Tampere University

SIZE, SPEED, AND SECURITY:
An Ed25519 Case Study

Cesar Pereida García[1]

cesar.pereidagarcia@tuni.fi

Sampo Sovio

sampo.sovio@huawei.com

The 26th Nordic Conference on Secure IT Systems (NordSec)
Nov 29-30, 2021

Contents

Contents

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

EdDSA and Ed25519

- EdDSA is an EC variant of Schnorr signatures.
- Deterministic signatures.
- Bernstein et al. [3] describe Ed25519 as an instance of EdDSA over a twisted Edwards curve equivalent to Curve25519.
- Security level same as NIST P-256 $\rightarrow 2^{128}$.
- As fast as NIST P-256.
- Why EdDSA anyway?
  - Small signatures $\rightarrow$ 64 bytes.
  - Small public keys $\rightarrow$ 32 bytes.
  - Design decisions to prevent common attacks: nonce reuse, SCA.

Ed25519

Key Generation. Given a private key k generated randomly, a hash function H, and the base point B; the private scalar a, auxiliary key b, and public key A are computed as follows:

$$H(k) = (h_0, h_1, ..., h_{2n-1}) = (a, b) \qquad A = [a]B$$

Signature Generation. Given the private scalar a, the auxiliary key b, and a hash function H, the signature (R, S) on the message M is created as follows:

$$r = H(b, M) \qquad h = H(R, A, M)$$

$$R = [r]B \qquad S = (r + ha) \bmod \ell$$

Signature Verification. Given the base point B, the public key A, and the signature (R, S), on the message M, the signature is valid if satisfies the equation:

$$R = [S]B - [H(R, A, M)]A$$

- Requires: Fixed-point scalar multiplication, double-scalar multiplication, and scalar recoding algorithms.

Ed25519

Key Generation. Given a private key k generated randomly, a hash function H, and the base point B; the private scalar a, auxiliary key b, and public key A are computed as follows:

$$H(k) = (h_0, h_1, ..., h_{2n-1}) = (a, b) \qquad A = [a]B$$

Signature Generation. Given the private scalar a, the auxiliary key b, and a hash function H, the signature (R, S) on the message M is created as follows:

$$r = H(b, M) \qquad h = H(R, A, M)$$

$$R = [r]B \qquad S = (r + ha) \bmod \ell$$

Signature Verification. Given the base point B, the public key A, and the signature (R, S), on the message M, the signature is valid if satisfies the equation:

$$R = [S]B - [H(R, A, M)]A$$

- Requires: Fixed-point scalar multiplication, double-scalar multiplication, and scalar recoding algorithms.

Ed25519 Reference Implementation

- Reference implementations on benchmarking toolkit SUPERCOP[2].
  - `ref`
  - `ref10`
  - x86-specific `amd64-64-24k`
  - x86-specific `amd64-51-30k`
- Double-scalar multiplication
  - Signature verification
  - Own scalar recoding algorithm
  - Small precomputation
- Fixed-point scalar multiplication
  - Key generation
  - Signature generation
  - Own scalar recoding algorithm
  - Big precomputation table (30 KB)

[2] https://bench.cr.yp.to/supercop.html

Ed25519 Reference Implementation

- Reference implementations on benchmarking toolkit SUPERCOP[2].
  - `ref`
  - `ref10`
  - x86-specific `amd64-64-24k`
  - x86-specific `amd64-51-30k`
- Double-scalar multiplication
  - Signature verification
  - Own scalar recoding algorithm
  - Small precomputation
- Fixed-point scalar multiplication
  - Key generation
  - Signature generation
  - Own scalar recoding algorithm
  - Big precomputation table (30 KB)

[2] `https://bench.cr.yp.to/supercop.html`

Ed25519 Reference Implementation

- Reference implementations on benchmarking toolkit SUPERCOP[2].
  - `ref`
  - `ref10`
  - x86-specific `amd64-64-24k`
  - x86-specific `amd64-51-30k`
- Double-scalar multiplication
  - Signature verification
  - Own scalar recoding algorithm
  - Small precomputation
- Fixed-point scalar multiplication
  - Key generation
  - Signature generation
  - Own scalar recoding algorithm
  - Big precomputation table (30 KB)

[2] `https://bench.cr.yp.to/supercop.html`

Contents

(Badly) Optimizing for Size

- Why optimize?
  - SUPERCOP's `ref10` implementation is 100+ KB size.
  - IoT devices $\rightarrow$ "optimized" for memory size.
  - Can't use other implementations + inexperience.
- No big precomputation
  - Big precomputed table (30 KB) must go.
  - Small precomputation can stay (40 B).
- Combine scalar multiplication algorithms
  - Double-scalar multiplication
    $R = [S]B \cancel{- [H(R, A, M)]A} \rightarrow [S]B \rightarrow$ fixed-point scalar multiplication!
  - If A is `null` do fixed-point, otherwise do double-scalar.
- Remove redundancy
  - Only one scalar recoding algorithm.

