Constructive and destructive implementations of elliptic-curve arithmetic

Peter Schwabe

Research Center for Information Technology Innovation
Academia Sinica

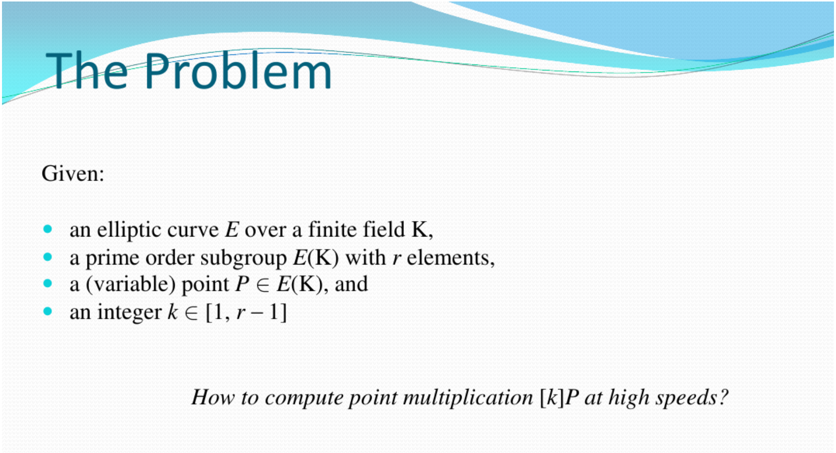
中央研究院

October 30, 2012

ECC 2012, Querétaro, Mexico

Variable-basepoint scalar multiplication

中央研究院



(Part of) Patrick Longa's first slide at ECC 2011
"Elliptic Curve Cryptography at High Speeds"

Answers to this question

中央研究院

- Three recent updates (all for Intel Sandy Bridge):
  - Aranha, Faz-Hernández, López, and Rodríguez-Henríquez: **Faster implementation of scalar multiplication on Koblitz curves**, Latincrypt 2012.
    Result: 99200 cycles on the NIST-K283 curve.
    Code will be available
  - Longa and Sica: **Four-Dimensional Gallant-Lambert-Vanstone Scalar Multiplication**, Asiacrypt 2012.
    Result: 91000 cycles on a 256-bit curve over a prime field.
    Code not available
  - Oliveira, Rodríguez-Henríquez, and López: **New timings for scalar multiplication using a new set of coordinates**, ECC 2012 rump session.
    Result: 75000 cycles on a 254-bit curve over a binary field.
    Code will be available

Outline of this talk

- In all ECC software I wrote I never answered the question "How fast can we do variable-basepoint scalar multiplication?"

Outline of this talk

- In all ECC software I wrote I never answered the question "How fast can we do variable-basepoint scalar multiplication?"
- Maybe I'm not doing my job properly, or maybe it is (often) the wrong question to ask in the first place?
- Certainly there is a lot more to do for ECC software performance

Outline of this talk

中央研究院

- In all ECC software I wrote I never answered the question "How fast can we do variable-basepoint scalar multiplication?"
- Maybe I'm not doing my job properly, or maybe it is (often) the wrong question to ask in the first place?
- Certainly there is a lot more to do for ECC software performance
- Example 1: Elliptic-curve Diffie-Hellman key exchange
- Example 2: Elliptic-curve signatures
- Example 3: Solving the ECDLP with Pollard's rho algorithm

Elliptic-curve Diffie-Hellman key exchange

中央研究院

- Alice and Bob each pick random secret scalar, compute scalar product with a *fixed base point*
- Alice and Bob each receive point from the other one, multiply by their secret scalar

Elliptic-curve Diffie-Hellman key exchange

中央研究院

- Alice and Bob each pick random secret scalar, compute scalar product with a *fixed base point*
- Alice and Bob each receive point from the other one, multiply by their secret scalar
- Second step sounds exactly like variable basepoint scalar multiplication

Elliptic-curve Diffie-Hellman key exchange 中央研究院

- Alice and Bob each pick random secret scalar, compute scalar product with a *fixed base point*
- Alice and Bob each receive point from the other one, multiply by their secret scalar
- Second step sounds exactly like variable basepoint scalar multiplication
- Usual way to make this fast:
  - High level: reduce number of EC additions and doublings
  - Mid level: reduce number of field operations per EC addition and doubling
  - Low level: reduce number of CPU cycles taken by field operations

Sliding-window scalar multiplication

中央研究院

- Choose window size w
- Precompute $P, 3P, 5P, \ldots, (2^w - 1)P$
- Rewrite scalar k as $k = \sum k_i 2^i$ with k_i in $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Double for each coefficient, add for nonzero coefficients
- Expected number of additions: $\approx \text{len}(k)/(w+1) + 2^{w-1}$

