High-speed high-security signatures

Peter Schwabe

National Taiwan University

Joint work with Daniel J. Bernstein, Niels Duif, Tanja Lange, and Bo-Yin Yang

September 29, 2011

CHES 2011, Nara, Japan
Summary

- Elliptic-curve signature scheme and corresponding software
- Based on arithmetic on twisted Edwards curves
Summary

- Elliptic-curve signature scheme and corresponding software
- Based on arithmetic on twisted Edwards curves

Security features

- 128 bits of security
- Timing-attack resistant implementation
- Foolproof session keys
- Hash-function-collision resilience
Summary

- Elliptic-curve signature scheme and corresponding software
- Based on arithmetic on twisted Edwards curves

Security features

- 128 bits of security
- Timing-attack resistant implementation
- Foolproof session keys
- Hash-function-collision resilience

Speed features

- Fast signing: 87548 cycles on Intel Nehalem/Westmere
- Fast verification: 273364 cycles
- Even faster batch verification: < 134000 cycles/signature
- Fast key generation: 93288 cycles
- Short signatures (64 bytes), short public keys (32 bytes)
Recall Schnorr signatures

- Variant of ElGamal Signatures
- Many more variants (DSA, ECDSA, KCDSA, . . .)
- Uses finite group $G = \langle B \rangle$, with $|G| = \ell$
- Uses hash-function $H : G \times \mathbb{Z} \to \{0, \ldots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
Recall Schnorr signatures

- Variant of ElGamal Signatures
- Many more variants (DSA, ECDSA, KCDSA, ...)
- Uses finite group $G = \langle B \rangle$, with $|G| = \ell$
- Uses hash-function $H : G \times \mathbb{Z} \rightarrow \{0, \ldots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
- Private key: $a \in \{1, \ldots, \ell\}$, public key: $A = -aB$
Recall Schnorr signatures

- Variant of ElGamal Signatures
- Many more variants (DSA, ECDSA, KCDSA, ...)
- Uses finite group $G = \langle B \rangle$, with $|G| = \ell$
- Uses hash-function $H : G \times \mathbb{Z} \to \{0, \ldots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
- Private key: $a \in \{1, \ldots, \ell\}$, public key: $A = -aB$
- Sign: Generate secret random $r \in \{1, \ldots, \ell\}$, compute signature $(H(R, M), S)$ on $M$ with

\[
R = rB \\
S = (r + H(R, M)a) \mod \ell
\]
Recall Schnorr signatures

- Variant of ElGamal Signatures
- Many more variants (DSA, ECDSA, KCDSA, ...)
- Uses finite group $G = \langle B \rangle$, with $|G| = \ell$
- Uses hash-function $H : G \times \mathbb{Z} \to \{0, \ldots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
- Private key: $a \in \{1, \ldots, \ell\}$, public key: $A = -aB$
- Sign: Generate secret random $r \in \{1, \ldots, \ell\}$, compute signature $(H(R, M), S)$ on $M$ with

\[
R = rB \\
S = (r + H(R, M)a) \mod \ell
\]

- Verifier computes $\overline{R} = SB + H(R, M)A$ and checks that

\[
H(\overline{R}, M) = H(R, M)
\]
EdDSA and Ed25519 parameters

EdDSA

- Integer $b \geq 10$

Ed25519-SHA-512

- $b = 256$
EdDSA and Ed25519 parameters

**EdDSA**
- Integer \( b \geq 10 \)
- Prime power \( q \equiv 1 \pmod{4} \)
- \((b - 1)\)-bit encoding of elements of \( \mathbb{F}_q \)

**Ed25519-SHA-512**
- \( b = 256 \)
- \( q = 2^{255} - 19 \) (prime)
- little-endian encoding of \( \{0, \ldots, 2^{255} - 20\} \)
EdDSA and Ed25519 parameters

EdDSA

- Integer \( b \geq 10 \)
- Prime power \( q \equiv 1 \pmod{4} \)
- \((b - 1)\)-bit encoding of elements of \( \mathbb{F}_q \)
- Hash function \( H \) with \( 2b \)-bit output

