \text{REDC}(\hat{X}, 1)$$

  - Montgomery representation is compatible with addition / subtraction in $\mathbb{F}_P$

  $\Rightarrow$ do all computations in Montgomery repr. instead of converting back and forth
- REDC can be computed iteratively (one word at a time) and interleaved with the computation of $\hat{X} \cdot \hat{Y}$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

A

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

$$A \xrightarrow{\text{S}} A^2$$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

$$A \xrightarrow{\text{S}} A^2 \xrightarrow{\text{S}} A^4$$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod P$ and $A^{-1} \equiv U \pmod P$
  - fast, but running time depends on A

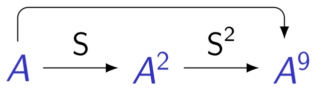
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod P$, whence $A^{P-2} \equiv A^{-1} \pmod P$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]

$$A \xrightarrow{\mathsf{S}} A^2 \xrightarrow{\mathsf{S}^2} A^8$$

MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

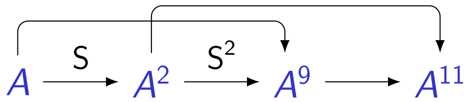
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

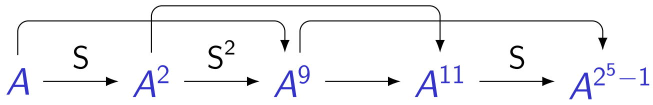
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

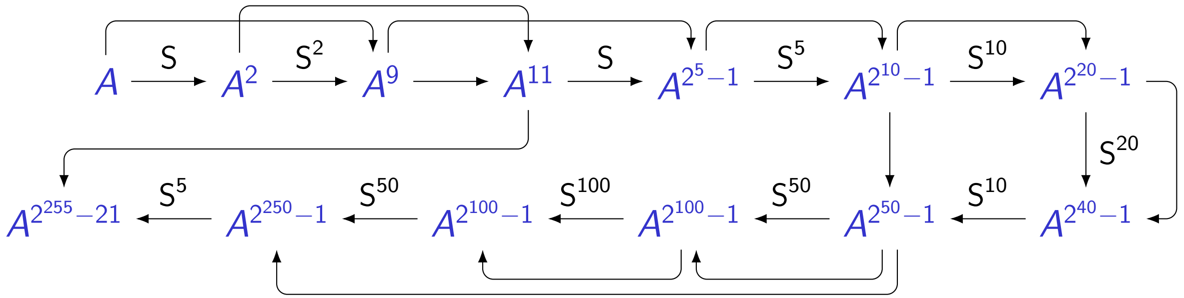
  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



MP field inversion

- Given $A \in \mathbb{F}_P^*$, compute $A^{-1} \bmod P$
- Extended Euclidean algorithm:
  - compute Bézout's coefficients: U and V such that $UA + VP = \gcd(A, P) = 1$
  - then $UA \equiv 1 \pmod{P}$ and $A^{-1} \equiv U \pmod{P}$
  - fast, but running time depends on A

  $\Rightarrow$ requires randomization of A to protect against timing attacks
- Fermat's little theorem:
  - we know that $A^{P-1} \equiv 1 \pmod{P}$, whence $A^{P-2} \equiv A^{-1} \pmod{P}$
  - precompute short sequence of squarings and multiplications for fast exponentiation of A
  - example: $P = 2^{255} - 19$ in 11M and 254S [Bernstein, 2006]



The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    $\rightarrow$ that is the RNS representation of A in base $\mathcal{B}$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:

$$m_i = 2^w - c_i, \text{ with small } c_i$$

  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$

- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    $\rightarrow$ that is the RNS representation of A in base $\mathcal{B}$
  - given $\overrightarrow{A} = (a_1, \ldots, a_k)$, retrieve the unique corresponding integer $A \in \mathbb{Z}/M\mathbb{Z}$ using the Chinese remaindering theorem (CRT):

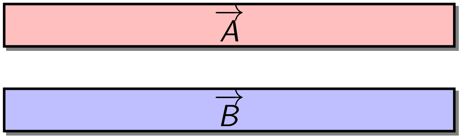
$$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M$$

The Residue Number System (RNS)

- Let $\mathcal{B} = (m_1, \ldots, m_k)$ a tuple of k pairwise coprime integers
  - typically, the m_i's are chosen to fit in a machine word (w bits)
  - pseudo-Mersenne primes allow for easy reduction modulo m_i:
    $$m_i = 2^w - c_i, \text{ with small } c_i$$
  - write $M = \prod_{i=1}^{k} m_i$ and, for all i, $M_i = M/m_i$
- Let $A < M$ be an integer
  - represent A as the tuple $\overrightarrow{A} = (a_1, \ldots, a_k)$ with $a_i = A \bmod m_i = |A|_{m_i}$, for all i
    $\rightarrow$ that is the RNS representation of A in base $\mathcal{B}$
  - given $\overrightarrow{A} = (a_1, \ldots, a_k)$, retrieve the unique corresponding integer $A \in \mathbb{Z}/M\mathbb{Z}$ using the Chinese remaindering theorem (CRT):
    $$A = \left| \sum_{i=1}^{k} |a_i \cdot M_i^{-1}|_{m_i} \cdot M_i \right|_M$$
- If $M > P$, we can represent elements of $\mathbb{F}_P$ in RNS