Sliding-window scalar multiplication

中央研究院

- Choose window size w
- Precompute $P, 3P, 5P, \dots, (2^w - 1)P$
- Rewrite scalar k as $k = \sum k_i 2^i$ with k_i in $\{0, 1, 3, 5, \dots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Double for each coefficient, add for nonzero coefficients
- Expected number of additions: $\approx \text{len}(k)/(w + 1) + 2^{w-1}$
- Standard optimization: Use signed representation

Sliding-window scalar multiplication

中央研究院

- Choose window size w
- Precompute $P, 3P, 5P, \dots, (2^w - 1)P$
- Rewrite scalar k as $k = \sum k_i 2^i$ with k_i in $\{0, 1, 3, 5, \dots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Double for each coefficient, add for nonzero coefficients
- Expected number of additions: $\approx \text{len}(k)/(w+1) + 2^{w-1}$
- Standard optimization: Use signed representation
- For curves with efficiently computable endomorphism φ:
  - Split scalar k in k_1, k_2, s.t. $kP = k_1 P + k_2 \varphi(P)$
  - Perform double-scalar multiplication with half-size scalars
  - Halves the number of doublings
  - Estimate by Galbraith, Lin, Scott (2009): speedup of 30% to 40%

Problem: timing attacks

- Branch conditions depend on secret data (scalar)
- Code takes different amount of time depending on the scalar
- This is true even if the code in both possible branches takes the same amount of time (reason: branch prediction)
- Attacker can measure time and deduce information about the scalar

Problem: timing attacks

中央研究院

- Branch conditions depend on secret data (scalar)
- Code takes different amount of time depending on the scalar
- This is true even if the code in both possible branches takes the same amount of time (reason: branch prediction)
- Attacker can measure time and deduce information about the scalar
- You don't think this is scary? Wait for Billy Bob Brumley's talk tomorrow.

Fixed-window scalar multiplication

中央研究院

- Choose window size w
- Represent scalar k in base 2^w: $k = \sum k_i 2^{iw}$
- Precompute $0P, 1P, 2P, 3P, \ldots, (2^w - 1)P$
- For each k_i: add $k_i P$ into result; do w point doublings

Fixed-window scalar multiplication

中央研究院

- Choose window size w
- Represent scalar k in base 2^w: $k = \sum k_i 2^{iw}$
- Precompute $0P, 1P, 2P, 3P, \ldots, (2^w - 1)P$
- For each k_i: add $k_i P$ into result; do w point doublings
- Standard optimization: Use signed representation

Fixed-window scalar multiplication

中央研究院

- Choose window size w
- Represent scalar k in base 2^w: $k = \sum k_i 2^{iw}$
- Precompute $0P, 1P, 2P, 3P, \ldots, (2^w - 1)P$
- For each k_i: add $k_i P$ into result; do w point doublings
- Standard optimization: Use signed representation
- Number of additions: $\lceil \text{len}(k)/w \rceil + 2^w$
- Penalty from more additions is relatively more serious for curves with endomorphisms

Fixed-window scalar multiplication

中央研究院

- Choose window size w
- Represent scalar k in base 2^w: $k = \sum k_i 2^{iw}$
- Precompute $0P, 1P, 2P, 3P, \ldots, (2^w - 1)P$
- For each k_i: add $k_i P$ into result; do w point doublings
- Standard optimization: Use signed representation
- Number of additions: $\lceil \text{len}(k)/w \rceil + 2^w$
- Penalty from more additions is relatively more serious for curves with endomorphisms
- **Dragons ahead!**
  - Requires constant-time EC addition, e.g., use complete Edwards addition formulas
  - Requires constant-time lookups of precomputed points (more later)
  - Requires constant-time finite-field arithmetic

Montgomery Ladder

中央研究院

- Use Montgomery curve $By^2 = x^3 + Ax^2 + x$
- Given the x-coordinate of P, compute the x-coordinate of kP
- For each bit of the scalar k perform a "ladder step":
  - From (x_{Q-P}, x_P, x_Q) compute $(x_{Q-P}, x_{2P}, x_{P+Q})$ (one addition, one doubling)
  - If the current bit is different from the next bit: swap x_{2P} and x_{P+Q}

Montgomery Ladder

中央研究院

- Use Montgomery curve $By^2 = x^3 + Ax^2 + x$
- Given the x-coordinate of P, compute the x-coordinate of kP
- For each bit of the scalar k perform a "ladder step":
  - From (x_{Q-P}, x_P, x_Q) compute $(x_{Q-P}, x_{2P}, x_{P+Q})$ (one addition, one doubling)
  - If the current bit is different from the next bit: swap x_{2P} and x_{P+Q}
- Advantage: Very regular structure, no table lookups
- Advantage: Point compression for free

Montgomery Ladder

中央研究院

- Use Montgomery curve $By^2 = x^3 + Ax^2 + x$
- Given the x-coordinate of P, compute the x-coordinate of kP
- For each bit of the scalar k perform a "ladder step":
  - From (x_{Q-P}, x_P, x_Q) compute $(x_{Q-P}, x_{2P}, x_{P+Q})$ (one addition, one doubling)
  - If the current bit is different from the next bit: swap x_{2P} and x_{P+Q}
- Advantage: Very regular structure, no table lookups
- Advantage: Point compression for free
- **Dragons ahead!**
  - Requires constant-time conditional swap
  - Requires constant-time finite-field arithmetic

