EdDSA signatures and Ed25519

Peter Schwabe

中央研究院

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

February 20, 2012

Coding Theory and Cryptography Seminar, University of Basel

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate
- 99 summits over 3000 meters (highest peak: 3952 m)
- Wildlife includes black bears, salmon, monkeys...

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate
- 99 summits over 3000 meters (highest peak: 3952 m)
- Wildlife includes black bears, salmon, monkeys...
- Academia Sinica is a research facility funded by ROC
- About 30 institutes
- About 800 principal investigators, more than 750 postdocs

Introduction – the NaCl library

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced "salt")

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced "salt")
- Aim of this library: High-speed, high-security, easy-to-use cryptographic protection for network communication

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced "salt")
- Aim of this library: High-speed, high-security, easy-to-use cryptographic protection for network communication
- We are willing to sacrifice compatability to other crypto libraries

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced “salt”)
- Aim of this library: High-speed, high-security, easy-to-use cryptographic protection for network communication
- We are willing to sacrifice compatability to other crypto libraries
- At the end of 2010 the library contained
  - the stream cipher Salsa20,
  - the Poly1305 secret-key authenticator, and
  - Curve25519 elliptic-curve Diffie-Hellman key-exchange software.

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced "salt")
- Aim of this library: High-speed, high-security, easy-to-use cryptographic protection for network communication
- We are willing to sacrifice compatability to other crypto libraries
- At the end of 2010 the library contained
  - the stream cipher Salsa20,
  - the Poly1305 secret-key authenticator, and
  - Curve25519 elliptic-curve Diffie-Hellman key-exchange software.
- This is wrapped in a `crypto_box` API that performs high-security public-key authenticated encryption
- This serves the typical one-to-one communication of most internet connections

How it started

- My research during Ph.D. was within the European project CACE (Computer Aided Cryptography Engineering)
- One of the deliverables: Networking and Cryptography Library (NaCl, pronounced "salt")
- Aim of this library: High-speed, high-security, easy-to-use cryptographic protection for network communication
- We are willing to sacrifice compatability to other crypto libraries
- At the end of 2010 the library contained
  - the stream cipher Salsa20,
  - the Poly1305 secret-key authenticator, and
  - Curve25519 elliptic-curve Diffie-Hellman key-exchange software.
- This is wrapped in a `crypto_box` API that performs high-security public-key authenticated encryption
- This serves the typical one-to-one communication of most internet connections
- Still required at the end of 2010: One-to-many authentication, i.e. cryptographic signatures

Designing a public-key signature scheme

- Core requirements: 128-bit security, fast signing, fast verification, secure software implementation
- Obvious candidates: RSA, ElGamal, DSA, ECDSA, Schnorr...

Designing a public-key signature scheme

- Core requirements: 128-bit security, fast signing, fast verification, secure software implementation
- Obvious candidates: RSA, ElGamal, DSA, ECDSA, Schnorr...
- Conventional wisdom: ECC is faster than anything based on factoring or the DLP in $\mathbb{Z}_n^*$
- (Twisted) Edwards curves support very fast arithmetic
- Edwards addition is complete (important for secure implementations)
- Curve25519 has an Edwards representation and offers very high security

Designing a public-key signature scheme

- Core requirements: 128-bit security, fast signing, fast verification, secure software implementation
- Obvious candidates: RSA, ElGamal, DSA, ECDSA, Schnorr...
- Conventional wisdom: ECC is faster than anything based on factoring or the DLP in $\mathbb{Z}_n^*$
- (Twisted) Edwards curves support very fast arithmetic
- Edwards addition is complete (important for secure implementations)
- Curve25519 has an Edwards representation and offers very high security
- Looks like "some" signature scheme using Edwards arithmetic on Curve25519 is a good choice

One step back: Is ECC really faster than, e.g., RSA?

- RSA with public exponent $e = 3$ can verify signatures with just one modular multiplication and one squaring
- Very hard to beat with any elliptic-curve-based signature scheme

One step back: Is ECC really faster than, e.g., RSA?

- RSA with public exponent $e = 3$ can verify signatures with just one modular multiplication and one squaring
- Very hard to beat with any elliptic-curve-based signature scheme
- Verification speed primarily matters in applications that need to verify many signatures
- Idea: To get close to RSA verification speed, support batch verification

One step back: Is ECC really faster than, e.g., RSA?

- RSA with public exponent $e = 3$ can verify signatures with just one modular multiplication and one squaring
- Very hard to beat with any elliptic-curve-based signature scheme
- Verification speed primarily matters in applications that need to verify many signatures
- Idea: To get close to RSA verification speed, support batch verification
- Easier: Verify batches of signatures under the same public key
- Harder (but much more useful!): Verify batches of signatures under different public keys
- We don't know where the NaCl library is used, so support the latter

One step back: Is ECC really faster than, e.g., RSA?