Ed25519-SHA-512

- \( b = 256 \)
- \( q = 2^{255} - 19 \) (prime)
- little-endian encoding of \( \{0, \ldots, 2^{255} - 20\} \)
- \( H = \text{SHA-512} \)
EdDSA and Ed25519 parameters

EdDSA
- Integer $b \geq 10$
- Prime power $q \equiv 1 \pmod{4}$
- $(b - 1)$-bit encoding of elements of $\mathbb{F}_q$
- Hash function $H$ with $2b$-bit output
- Non-square $d \in \mathbb{F}_q$
- $B \in \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q, -x^2 + y^2 = 1 + dx^2 y^2 \}$ (twisted Edwards curve $E$)
- prime $\ell \in (2^{b-4}, 2^{b-3})$ with $\ell B = (0, 1)$

Ed25519-SHA-512
- $b = 256$
- $q = 2^{255} - 19$ (prime)
- little-endian encoding of $\{0, \ldots, 2^{255} - 20\}$
- $H = \text{SHA-512}$
- $d = -121665/121666$
- $B = (x, 4/5)$, with $x$ “even”
- $\ell$ a 253-bit prime

Ed25519 curve is birationally equivalent to the Curve25519 curve.

High-speed high-security signatures
EdDSA and Ed25519 parameters

**EdDSA**

- Integer \( b \geq 10 \)
- Prime power \( q \equiv 1 \pmod{4} \)
- \((b - 1)\)-bit encoding of elements of \( \mathbb{F}_q \)
- Hash function \( H \) with \( 2b \)-bit output
- Non-square \( d \in \mathbb{F}_q \)
- \( B \in \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q, -x^2 + y^2 = 1 + dx^2y^2\} \) (twisted Edwards curve \( E \))
- prime \( \ell \in (2^{b-4}, 2^{b-3}) \) with \( \ell B = (0, 1) \)

**Ed25519-SHA-512**

- \( b = 256 \)
- \( q = 2^{255} - 19 \) (prime)
- little-endian encoding of \( \{0, \ldots, 2^{255} - 20\} \)
- \( H = SHA-512 \)
- \( d = -121665/121666 \)
- \( B = (x, 4/5), \) with \( x \) “even”

Ed25519 curve is birationally equivalent to the Curve25519 curve.
EdDSA keys

- Secret key: $b$-bit string $k$
- Compute $H(k) = (h_0, \ldots, h_{2b-1})$
EdDSA keys

- Secret key: $b$-bit string $k$
- Compute $H(k) = (h_0, \ldots, h_{2b-1})$
- Derive integer $a = 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$
- Note that $a$ is a multiple of 8
EdDSA keys

- Secret key: $b$-bit string $k$
- Compute $H(k) = (h_0, \ldots, h_{2b-1})$
- Derive integer $a = 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$
- Note that $a$ is a multiple of 8
- Compute $A = aB$
- Public key: Encoding $A$ of $A = (x_A, y_A)$ as $y_A$ and one (parity) bit of $x_A$ (needs $b$ bits)
EdDSA keys

- Secret key: $b$-bit string $k$
- Compute $H(k) = (h_0, \ldots, h_{2b-1})$
- Derive integer $a = 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$
- Note that $a$ is a multiple of 8
- Compute $A = aB$
- Public key: Encoding $A$ of $A = (x_A, y_A)$ as $y_A$ and one (parity) bit of $x_A$ (needs $b$ bits)
- Compute $A$ from $A$: $x_A = \pm \sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$
EdDSA signatures

Signing

- Message $M$ determines $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, \ldots, 2^{2b} - 1\}$
- Define $R = rB$
- Define $S = (r + H(R, A, M)a) \mod \ell$
- Signature: $(R, S)$, with $S$ the $b$-bit little-endian encoding of $S$
- $(R, S)$ has $2b$ bits (3 known to be zero)
EdDSA signatures