Constant-time field arithmetic

中央研究院

- Typical operation for reduction: **If** $a \geq p$ **then** $a \leftarrow (a - p)$
- Same problem as before if a depends on secret data

Constant-time field arithmetic

中央研究院

- Typical operation for reduction: **If** $a \geq p$ **then** $a \leftarrow (a - p)$
- Same problem as before if a depends on secret data
- One way around this: Always subtract p:

  $b \leftarrow (a \geq p)$
  $t \leftarrow (a - p)$
  $a \leftarrow b \cdot t + (1 - b) \cdot a$

Constant-time field arithmetic

中央研究院

- Typical operation for reduction: **If** $a \geq p$ **then** $a \leftarrow (a - p)$
- Same problem as before if a depends on secret data
- One way around this: Always subtract p:

  $b \leftarrow (a \geq p)$
  $t \leftarrow (a - p)$
  $a \leftarrow b \cdot t + (1 - b) \cdot a$
- Better way around this: Never subtract p:
  - Choose a representation that leaves room for values $\geq p$
  - For example: 5 64-bit registers, radix 2^{51} to represent elements of $\mathbb{F}_{2^{255}-19}$
  - Another advantage of such a redundant representation: fewer carries

Constant-time field arithmetic

中央研究院

- Typical operation for reduction: **If** $a \geq p$ **then** $a \leftarrow (a - p)$
- Same problem as before if a depends on secret data
- One way around this: Always subtract p:

  $b \leftarrow (a \geq p)$
  $t \leftarrow (a - p)$
  $a \leftarrow b \cdot t + (1 - b) \cdot a$
- Better way around this: Never subtract p:
  - Choose a representation that leaves room for values $\geq p$
  - For example: 5 64-bit registers, radix 2^{51} to represent elements of $\mathbb{F}_{2^{255}-19}$
  - Another advantage of such a redundant representation: fewer carries
- Optimal choice of representation highly depends on the field and the target microarchitecture
- Very often redundant-representation software outperforms non-redundant software (and is constant time!)

Some recent results, Intel processors

Performance on Nehalem/Westmere

- Bernstein, Duif, Lange, Schwabe, Yang (2011): 227348 cycles, no endomorphisms, including point compression.
  Included as `crypto_scalarmult/curve25519/amd64-51/` in SUPERCOP, http://bench.cr.yp.to/supercop.html

Performance on Sandy Bridge

- Hamburg (2012): 153000 cycles, no endomorphisms, including point compression. Code not available.
- Longa, Sica (2012): 137000 cycles (or is it 145000?), endomorphisms, not including point compression. Code not available.

Some recent results, Intel processors

中央研究院

Performance on Sandy Bridge

- Hamburg (2012): 153000 cycles, no endomorphisms, including point compression. Code not available.
- Longa, Sica (2012): 137000 cycles (or is it 145000?), endomorphisms, not including point compression. Code not available.

Performance on Ivy Bridge

- Bos, Costello, Hisil, Lauter (2012): $\ll 140000$ cycles, genus 2, no endomorphisms, some compression. Code will be available in 13 days.

Some recent results, Intel processors

中央研究院

Performance on Sandy Bridge

- Hamburg (2012): 153000 cycles, no endomorphisms, including point compression. Code not available.
- Longa, Sica (2012): 137000 cycles (or is it 145000?), endomorphisms, not including point compression. Code not available.
- Schwabe (2012): 567000 cycles for 4 independent scalar multiplications (141750 cycles per scalar multiplication), no endomorphisms, including point compression. Code online soon at http://cryptojedi.org/crypto/#curve25519avx.

Performance on Ivy Bridge

- Bos, Costello, Hisil, Lauter (2012): $\ll 140000$ cycles, genus 2, no endomorphisms, some compression. Code will be available in 13 days.

Some recent results, Intel processors

中央研究院

Performance on Sandy Bridge

- Hamburg (2012): 153000 cycles, no endomorphisms, including point compression. Code not available.
- Longa, Sica (2012): 137000 cycles (or is it 145000?), endomorphisms, not including point compression. Code not available.
- Schwabe (2012): 567000 cycles for 4 independent scalar multiplications ($\ll$ 142000 cycles per scalar multiplication), no endomorphisms, including point compression. Code online soon at http://cryptojedi.org/crypto/#curve25519avx.

Performance on Ivy Bridge

- Bos, Costello, Hisil, Lauter (2012): $\ll$ 140000 cycles, genus 2, no endomorphisms, some compression. Code will be available in 13 days.