RNS arithmetic

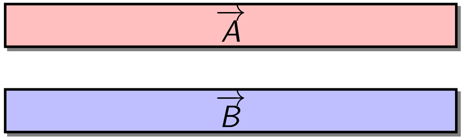
- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \dots, a_k)$ and $\overrightarrow{B} = (b_1, \dots, b_k)$
  - add., sub. and mult. can be performed in parallel on all “channels”:

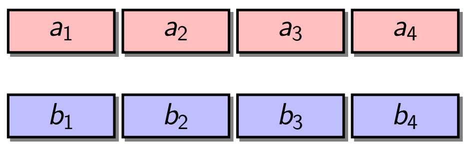
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \dots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \dots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

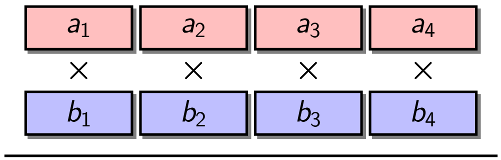
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

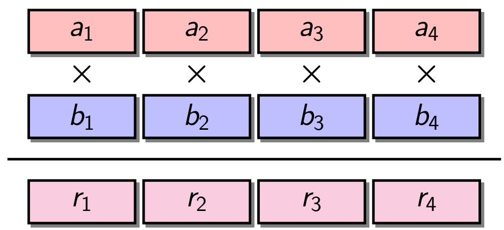
$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$



RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

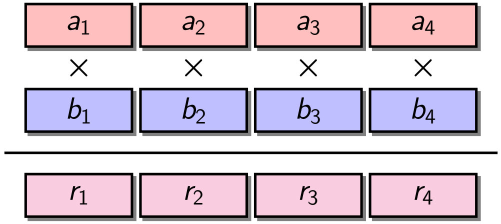


RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation

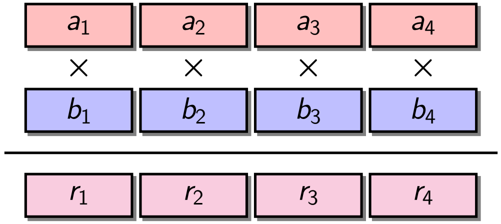


RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation



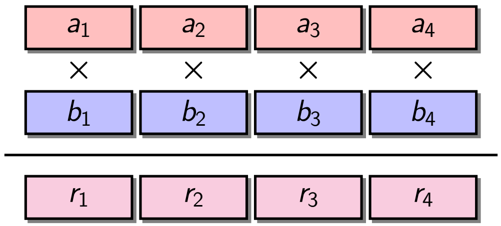
- Limitations:

RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation



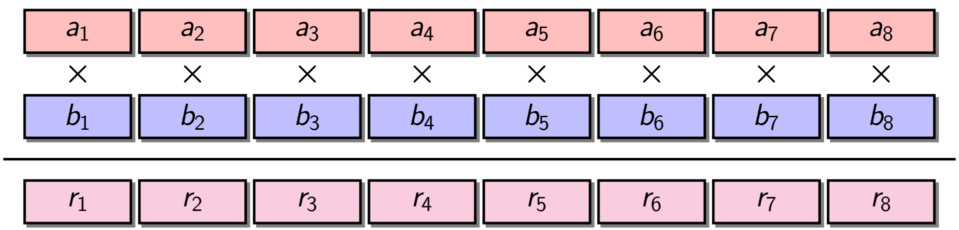
- Limitations:
  - operations are computed in $\mathbb{Z}/M\mathbb{Z}$: beware of overflows! (we need $M > P^2$)

RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation



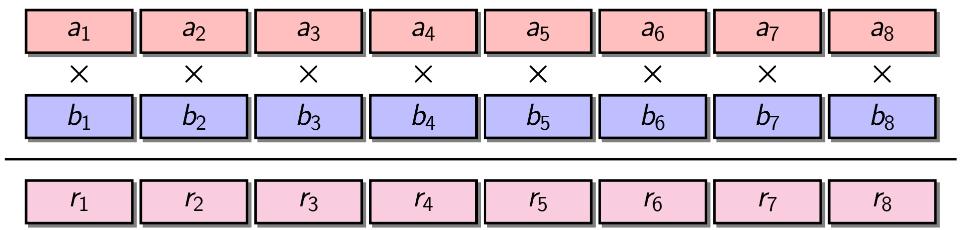
- Limitations:
  - operations are computed in $\mathbb{Z}/M\mathbb{Z}$: beware of overflows! (we need $M > P^2$)