Signing

- Message $M$ determines $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, \ldots, 2^{2b} - 1\}$
- Define $R = rB$
- Define $S = (r + H(R, A, M)a) \mod \ell$
- Signature: $(R, S)$, with $S$ the $b$-bit little-endian encoding of $S$
- $(R, S)$ has $2b$ bits (3 known to be zero)

Verification

- Verifier parses $A$ from $A$ and $R$ from $R$
- Computes $H(R, A, M)$
- Checks group equation

$$8SB = 8R + 8H(R, A, M)A$$

- Rejects if parsing fails or equation does not hold
Collision resilience

- ECDSA uses $H(M)$
- Collisions in $H$ allow existential forgery
Collision resilience

- ECDSA uses $H(M)$
- Collisions in $H$ allow existential forgery
- Schnorr signatures and EdDSA include $R$ in the hash
  - Schnorr: $H(R, M)$
  - EdDSA: $H(R, A, M)$
- Signatures are hash-function-collision resilient
Collision resilience

- ECDSA uses $H(M)$
- Collisions in $H$ allow existential forgery
- Schnorr signatures and EdDSA include $R$ in the hash
  - Schnorr: $H(R, M)$
  - EdDSA: $H(R, A, M)$
- Signatures are hash-function-collision resilient
- Including $A$ alleviates concerns about attacks against multiple keys
Foolproof session keys

- Each message needs a different, hard-to-predict $r$ ("session key")
- Just knowing a few bits of $r$ for many signatures allows to recover $a$
- Usual approach (e.g., Schnorr signatures): Choose random $r$ for each message
Foolproof session keys

- Each message needs a different, hard-to-predict $r$ (“session key”)
- Just knowing a few bits of $r$ for many signatures allows to recover $a$
- Usual approach (e.g., Schnorr signatures): Choose random $r$ for each message
- Potential problems: Bad random-number generators, off-by-one(-byte) bugs
Foolproof session keys

- Each message needs a different, hard-to-predict $r$ ("session key")
- Just knowing a few bits of $r$ for many signatures allows to recover $a$
- Usual approach (e.g., Schnorr signatures): Choose random $r$ for each message
- Potential problems: Bad random-number generators, off-by-one(-byte) bugs
- Even worse: No random-number generator: Sony’s PS3 security disaster

EdDSA uses deterministic, pseudo-random session keys

$H(h_b, \ldots, h_{2b-1}, M)$

- Same security as random under standard PRF assumptions
- Does not consume per-message randomness
- Better for testing (deterministic output)

High-speed high-security signatures
Foolproof session keys

- Each message needs a different, hard-to-predict $r$ (“session key”)
- Just knowing a few bits of $r$ for many signatures allows to recover $a$
- Usual approach (e.g., Schnorr signatures): Choose random $r$ for each message
- Potential problems: Bad random-number generators, off-by-one(-byte) bugs
- Even worse: No random-number generator: Sony’s PS3 security disaster
- EdDSA uses deterministic, pseudo-random session keys $H(h_b, \ldots, h_{2b-1}, M)$
Foolproof session keys

- Each message needs a different, hard-to-predict $r$ (“session key”)
- Just knowing a few bits of $r$ for many signatures allows to recover $a$
- Usual approach (e.g., Schnorr signatures): Choose random $r$ for each message
- Potential problems: Bad random-number generators, off-by-one(-byte) bugs
- Even worse: No random-number generator: Sony’s PS3 security disaster
- EdDSA uses deterministic, pseudo-random session keys $H(h_b, \ldots, h_{2b-1}, M)$
- Same security as random $r$ under standard PRF assumptions
- Does not consume per-message randomness
- Better for testing (deterministic output)
Fast arithmetic in $\mathbb{F}_{2^{255}-19}$

Radix $2^{64}$

- Standard: break elements of $\mathbb{F}_{2^{255}-19}$ into 4 64-bit integers
- (Schoolbook) multiplication breaks down into 16 64-bit integer multiplications
- Adding up partial results requires many add-with-carry (adc)
- Westmere bottleneck: 1 adc every two cycles vs. 3 add per cycle
Fast arithmetic in $\mathbb{F}_{2^{255}-19}$