Some recent results, ARM processors

中央研究院

Performance on ARM Cortex A8

- Bernstein, Schwabe (2012): 460200 cycles, no endomorphisms, including point compression.
  Included as `crypto_scalarmult/curve25519/neon2/` in SUPERCOP, http://bench.cr.yp.to/supercop.html

Performance on ARM Cortex A9

- Bernstein, Schwabe (2012): 577997 cycles, no endomorphisms, including point compression. Same code as above.
- Hamburg (2012): 619000 cycles, no endomorphisms, including point compression. Code not available.

Performance on Qualcomm Snapdragon S3

- Bernstein, Schwabe (2012): 425582 cycles, no endomorphisms, including point compression. Same code as above.

Ed25519 elliptic-curve signatures

中央研究院

- Joint work with Bernstein, Duif, Lange, and Yang
- Signature scheme (variant of Schnorr signatures) based on arithmetic on twisted Edwards curve $\mathbb{F}_{2^{255}-19}$
- Curve is birationally equivalent to the Montgomery curve used in Curve25519
- B is a fixed base point on the curve
- ℓ is a 253-bit prime, s.t. $\ell B = (0, 1)$
- ECC secret key: random scalar a
- Public key: 32-byte encoding $\underline{A}$ of $A = aB$ (y and one bit of x)

EdDSA signing

中央研究院

- Compute $R = rB$ for pseudorandom per-message secret r
- Define $S = (r + \text{SHA-512}(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature on message M: $(\underline{R}, \underline{S})$, with $\underline{S}$ the 256-bit little-endian encoding of S

EdDSA signing

中央研究院

- Compute $R = rB$ for pseudorandom per-message secret r
- Define $S = (r + \text{SHA-512}(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature on message M: $(\underline{R}, \underline{S})$, with $\underline{S}$ the 256-bit little-endian encoding of S
- Main operation: Compute rB:
  - First compute $r \mod \ell$, write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, with
    $$r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$$
  - Precompute $16^i|r_i|B$ for $i = 0, \ldots, 63$ and $|r_i| \in \{1, \ldots, 8\}$, in a lookup table at compile time

EdDSA signing

中央研究院

- Compute $R = rB$ for pseudorandom per-message secret r
- Define $S = (r + \text{SHA-512}(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature on message M: $(\underline{R}, \underline{S})$, with $\underline{S}$ the 256-bit little-endian encoding of S
- Main operation: Compute rB:
  - First compute $r \mod \ell$, write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, with

    $$r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$$

  - Precompute $16^i|r_i|B$ for $i = 0, \ldots, 63$ and $|r_i| \in \{1, \ldots, 8\}$, in a lookup table at compile time
  - Compute

    $$R = \sum_{i=0}^{63} 16^i r_i B$$

EdDSA signing

中央研究院

- Compute $R = rB$ for pseudorandom per-message secret r
- Define $S = (r + \text{SHA-512}(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature on message M: $(\underline{R}, \underline{S})$, with $\underline{S}$ the 256-bit little-endian encoding of S
- Main operation: Compute rB:
  - First compute $r \mod \ell$, write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, with

$$r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$$

  - Precompute $16^i|r_i|B$ for $i = 0, \ldots, 63$ and $|r_i| \in \{1, \ldots, 8\}$, in a lookup table at compile time
  - Compute

$$R = \sum_{i=0}^{63} 16^i r_i B$$

  - 64 table lookups, 64 conditional point negations, 63 point additions

EdDSA signing

中央研究院

- Compute $R = rB$ for pseudorandom per-message secret r
- Define $S = (r + \text{SHA-512}(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature on message M: $(\underline{R}, \underline{S})$, with $\underline{S}$ the 256-bit little-endian encoding of S
- Main operation: Compute rB:
  - First compute $r \mod \ell$, write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, with

$$r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$$

  - Precompute $16^i|r_i|B$ for $i = 0, \dots, 63$ and $|r_i| \in \{1, \dots, 8\}$, in a lookup table at compile time
  - Compute

$$R = \sum_{i=0}^{63} 16^i r_i B$$

  - 64 table lookups, 64 conditional point negations, 63 point additions
  - R is represented in extended coordinates (X, Y, Z, T) (Hisil, Wong, Carter, Dawson, 2008)
  - Table entries (x, y) are stored as $(y - x, y + x, 2dxy)$

Timing attacks strike again

中央研究院

- Lookup addresses depend on secret scalar
- Lookups are fast if data is in cache, slow otherwise
- Attacker measures time, deduces information about the key

Timing attacks strike again

中央研究院

- Lookup addresses depend on secret scalar
- Lookups are fast if data is in cache, slow otherwise
- Attacker measures time, deduces information about the key
- Example for a cache-timing attack: In 2006 Osvik, Shamir, and Tromer showed how to steal the 256-bit AES key of the Linux `dmcrypt` harddisk encryption in just 65 ms.