- RSA with public exponent $e = 3$ can verify signatures with just one modular multiplication and one squaring
- Very hard to beat with any elliptic-curve-based signature scheme
- Verification speed primarily matters in applications that need to verify many signatures
- Idea: To get close to RSA verification speed, support batch verification
- Easier: Verify batches of signatures under the same public key
- Harder (but much more useful!): Verify batches of signatures under different public keys
- We don't know where the NaCl library is used, so support the latter
- None of the above-mentioned schemes supports fast batch verification
- Schnorr signatures only require small changes (and have many nice features anyways)

One step back: Is ECC really faster than, e.g., RSA?

- RSA with public exponent $e = 3$ can verify signatures with just one modular multiplication and one squaring
- Very hard to beat with any elliptic-curve-based signature scheme
- Verification speed primarily matters in applications that need to verify many signatures
- Idea: To get close to RSA verification speed, support batch verification
- Easier: Verify batches of signatures under the same public key
- Harder (but much more useful!): Verify batches of signatures under different public keys
- We don't know where the NaCl library is used, so support the latter
- None of the above-mentioned schemes supports fast batch verification
- Schnorr signatures only require small changes (and have many nice features anyways)

$\Rightarrow$ Start with Schnorr signatures, modify as required

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 $H : G \times \mathbb{Z} \to \{0, \dots, 2^t - 1\}$
- 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 $H : G \times \mathbb{Z} \to \{0, \dots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
- Private key: $a \in \{1, \dots, \ell\}$, 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 $H : G \times \mathbb{Z} \to \{0, \dots, 2^t - 1\}$
- Originally: $G \leq \mathbb{F}_q^*$, here: consider elliptic-curve group
- Private key: $a \in \{1, \dots, \ell\}$, public key: $A = -aB$
- Sign: Generate secret random $r \in \{1, \dots, \ell\}$, 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

$$H(\overline{R}, M) = H(R, M)$$

The EdDSA signature scheme

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 $\{0, \ldots, 2^{255} - 20\}$
- $H =$ 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 $d \in \mathbb{F}_q$
- $B \in \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q, -x^2+y^2 = 1+dx^2y^2\}$ (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 $\{0, \dots, 2^{255} - 20\}$
- H = SHA-512
- $d = -121665/121666$
- $B = (x, 4/5)$, with x "even"
- ℓ a 253-bit prime

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"
- ℓ a 253-bit prime

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 $\underline{A}$ 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 $a = 2^{b-2} + \sum_{3 \le i \le b-3} 2^i h_i$
- Note that a is a multiple of 8
- Compute $A = aB$
- Public key: Encoding $\underline{A}$ of $A = (x_A, y_A)$ as y_A and one (parity) bit of x_A (needs b bits)
- Compute A from $\underline{A}$: $x_A = \pm\sqrt{(y_A^2 - 1)/(dy_A^2 + 1)}$

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 $S = (r + H(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature: $(\underline{R}, \underline{S})$, with $\underline{S}$ the b-bit little-endian encoding of S
- $(\underline{R}, \underline{S})$ 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 $S = (r + H(\underline{R}, \underline{A}, M)a) \mod \ell$
- Signature: $(\underline{R}, \underline{S})$, with $\underline{S}$ the b-bit little-endian encoding of S
- $(\underline{R}, \underline{S})$ has $2b$ bits (3 known to be zero)

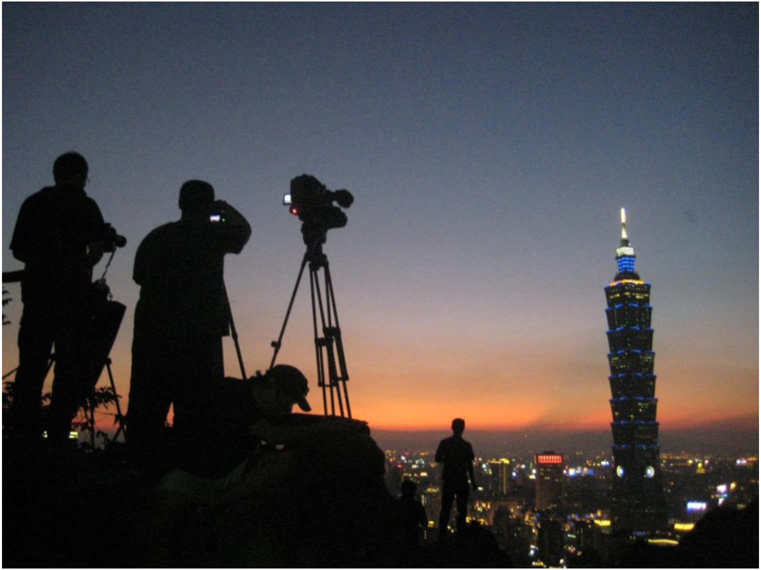
Verification

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

$$8SB = 8R + 8H(\underline{R}, \underline{A}, M)A$$

- Rejects if parsing fails or equation does not hold

EdDSA and Ed25519 security



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 $\underline{R}$ in the hash
  - Schnorr: $H(\underline{R}, M)$
  - EdDSA: $H(\underline{R}, \underline{A}, M)$
- Signatures are hash-function-collision resilient

Collision resilience

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

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

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)