RNS arithmetic

- Let $\overrightarrow{A} = (a_1, \ldots, a_k)$ and $\overrightarrow{B} = (b_1, \ldots, b_k)$
  - add., sub. and mult. can be performed in parallel on all "channels":

$$\overrightarrow{A} \pm \overrightarrow{B} = (|a_1 \pm b_1|_{m_1}, \ldots, |a_k \pm b_k|_{m_k})$$
$$\overrightarrow{A} \times \overrightarrow{B} = (|a_1 \times b_1|_{m_1}, \ldots, |a_k \times b_k|_{m_k})$$

  - native parallelism: suited to SIMD instructions and hardware implementation



- Limitations:
  - operations are computed in $\mathbb{Z}/M\mathbb{Z}$: beware of overflows! (we need $M > P^2$)
  - RNS modular reduction has quadratic complexity $O(k^2)$

RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $M_\alpha > P$, $M_\beta > P$, and $\gcd(M_\alpha, M_\beta) = 1$

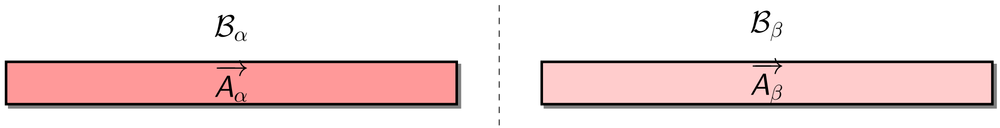
RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $M_\alpha > P$, $M_\beta > P$, and $\gcd(M_\alpha, M_\beta) = 1$
- RNS base extension algorithm (BE) [Kawamura *et al.*, 2000]
  - given $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$, $\mathrm{BE}(\overrightarrow{X_\alpha}, \mathcal{B}_\alpha, \mathcal{B}_\beta)$ computes $\overrightarrow{X_\beta}$, the repr. of X in base $\mathcal{B}_\beta$
  - similarly, $\mathrm{BE}(\overrightarrow{X_\beta}, \mathcal{B}_\beta, \mathcal{B}_\alpha)$ computes $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$

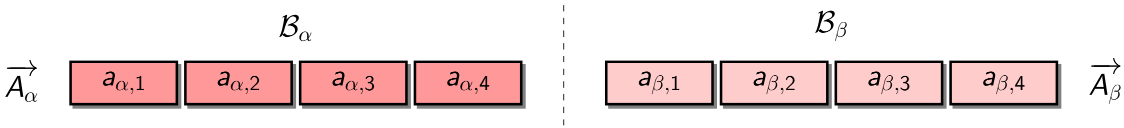
RNS Montgomery reduction

- Requires two RNS bases $\mathcal{B}_\alpha = (m_{\alpha,1}, \ldots, m_{\alpha,k})$ and $\mathcal{B}_\beta = (m_{\beta,1}, \ldots, m_{\beta,k})$ such that $M_\alpha > P$, $M_\beta > P$, and $\gcd(M_\alpha, M_\beta) = 1$
- RNS base extension algorithm (BE) [Kawamura *et al.*, 2000]
  - given $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$, $\mathrm{BE}(\overrightarrow{X_\alpha}, \mathcal{B}_\alpha, \mathcal{B}_\beta)$ computes $\overrightarrow{X_\beta}$, the repr. of X in base $\mathcal{B}_\beta$
  - similarly, $\mathrm{BE}(\overrightarrow{X_\beta}, \mathcal{B}_\beta, \mathcal{B}_\alpha)$ computes $\overrightarrow{X_\alpha}$ in base $\mathcal{B}_\alpha$
  - similar to RNS modular reduction $\rightarrow$ $O(k^2)$ complexity