Timing attacks strike again

中央研究院

- Lookup addresses depend on secret scalar
- Lookups are fast if data is in cache, slow otherwise
- Attacker measures time, deduces information about the key
- Example for a cache-timing attack: In 2006 Osvik, Shamir, and Tromer showed how to steal the 256-bit AES key of the Linux `dmcrypt` harddisk encryption in just 65 ms.
- Countermeasure used in Ed25519: Always load all 8 table entries, use arithmetic to choose the right one, e.g. at position r_0:

$$D \leftarrow (1,1,0)$$
$$b \leftarrow (|r_0| = 1)$$
$$D \leftarrow b \cdot \mathsf{Table}[1] + (1-b) \cdot D$$
$$b \leftarrow (|r_0| = 2)$$
$$D \leftarrow b \cdot \mathsf{Table}[2] + (1-b) \cdot D$$
$$\ldots$$

Timing attacks strike again

中央研究院

- Lookup addresses depend on secret scalar
- Lookups are fast if data is in cache, slow otherwise
- Attacker measures time, deduces information about the key
- Example for a cache-timing attack: In 2006 Osvik, Shamir, and Tromer showed how to steal the 256-bit AES key of the Linux `dmcrypt` harddisk encryption in just 65 ms.
- Countermeasure used in Ed25519: Always load all 8 table entries, use arithmetic to choose the right one, e.g. at position r_0:

$$D \leftarrow (1,1,0)$$
$$b \leftarrow (|r_0| = 1)$$
$$D \leftarrow b \cdot \text{Table}[1] + (1-b) \cdot D$$
$$b \leftarrow (|r_0| = 2)$$
$$D \leftarrow b \cdot \text{Table}[2] + (1-b) \cdot D$$
$$\ldots$$

- Always compute negation, use arithmetic to choose D or $-D$

EdDSA verification

中央研究院

- Verify signature $(\underline{R}, \underline{S})$ on message M with public key $\underline{A}$
- Check equation

$$SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A = R$$

EdDSA verification

中央研究院

- Verify signature $(\underline{R}, \underline{S})$ on message M with public key $\underline{A}$
- Check equation

$$SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A = R$$

- Actually: Compare encoding of $SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A$ with $\underline{R}$

EdDSA verification

中央研究院

- Verify signature $(\underline{R}, \underline{S})$ on message M with public key $\underline{A}$
- Check equation

$$SB - \text{SHA-}512(\underline{R}, \underline{A}, M)A = R$$

- Actually: Compare encoding of $SB - \text{SHA-}512(\underline{R}, \underline{A}, M)A$ with $\underline{R}$
- Two main parts:
  - Decompression of A
  - Computation of $SB - \text{SHA-}512(\underline{R}, \underline{A}, M)A$

EdDSA verification

- Verify signature $(\underline{R}, \underline{S})$ on message M with public key $\underline{A}$
- Check equation

$$SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A = R$$

- Actually: Compare encoding of $SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A$ with $\underline{R}$
- Two main parts:
  - Decompression of A
  - Computation of $SB - \text{SHA-512}(\underline{R}, \underline{A}, M)A$
- For second part do the following:
  - Double-scalar multiplication using signed sliding windows
  - Different window sizes for B (compile time) and A (run time)

Point decompression

中央研究院

- Before double-scalar multiplication: compute x coordinate x_A of A as

$$x_A = \pm\sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$$

- Looks like a square root and an inversion is required

Point decompression

中央研究院

- Before double-scalar multiplication: compute x coordinate x_A of A as
$$x_A = \pm\sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$$
- Looks like a square root and an inversion is required
- As $2^{255} - 19 \equiv 5 \pmod 8$, for each square α we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$
- Standard: Compute β, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$

Point decompression

中央研究院

- Before double-scalar multiplication: compute x coordinate x_A of A as

$$x_A = \pm\sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$$

- Looks like a square root and an inversion is required
- As $2^{255} - 19 \equiv 5 \pmod 8$, for each square α we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$
- Standard: Compute β, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$
- Decompression has $\alpha = u/v$, merge square root with inversion:

$$\beta = (u/v)^{(q+3)/8}$$

Point decompression

中央研究院

- Before double-scalar multiplication: compute x coordinate x_A of A as
$$x_A = \pm\sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$$
- Looks like a square root and an inversion is required
- As $2^{255} - 19 \equiv 5 \pmod 8$, for each square α we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$
- Standard: Compute β, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$
- Decompression has $\alpha = u/v$, merge square root with inversion:
$$\begin{aligned}\beta = (u/v)^{(q+3)/8} &= u^{(q+3)/8} v^{q-1-(q+3)/8} \\ &= u^{(q+3)/8} v^{(7q-11)/8} = uv^3(uv^7)^{(q-5)/8}.\end{aligned}$$
- Only one big exponentiation, cost similar to just inversion with Fermat

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i
- Choose independent uniform random 128-bit integers z_i
- Compute $H_i = \text{SHA-}512(\underline{R_i}, \underline{A_i}, M_i)$