Constant-time implementation
Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

  ```
  if(s) do A
  else do B
  ```

  where `s` is a part (e.g., a bit) of the secret scalar

Constant-time implementation
Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

  ```
  if(s) do A
  else do B
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Program takes different amount of time depending on the value of `s`

Constant-time implementation

Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

  ```
  if(s) do A
  else do B
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Program takes different amount of time depending on the value of `s`
- This is true, even if `A` and `B` take the same amount of time!
- Reason: Branch predictors contained in all modern CPUs

Constant-time implementation
Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

```
if(s) do A
else do B
```

  where `s` is a part (e.g., a bit) of the secret scalar
- Program takes different amount of time depending on the value of `s`
- This is true, even if `A` and `B` take the same amount of time!
- Reason: Branch predictors contained in all modern CPUs
- Attacker can gain information about the secret scalar by timing the execution of the program

Constant-time implementation

Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

  ```
  if(s) do A
  else do B
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Program takes different amount of time depending on the value of `s`
- This is true, even if `A` and `B` take the same amount of time!
- Reason: Branch predictors contained in all modern CPUs
- Attacker can gain information about the secret scalar by timing the execution of the program
- In 2011, Brumley and Tuveri recoverd the OpenSSL ECDSA secret signing key through such a timing attack

Constant-time implementation

Avoiding secret branch conditions

- Many scalar-multiplication algorithms contain parts like

```
if(s) do A
else do B
```

  where `s` is a part (e.g., a bit) of the secret scalar
- Program takes different amount of time depending on the value of `s`
- This is true, even if `A` and `B` take the same amount of time!
- Reason: Branch predictors contained in all modern CPUs
- Attacker can gain information about the secret scalar by timing the execution of the program
- In 2011, Brumley and Tuveri recoverd the OpenSSL ECDSA secret signing key through such a timing attack
- **Ed25519 software does not contain any secret branch conditions**

Constant-time implementation
Avoiding secret lookup indices

- In particular fixed-basepoint scalar-multiplication algorithms contain parts like

  `P += precomputed_points[s]`

  where `s` is a part (e.g., a bit) of the secret scalar

Constant-time implementation

Avoiding secret lookup indices

- In particular fixed-basepoint scalar-multiplication algorithms contain parts like

  ```
  P += precomputed_points[s]
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Loading from memory can take a different amount of time depending on the (secret) address `s`
- Reason: Access to memory is cached, if data is found in cache the load is fast (cache hit), otherwise it's slow

Constant-time implementation
Avoiding secret lookup indices

- In particular fixed-basepoint scalar-multiplication algorithms contain parts like

  ```
  P += precomputed_points[s]
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Loading from memory can take a different amount of time depending on the (secret) address `s`
- Reason: Access to memory is cached, if data is found in cache the load is fast (cache hit), otherwise it's slow
- Again: Attacker can gain information about the secret scalar by timing the exeuction of the program

Constant-time implementation
Avoiding secret lookup indices

- In particular fixed-basepoint scalar-multiplication algorithms contain parts like

  ```
  P += precomputed_points[s]
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Loading from memory can take a different amount of time depending on the (secret) address `s`
- Reason: Access to memory is cached, if data is found in cache the load is fast (cache hit), otherwise it's slow
- Again: Attacker can gain information about the secret scalar by timing the exeuction of the program
- In 2005, Osvik, Shamir, and Tromer discovered the AES key used for hard-disk encryption in Linux in just 65 ms using such a cache-timing attack

Constant-time implementation

Avoiding secret lookup indices

- In particular fixed-basepoint scalar-multiplication algorithms contain parts like

  ```
  P += precomputed_points[s]
  ```

  where `s` is a part (e.g., a bit) of the secret scalar
- Loading from memory can take a different amount of time depending on the (secret) address `s`
- Reason: Access to memory is cached, if data is found in cache the load is fast (cache hit), otherwise it's slow
- Again: Attacker can gain information about the secret scalar by timing the exeuction of the program
- In 2005, Osvik, Shamir, and Tromer discovered the AES key used for hard-disk encryption in Linux in just 65 ms using such a cache-timing attack
- **Ed25519 software does not perform any loads from secret addresses**

Speed of Ed25519

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 $R = rB$

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

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

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

- 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 α we have $\alpha^2 = \beta^4$, with $\beta = \alpha^{(q+3)/8}$
- Standard: Compute β, conditionally multiply by $\sqrt{-1}$ if $\beta^2 = -\alpha$

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 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:

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

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 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:

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

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 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:

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

- 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 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:

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

- 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)
- 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 $H_i = H(\underline{R_i}, \underline{A_i}, M_i)$

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 $H_i = H(\underline{R_i}, \underline{A_i}, M_i)$
- 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

- 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 $H_i = H(\underline{R_i}, \underline{A_i}, M_i)$
- 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

- 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 $H_i = H(\underline{R_i}, \underline{A_i}, M_i)$
- 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
- Verifying a batch of 64 valid signatures takes 8.55 million cycles (i.e., < 134000 cycles/signature)

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 $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- 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

- 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

- 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

- 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

- 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

- 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

- 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

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/
http://nacl.cr.yp.to/