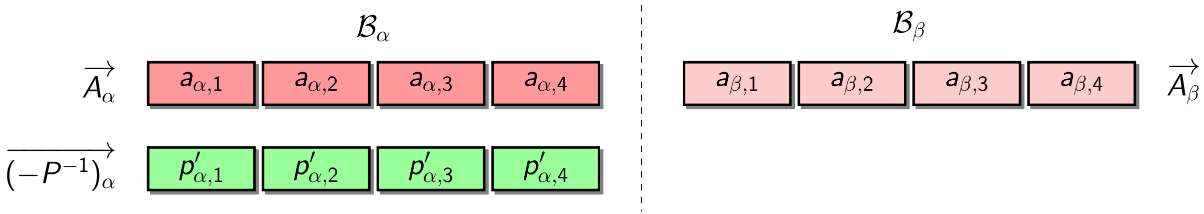
RNS Montgomery reduction



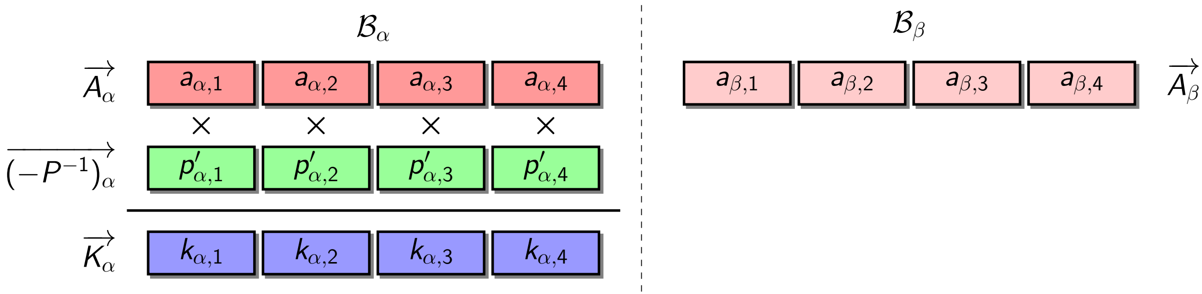
RNS Montgomery reduction



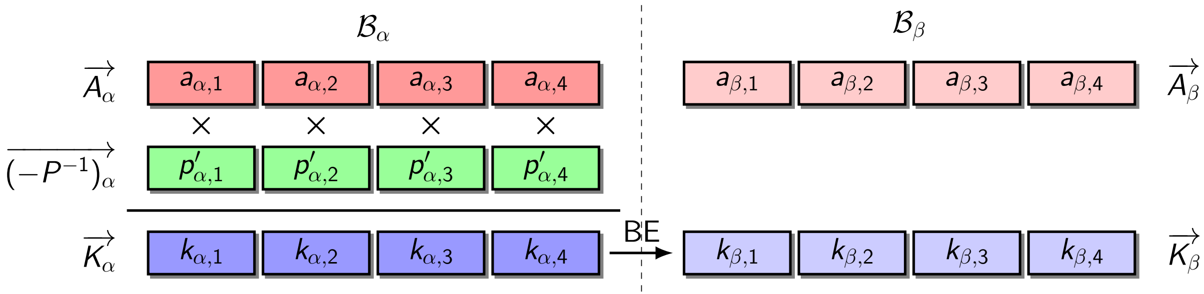
RNS Montgomery reduction



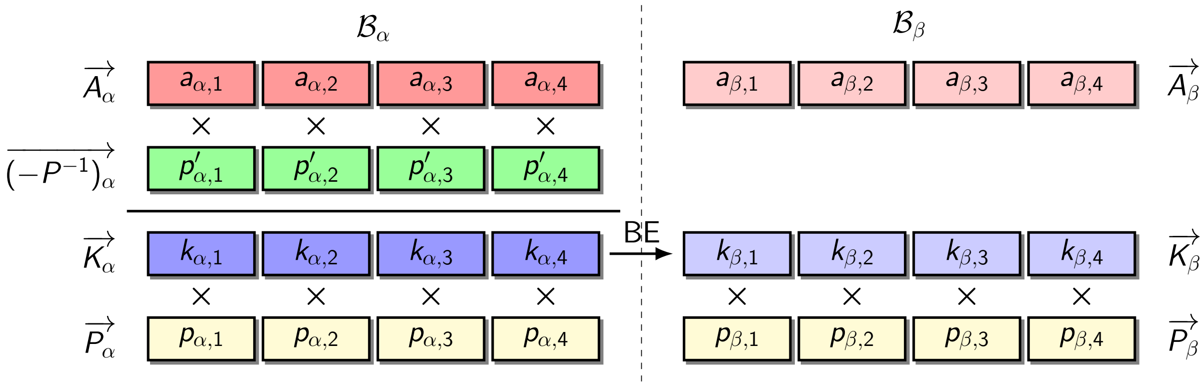
RNS Montgomery reduction