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i
- Choose independent uniform random 128-bit integers z_i
- Compute $H_i = \text{SHA-}512(\underline{R_i}, \underline{A_i}, M_i)$
- Verify the equation

$$\left(-\sum_i z_i S_i \bmod \ell\right)B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod \ell) A_i = 0$$

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i
- Choose independent uniform random 128-bit integers z_i
- Compute $H_i = \text{SHA-}512(\underline{R_i}, \underline{A_i}, M_i)$
- Verify the equation

$$\left(-\sum_i z_i S_i \bmod \ell\right)B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod \ell) A_i = 0$$

- Use Bos-Coster algorithm for multi-scalar multiplication

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i
- Choose independent uniform random 128-bit integers z_i
- Compute $H_i = \text{SHA-}512(\underline{R_i}, \underline{A_i}, M_i)$
- Verify the equation

$$\left(-\sum_i z_i S_i \bmod \ell\right)B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod \ell)A_i = 0$$

- Use Bos-Coster algorithm for multi-scalar multiplication
- Karati, Das, Roychowdhury, Bellur, Bhattacharya, and Lyer at Africacrypt 2012: Batch verification without randomizers; **broken** by Bernstein, Doumen, Lange, and Oosterwijk (Indocrypt 2012)

Faster batch verification

中央研究院

- Verify a batch of (M_i, A_i, R_i, S_i), where (R_i, S_i) is the alleged signature of M_i under key A_i
- Choose independent uniform random 128-bit integers z_i
- Compute $H_i = \text{SHA-}512(\underline{R_i}, \underline{A_i}, M_i)$
- Verify the equation

$$\left(-\sum_i z_i S_i \bmod \ell\right) B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod \ell) A_i = 0$$

- Use Bos-Coster algorithm for multi-scalar multiplication
- Karati, Das, Roychowdhury, Bellur, Bhattacharya, and Lyer at Africacrypt 2012: Batch verification without randomizers; **broken** by Bernstein, Doumen, Lange, and Oosterwijk (Indocrypt 2012)
- Same Indocrypt 2012 paper: faster batch forgery identification

The Bos-Coster algorithm

中央研究院

- Computation of $Q = \sum_1^n s_i P_i$

The Bos-Coster algorithm

中央研究院

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Each step "eliminates" expected $\log n$ scalar bits

The Bos-Coster algorithm

中央研究院

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Each step "eliminates" expected $\log n$ scalar bits
- Requires fast access to the two largest scalars: put scalars into a heap
- Crucial for good performance: fast heap implementation

A fast heap

中央研究院

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i - 1)/2 \rfloor$

A fast heap

中央研究院

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i-1)/2 \rfloor$
- Typical heap root replacement (pop operation): start at the root, swap down for a variable amount of times

A fast heap

中央研究院

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i-1)/2 \rfloor$
- Typical heap root replacement (pop operation): start at the root, swap down for a variable amount of times
- Floyd's heap: swap down to the bottom, swap up for a variable amount of times, advantages:
  - Each swap-down step needs only one comparison (instead of two)
  - Swap-down loop is more friendly to branch predictors

The Bos-Coster algorithm

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Each step "eliminates" expected $\log n$ scalar bits
- Requires fast access to the two largest scalars: put scalars into a heap
- Crucial for good performance: fast heap implementation

The Bos-Coster algorithm

中央研究院

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Each step "eliminates" expected $\log n$ scalar bits
- Requires fast access to the two largest scalars: put scalars into a heap
- Crucial for good performance: fast heap implementation
- Further optimization: Start with heap without the z_i until largest scalar has ≤ 128 bits
- Then: extend heap with the z_i

The Bos-Coster algorithm

中央研究院

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Each step "eliminates" expected $\log n$ scalar bits
- Requires fast access to the two largest scalars: put scalars into a heap
- Crucial for good performance: fast heap implementation
- Further optimization: Start with heap without the z_i until largest scalar has ≤ 128 bits
- Then: extend heap with the z_i
- Optimize the heap on the assembly level

Ed25519 performance

中央研究院

Performance on Intel Nehalem/Westmere

- 87548 cycles for signing
- 273364 cycles for verification
- 8550000 cycles to verify a batch of 64 valid signatures ($\ll 134000$ cycles per signature)

Performance on ARM Cortex A8

- Bernstein, Schwabe (2012): 244655 cycles for signing
- Bernstein, Schwabe (2012): 624846 cycles for verification

Ed25519 performance

中央研究院

Performance on Intel Nehalem/Westmere

- 87548 cycles for signing
- 273364 cycles for verification
- 8550000 cycles to verify a batch of 64 valid signatures ($\ll 134000$ cycles per signature)

Performance on ARM Cortex A8

- Bernstein, Schwabe (2012): 244655 cycles for signing
- Bernstein, Schwabe (2012): 624846 cycles for verification