Radix $2^{64}$

- Standard: break elements of $\mathbb{F}_{2^{255}-19}$ into 4 64-bit integers
- (Schoolbook) multiplication breaks down into 16 64-bit integer multiplications
- Adding up partial results requires many add-with-carry (adc)
- Westmere bottleneck: 1 adc every two cycles vs. 3 add per cycle

Radix $2^{51}$

- Instead break into 5 64-bit integers, use radix $2^{51}$
- Schoolbook multiplication now 25 64-bit integer multiplications
- Partial results have $< 128$ bits, adding upper part is add, not adc
- Easy to merge multiplication with reduction (multiplies by 19)
- Better performance on Westmere/Nehalem, worse on 65 nm Core 2 and AMD processors
Fast signing

- Main computational task: Compute $R = rB$

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

- Wait, table lookups?

- In each lookup load all 8 relevant entries from the table, use arithmetic to obtain the desired one

- Signing takes 87548 cycles on an Intel Westmere CPU

- Key generation takes about 6000 cycles more (read from \(/dev/urandom\) )
Fast signing

- Main computational task: Compute $R = rB$
- First compute $r \mod \ell$, write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, with

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

- Main computational task: Compute $R = rB$
- 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
Fast signing

- Main computational task: Compute $R = rB$
- 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 $R = \sum_{i=0}^{63} 16^i r_i B$

Wait, table lookups?

In each lookup load all 8 relevant entries from the table, use arithmetic to obtain the desired one

Signing takes 87,548 cycles on an Intel Westmere CPU

Key generation takes about 6,000 cycles more (read from /dev/urandom)

High-speed high-security signatures
Fast signing

- Main computational task: Compute $R = rB$
- 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
Fast signing

- Main computational task: Compute $R = rB$
- 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
- Wait, table lookups?
Fast signing

- Main computational task: Compute $R = rB$
- 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
- Wait, table lookups?
- In each lookup load all 8 relevant entries from the table, use arithmetic to obtain the desired one
Fast signing

- Main computational task: Compute $R = rB$
- 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
- Wait, table lookups?
- In each lookup load all 8 relevant entries from the table, use arithmetic to obtain the desired one
- Signing takes 87548 cycles on an Intel Westmere CPU
- Key generation takes about 6000 cycles more (read from /dev/urandom)
Fast verification

- First part: point decompression, compute $x$ coordinate $x_R$ of $R$ as

$$x_R = \pm \sqrt{\frac{y_R^2 - 1}{dy_R^2 + 1}}$$

- Looks like a square root and an inversion is required
Fast verification

- First part: point decompression, compute $x$ coordinate $x_R$ of $R$ as

$$x_R = \pm \sqrt{(y_R^2 - 1)/(dy_R^2 + 1)}$$

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

- First part: point decompression, compute $x$ coordinate $x_R$ of $R$ as

$$x_R = \pm \sqrt{(y_R^2 - 1)/(d y_R^2 + 1)}$$

- Looks like a square root and an inversion is required

- As $q \equiv 5 \pmod{8}$ for each square $\alpha$ we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$

- Standard: Compute $\beta$, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$

- Decompression has $\alpha = u/v$, merge square root with inversion:

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

- First part: point decompression, compute \( x \) coordinate \( x_R \) of \( R \) as

\[
x_R = \pm \sqrt{(y_R^2 - 1)/(d y_R^2 + 1)}
\]

- Looks like a square root and an inversion is required

- As \( q \equiv 5 \pmod{8} \) for each square \( \alpha \) we have \( \alpha^2 = \beta^4 \), with \( \beta = \alpha^{(q+3)/8} \)

- Standard: Compute \( \beta \), conditionally multiply by \( \sqrt{-1} \) if \( \beta^2 = -\alpha \)

- Decompression has \( \alpha = u/v \), merge square root with inversion:

\[
\beta = (u/v)^{(q+3)/8} = u^{(q+3)/8}v^{q-1-(q+3)/8} = u^{(q+3)/8}v^{(7q-11)/8} = uv^3(uv^7)^{(q-5)/8}.
\]
Fast verification