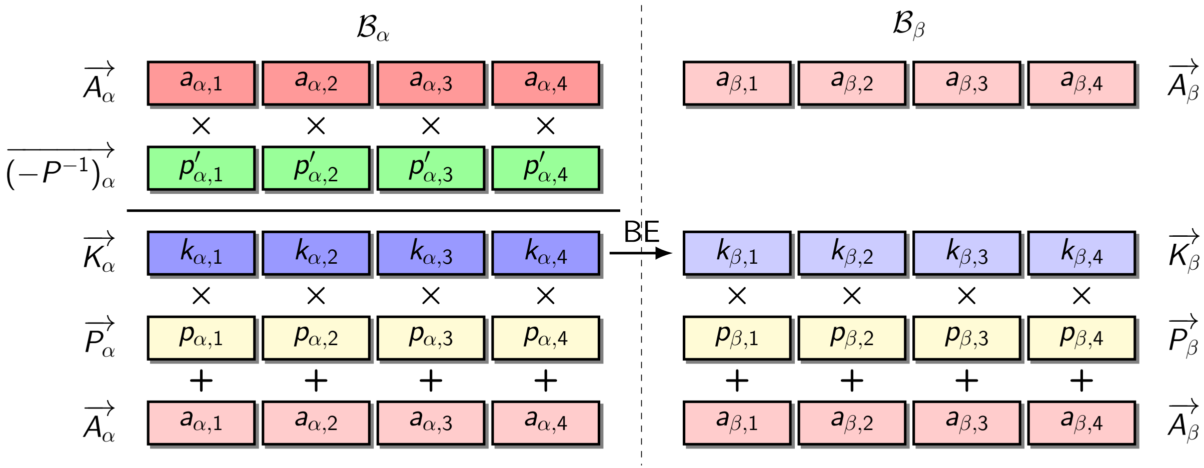
RNS Montgomery reduction



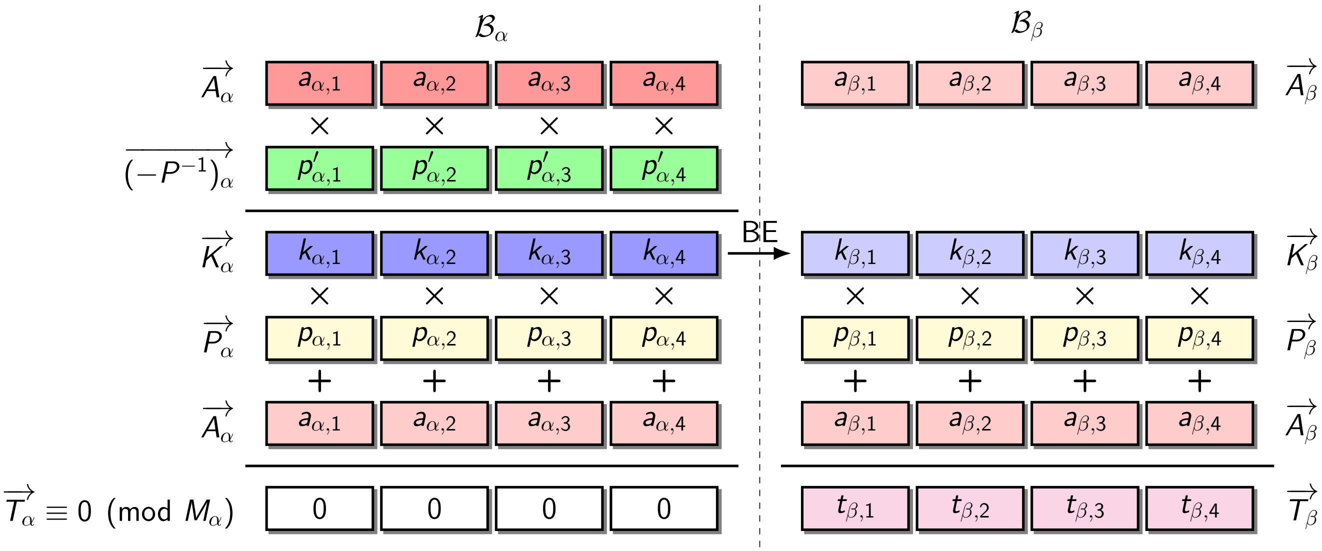
RNS Montgomery reduction



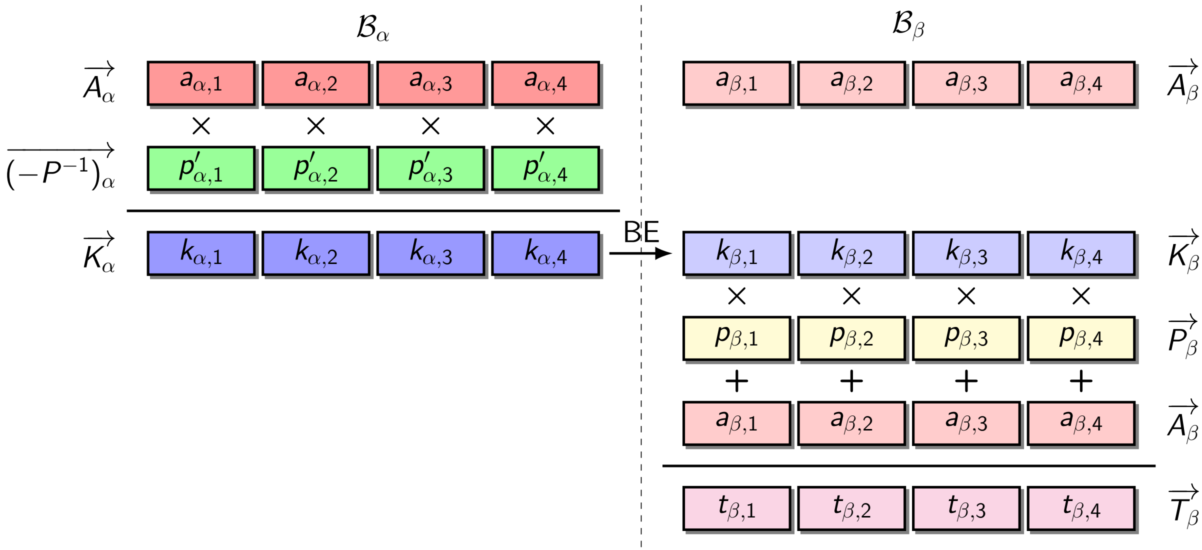
RNS Montgomery reduction



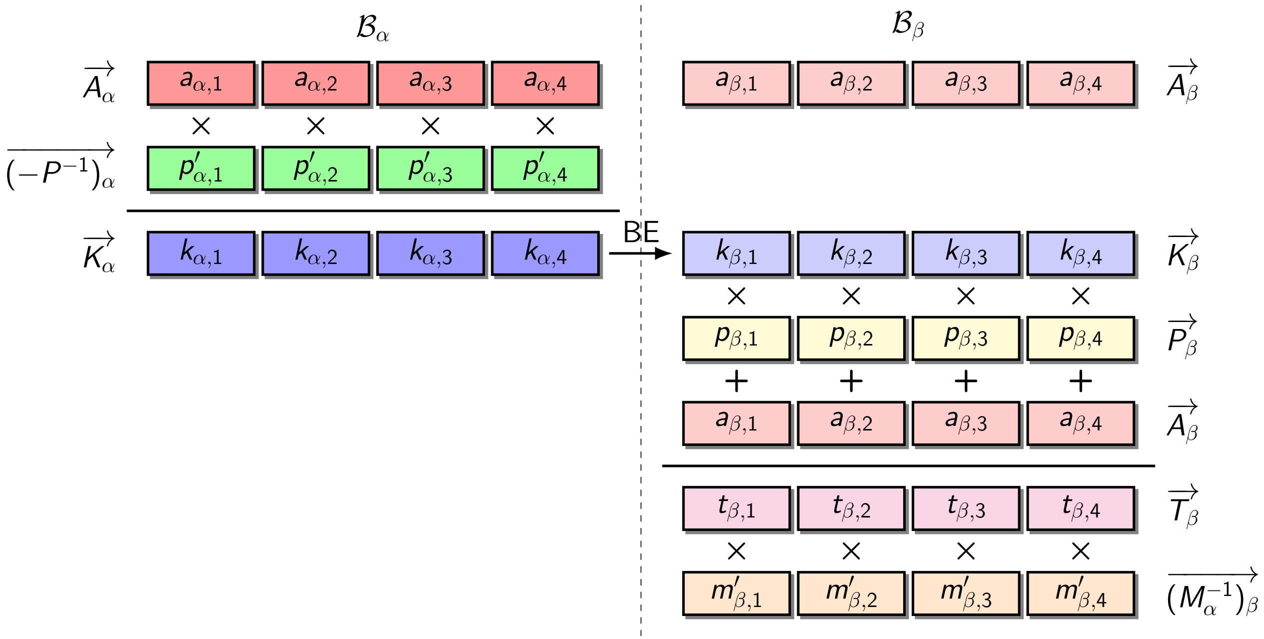
RNS Montgomery reduction