Followup results by Hamburg

- $52000/170000$ cycles for signing/verification on Sandy Bridge
- $256000/624000$ cycles for signing/verification on Cortex A9

Pollard rho for the ECDLP

中央研究院

- So far: Branches and table lookups were bad with secret scalars
- They should be no problem at all in cryptanalysis
- Consider the parallel Pollard rho algorithm to find k, given P and $Q = kP$ in $G \subseteq E(\mathbb{F}_q)$

Parallel Pollard rho (clients)

中央研究院

- Use pseudorandom function f
- Start with $W_0 = n_0 P + m_0 Q$ for random n_0, m_0
- Iteratively apply f to obtain $W_{i+1} = f(W_i)$
- At each step, check whether W_i is a *distinguished point (DP)*, e.g., "last k bits of W_i's encoding are 0"
- When finding a DP W_d: send (n_0, m_0, W_d) to the server, start with new W_0

Parallel Pollard rho (server)

中央研究院

- Server searches in incoming data for collisions (n_0, m_0, W_d), (n'_0, m'_0, W_d)
- Recomputes the two walks to W_d, updates n_i, m_i and n'_i, m'_i to obtain n_d, m_d, n'_d, m'_d with

$$n_d P + m_d Q = n'_d P + m'_d Q = W_d$$

- Computes discrete log

$$k = (n_d - n'_d)/(m'_d - m_d) \pmod{|G|}$$

- Note that f needs to preserve knowledge about the linear combination in P and Q

Additive walks

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication

Additive walks

中央研究院

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, \dots, r-1\}$
- Define $f(W) = W + R_{h(W)}$

Additive walks

中央研究院

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, \dots, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition

Additive walks

中央研究院

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, \dots, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticeably nonrandom, they require more iterations

Additive walks

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, \dots, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticeably nonrandom, they require more iterations
- Teske showed that large r provides close-to-random behavior (e.g. $r = 20$)

Additive walks

中央研究院

- Easy way to define f:

$$f(W) = n(W)P + m(W)Q$$

  with pseudorandom functions n, m
- Cost: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, \dots, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticeably nonrandom, they require more iterations
- Teske showed that large r provides close-to-random behavior (e.g. $r = 20$)
- Summary: additive walks provide much better performance in practice

Walks modulo negation

中央研究院

- So far, everything worked with any group G
- Now consider groups of points on elliptic curves
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$

Walks modulo negation

中央研究院

- So far, everything worked with any group G
- Now consider groups of points on elliptic curves
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves, GLS curves, and BN curves

Walks modulo negation

中央研究院

- So far, everything worked with any group G
- Now consider groups of points on elliptic curves
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves, GLS curves, and BN curves
- Idea: Define iterations on equivalence classes modulo negation
- For example: always take the lexicographic minimum of $(x, -y)$ and (x, y)

Walks modulo negation

中央研究院

- So far, everything worked with any group G
- Now consider groups of points on elliptic curves
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves, GLS curves, and BN curves
- Idea: Define iterations on equivalence classes modulo negation
- For example: always take the lexicographic minimum of $(x, -y)$ and (x, y)
- This halves the size of the search space, expected number of iterations drops by a factor of $\sqrt{2}$

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

- Probability for such fruitless cycles: $1/2r$

Putting it together

中央研究院

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

- Probability for such fruitless cycles: $1/2r$
- Similar observations hold for longer fruitless cycles (length $4, 6, \ldots$)
- Probability of a cycle of length $2c$ is $\approx 1/r^c$

How expensive are fruitless cycles

中央研究院

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles

How expensive are fruitless cycles

中央研究院

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

> *"We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"*

How expensive are fruitless cycles

中央研究院

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

  > *"We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"*

- Paper at ANTS 2010 by Bos, Kleinjung, and Lenstra: Among many ways of dealing with fruitless cycles best speedup is 1.29, but

How expensive are fruitless cycles

中央研究院

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

  > *"We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"*

- Paper at ANTS 2010 by Bos, Kleinjung, and Lenstra: Among many ways of dealing with fruitless cycles best speedup is 1.29, but

  > *"If the Pollard rho method is parallelized in SIMD fashion, it is a challenge to achieve any speedup at all. . . . Dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. . . . [This] is a major obstacle to the negation map in SIMD environments."*

Why are fruitless cycles so expensive? 中央研究院

The problem with large tables

- Probability of fruitless cycles gets smaller with larger r
- Using a huge r seems like an obvious fix

Why are fruitless cycles so expensive?

中央研究院

The problem with large tables

- Probability of fruitless cycles gets smaller with larger r
- Using a huge r seems like an obvious fix, but:
- precomputed points won't fit into cache $\rightarrow$ performance penalty from slow loads

Why are fruitless cycles so expensive?

中央研究院

The problem with large tables

- Probability of fruitless cycles gets smaller with larger r
- Using a huge r seems like an obvious fix, but:
- precomputed points won't fit into cache $\rightarrow$ performance penalty from slow loads

SIMD computations

- SIMD: Same sequence of instructions carried out on different data
- Branching means (in the worst case): Sequentially execute both branches

Why are fruitless cycles so expensive?