▶ First part: point decompression, compute $x$ coordinate $x_R$ of $R$ as

$$x_R = \pm \sqrt{(y_R^2 - 1)/(dy_R^2 + 1)}$$

▶ Looks like a square root and an inversion is required

▶ As $q \equiv 5 \pmod{8}$ for each square $\alpha$ we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$

▶ Standard: Compute $\beta$, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$

▶ Decompression has $\alpha = u/v$, merge square root with inversion:

$$\beta = (u/v)^{(q+3)/8} = u^{(q+3)/8}v^{q-1-(q+3)/8}$$
$$= u^{(q+3)/8}v^{(7q-11)/8} = uv^3(uv^7)^{(q-5)/8}.$$ 

▶ Second part: computation of $SB - H(\underline{R}, \underline{A}, M)A$

▶ Double-scalar multiplication using signed sliding windows

▶ Different window sizes for $B$ (compile time) and $A$ (run time)
Fast verification

- First part: point decompression, compute $x$ coordinate $x_R$ of $R$ as

$$x_R = \pm \sqrt{(y_R^2 - 1)/(dy_R^2 + 1)}$$

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

$$\beta = (u/v)^{(q+3)/8} = u^{(q+3)/8}v^{q-1-(q+3)/8} = u^{(q+3)/8}v^{(7q-11)/8} = uv^3(uv^7)^{(q-5)/8}.$$

- Second part: computation of $SB - H(R, A, M)A$
- Double-scalar multiplication using signed sliding windows
- Different window sizes for $B$ (compile time) and $A$ (run time)
- Verification takes 273364 cycles
Faster batch verification

- 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

- 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 \(H_i = H(R_i, A_i, M_i)\)
Faster batch verification

- 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 \(H_i = H(R_i, A_i, M_i)\)
- Verify the equation

\[
\left(- \sum_i z_i S_i \mod \ell\right)B + \sum_i z_i R_i + \sum_i (z_i H_i \mod \ell)A_i = 0
\]
Faster batch verification

- 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 \(H_i = H(R_i, A_i, M_i)\)
- Verify the equation

\[
\left( - \sum_i z_i S_i \mod \ell \right) B + \sum_i z_i R_i + \sum_i (z_i H_i \mod \ell) A_i = 0
\]

- Use Bos-Coster algorithm for multi-scalar multiplication
Faster batch verification

- 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 \(H_i = H(R_i, A_i, M_i)\).
- Verify the equation
  \[
  \left(-\sum_i z_i S_i \mod \ell\right) B + \sum_i z_i R_i + \sum_i (z_i H_i \mod \ell) A_i = 0
  \]
- Use Bos-Coster algorithm for multi-scalar multiplication.
- Verifying a batch of 64 signatures takes 8.55 million cycles (i.e., < 134000 cycles/signature).

High-speed high-security signatures
The Bos-Coster algorithm

- Computation of \( Q = \sum_{1}^{n} s_i P_i \)
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 $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3P_3 + \cdots + s_nP_n$
- 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

- Computation of \( Q = \sum_{1}^{n} s_i P_i \)
- Idea: Assume \( s_1 > s_2 > \cdots > s_n \). Recursively compute
  \[
  Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3P_3 \cdots + s_nP_n
  \]
- 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

- Computation of \( Q = \sum_{i=1}^{n} s_i P_i \)
- Idea: Assume \( s_1 > s_2 > \cdots > s_n \). Recursively compute
  \[ Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3P_3 \cdots + s_nP_n \]
- 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
- Typical heap root replacement (pop operation): start at the root,
  swap down for a variable amount of times
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 $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3P_3 \cdots + s_nP_n$
- 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
- 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
Results

- New fast and secure signature scheme
- (Slow) C and Python reference implementations
- Fast AMD64 assembly implementations
- Also new speed records for Curve25519 ECDH
- All software in the public domain and included in eBATS
- All reported benchmarks (except batch verification) are eBATS benchmarks
- All reported benchmarks had TurboBoost switched off
- Software to be included in the NaCl library

http://ed25519.cr.yp.to/