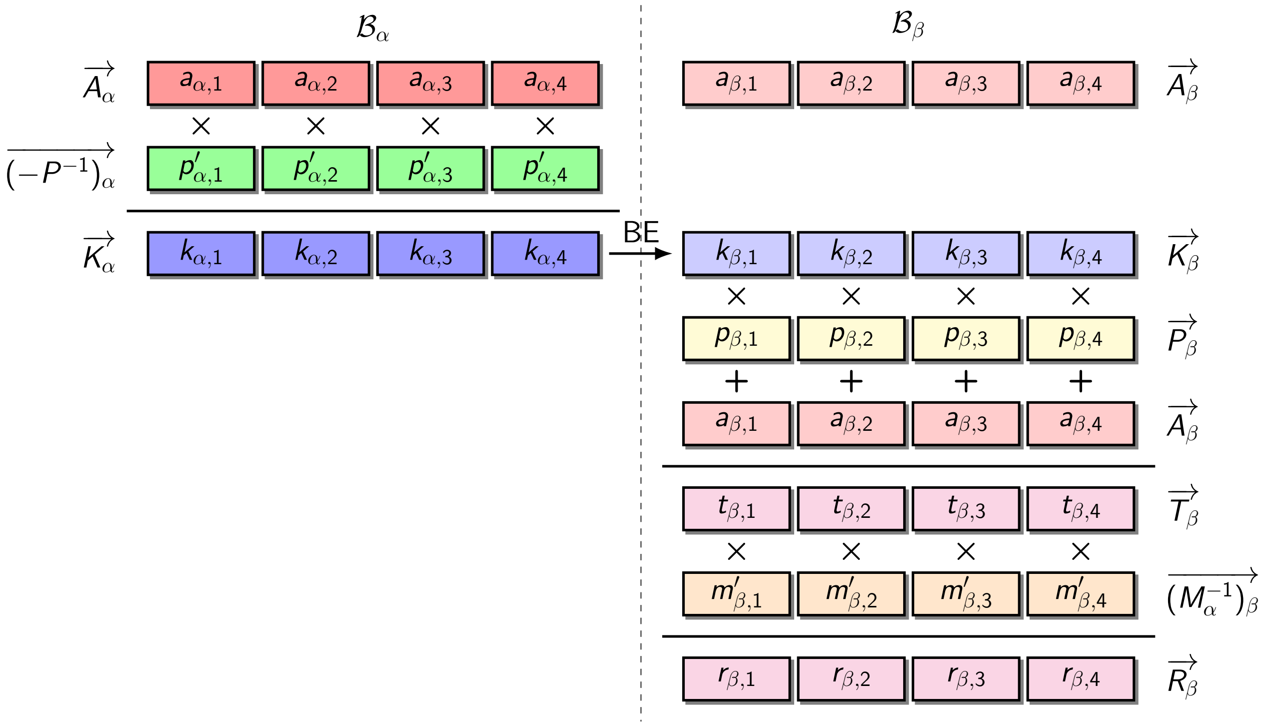
RNS Montgomery reduction



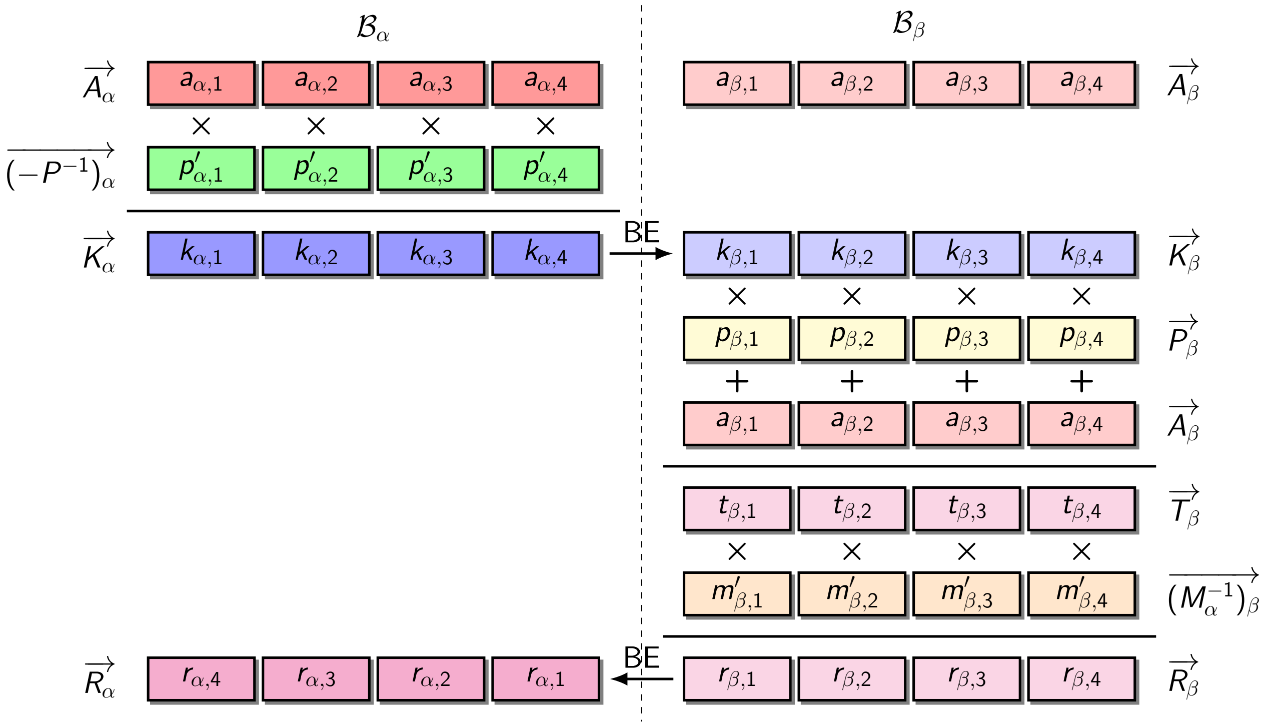
RNS Montgomery reduction



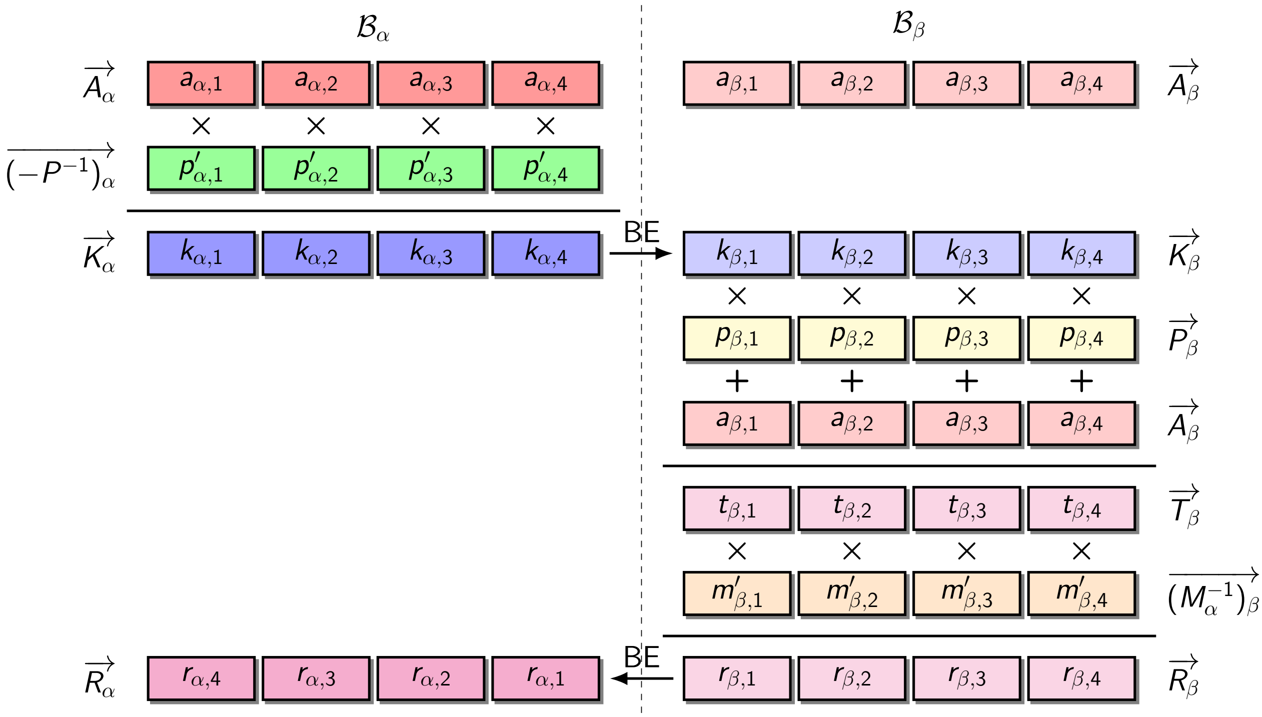
RNS Montgomery reduction



RNS Montgomery reduction

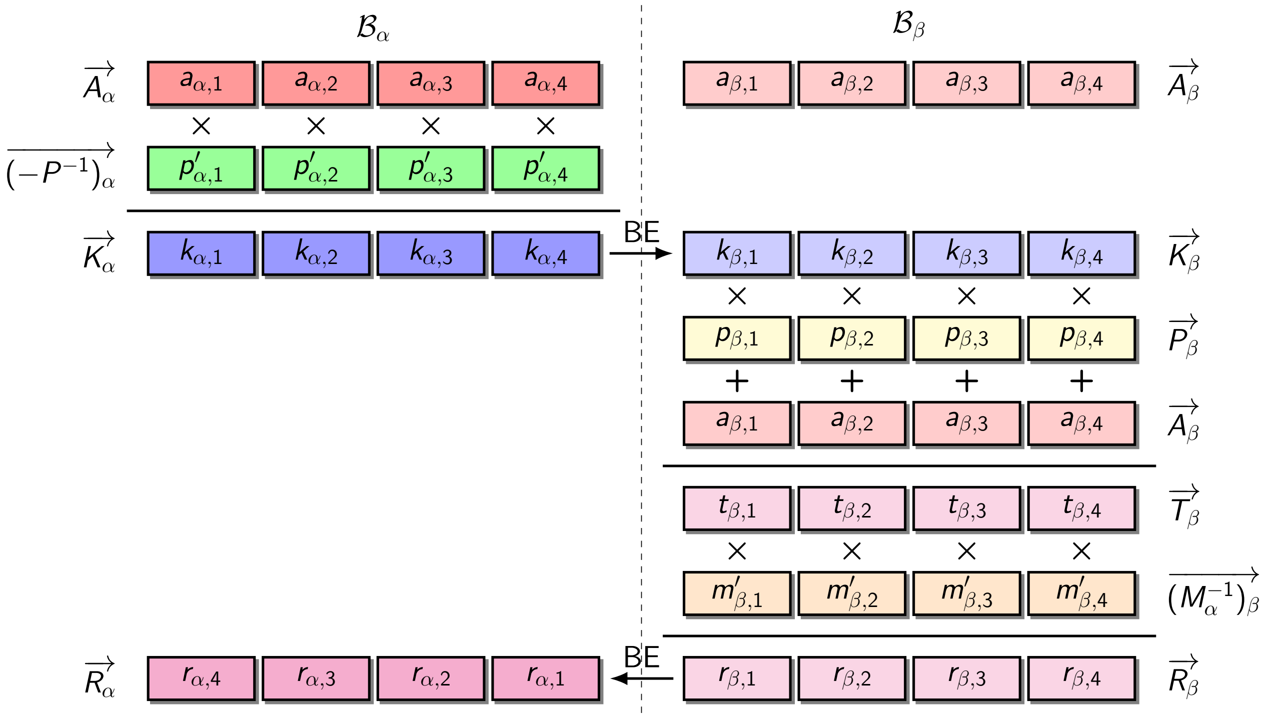


RNS Montgomery reduction



- Result is $(\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M_\alpha^{-1}) \pmod{P}$

RNS Montgomery reduction



- Result is $(\overrightarrow{R_\alpha}, \overrightarrow{R_\beta}) \equiv (A \cdot M_\alpha^{-1}) \pmod{P}$
- See also the hybrid position–residues number system [Bigou & Tisserand, 2016]

Un peu de publicité éhontée...

Journées Codage & Cryptographie 2017

du 23 au 28 avril à La Bresse (Vosges)

Soumission de résumés:	jusqu'au 8 mars
Inscriptions:	jusqu'au 3 avril

https://jc2-2017.inria.fr/

À très bientôt dans les Vosges !