The problem with large tables

- Probability of fruitless cycles gets smaller with larger r
- Using a huge r seems like an obvious fix, but:
- precomputed points won't fit into cache $\rightarrow$ performance penalty from slow loads

SIMD computations

- SIMD: Same sequence of instructions carried out on different data
- Branching means (in the worst case): Sequentially execute both branches
- Computing power of the the PlayStation 3 is entirely based on SIMD computations
- SIMD becomes more and more important on all modern microprocessors

Our approach

中央研究院

- Joint work with Bernstein and Lange: Get the $\sqrt{2}$-speedup with SIMD
- Consider ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P

Our approach

中央研究院

- Joint work with Bernstein and Lange: Get the $\sqrt{2}$-speedup with SIMD
- Consider ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)

Our approach

中央研究院

- Joint work with Bernstein and Lange: Get the $\sqrt{2}$-speedup with SIMD
- Consider ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \ldots, p-1\}$, $(x, -y)$ otherwise

Our approach

中央研究院

- Joint work with Bernstein and Lange: Get the $\sqrt{2}$-speedup with SIMD
- Consider ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \ldots, p-1\}$, $(x, -y)$ otherwise
- *Occasionally* check for 2-cycles:
  - If $W_{i-1} = W_{i-3}$, set $W_i = |2\min\{W_{i-1}, W_{i-2}\}|$
  - Otherwise set $W_i = W_{i-1}$

Our approach

中央研究院

- Joint work with Bernstein and Lange: Get the $\sqrt{2}$-speedup with SIMD
- Consider ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \dots, p-1\}$, $(x, -y)$ otherwise
- *Occasionally* check for 2-cycles:
  - If $W_{i-1} = W_{i-3}$, set $W_i = |2 \min\{W_{i-1}, W_{i-2}\}|$
  - Otherwise set $W_i = W_{i-1}$
- With even lower frequency check for 4-cycles, 6-cycles etc.
- Implementation actually checks for 12-cycles (with very low frequency)

Eliminating branches

中央研究院

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$

Eliminating branches

中央研究院

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations

Eliminating branches

中央研究院

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations
- Always compute doublings, even if they are not used
- Select W_i from W_{i-1} and $2W_{\min}$ without branch
- Selection bit is output of branchfree comparison between W_{i-1} and W_{i-1-c} when detecting c-cycles

Eliminating branches

中央研究院

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations
- Always compute doublings, even if they are not used
- Select W_i from W_{i-1} and $2W_{\min}$ without branch
- Selection bit is output of branchfree comparison between W_{i-1} and W_{i-1-c} when detecting c-cycles
- All selections, subtractions, additions and comparisons are linear-time
- Asymptotically negligible compared to finite-field multiplications in EC arithmetic

Optimization and analysis

中央研究院

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$
- Checking without finding a fruitless cycle wastes one iteration

Optimization and analysis

中央研究院

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations

Optimization and analysis

中央研究院

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$

Optimization and analysis

中央研究院

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$
- Slowdown from fruitless cycles by a factor of $1 + \Theta(1/\sqrt{r})$

Optimization and analysis

中央研究院

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occurred: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$
- Slowdown from fruitless cycles by a factor of $1 + \Theta(1/\sqrt{r})$
- Negligible if $r \to \infty$ as $p \to \infty$

Solving the 112-bit ECDLP faster

中央研究院

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP

Solving the 112-bit ECDLP faster

中央研究院

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map

Solving the 112-bit ECDLP faster

中央研究院

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map
- Faster iterations
  - Faster arithmetic in $\mathbb{Z}/(2^{128}-3)\mathbb{Z}$ (prime field has order $(2^{128}-3)/76439$)
  - Non-standard radix $2^{12.8}$ to represent elements of $(2^{128}-3)/76439$
  - Careful design of iteration function, arithmetic and handling of fruitless cycles

Solving the 112-bit ECDLP faster

中央研究院

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map
- Faster iterations
  - Faster arithmetic in $\mathbb{Z}/(2^{128}-3)\mathbb{Z}$ (prime field has order $(2^{128}-3)/76439$)
  - Non-standard radix $2^{12.8}$ to represent elements of $(2^{128}-3)/76439$
  - Careful design of iteration function, arithmetic and handling of fruitless cycles
- Negligible overhead (in practice!) from fruitless cycles

References

中央研究院

Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang: **High-speed high-security signatures**.
http://cryptojedi.org/papers/#ed25519

Daniel J. Bernstein, Tanja Lange, and Peter Schwabe: **On the correct use of the negation map in the Pollard rho method.**
http://cryptojedi.org/papers/#negation

Daniel J. Bernstein and Peter Schwabe: **NEON crypto.**
http://cryptojedi.org/papers/#neoncrypto