(Badly) Optimizing for Size

- Why optimize?
  - SUPERCOP's `ref10` implementation is 100+ KB size.
  - IoT devices $\to$ "optimized" for memory size.
  - Can't use other implementations + inexperience.
- No big precomputation
  - Big precomputed table (30 KB) must go.
  - Small precomputation can stay (40 B).
- Combine scalar multiplication algorithms
  - Double-scalar multiplication
    $R = [S]B - \cancel{[H(R, A, M)]A} \to [S]B \to$ fixed-point scalar multiplication!
  - If A is `null` do fixed-point, otherwise do double-scalar.
- Remove redundancy
  - Only one scalar recoding algorithm.

(Badly) Optimizing for Size

- Why optimize?
  - SUPERCOP's `ref10` implementation is 100+ KB size.
  - IoT devices $\rightarrow$ "optimized" for memory size.
  - Can't use other implementations + inexperience.
- No big precomputation
  - Big precomputed table (30 KB) must go.
  - Small precomputation can stay (40 B).
- Combine scalar multiplication algorithms
  - Double-scalar multiplication
    $R = [S]B \cancel{-[H(R, A, M)]A} \rightarrow [S]B \rightarrow$ fixed-point scalar multiplication!
  - If A is `null` do fixed-point, otherwise do double-scalar.
- Remove redundancy
  - Only one scalar recoding algorithm.

(Badly) Optimizing for Size

- Why optimize?
  - SUPERCOP's `ref10` implementation is 100+ KB size.
  - IoT devices $\rightarrow$ "optimized" for memory size.
  - Can't use other implementations + inexperience.
- No big precomputation
  - Big precomputed table (30 KB) must go.
  - Small precomputation can stay (40 B).
- Combine scalar multiplication algorithms
  - Double-scalar multiplication
    $R = [S]B \cancel{- [H(R, A, M)]A} \rightarrow [S]B \rightarrow$ fixed-point scalar multiplication!
  - If A is `null` do fixed-point, otherwise do double-scalar.
- Remove redundancy
  - Only one scalar recoding algorithm.

Contents

What Went Wrong?

- Recommendations from an SCA perspective.
  - Constant-time algorithms.
  - No table-lookups based on secrets.
  - No branching based on secrets.
  - No looping based on secrets.
- Only one scalar recoding algorithm.
  - Branches on a per-bit basis.
  - Loops based on the scalar size.
  - Variable-time algorithm.
- Only one scalar multiplication algorithm.
  - Performs table-lookups based on the recoded secret scalar.
  - Branches based on the secret scalar, revealing some scalar bits.
  - Variable-time algorithm.

What Went Wrong?

- Recommendations from an SCA perspective.
  - Constant-time algorithms.
  - No table-lookups based on secrets.
  - No branching based on secrets.
  - No looping based on secrets.
- Only one scalar recoding algorithm.
  - Branches on a per-bit basis.
  - Loops based on the scalar size.
  - Variable-time algorithm.
- Only one scalar multiplication algorithm.
  - Performs table-lookups based on the recoded secret scalar.
  - Branches based on the secret scalar, revealing some scalar bits.
  - Variable-time algorithm.

What Went Wrong?

- Recommendations from an SCA perspective.
  - Constant-time algorithms.
  - No table-lookups based on secrets.
  - No branching based on secrets.
  - No looping based on secrets.
- Only one scalar recoding algorithm.
  - Branches on a per-bit basis.
  - Loops based on the scalar size.
  - Variable-time algorithm.
- Only one scalar multiplication algorithm.
  - Performs table-lookups based on the recoded secret scalar.
  - Branches based on the secret scalar, revealing some scalar bits.
  - Variable-time algorithm.

What Went Wrong?

- Recommendations from an SCA perspective.
  - Constant-time algorithms.
  - No table-lookups based on secrets.
  - No branching based on secrets.
  - No looping based on secrets.
- Only one scalar recoding algorithm.
  - Branches on a per-bit basis.
  - Loops based on the scalar size.
  - Variable-time algorithm.
- Only one scalar multiplication algorithm.
  - Performs table-lookups based on the recoded secret scalar.
  - Branches based on the secret scalar, revealing some scalar bits.
  - Variable-time algorithm.

What now?

- Undo optimizations and use reference implementation.
- Make constant-time what needs to be constant-time.
- Use other implementation.
  - Reference implementations on benchmarking toolkit SUPERCOP[3].
    - ref
    - ref10
    - x86-specific amd64-64-24k
    - x86-specific amd64-51-30k
  - donna implementation, portable, 32-bit and 64-bit.
  - Google's BoringSSL uses `fiat-crypto` for field arithmetic.
  - `Monocypher` targets small IoT devices.
  - OpenSSL
  - Mozilla's NSS
  - Many more...

[3] `https://bench.cr.yp.to/supercop.html`

What now?

- Undo optimizations and use reference implementation.
- Make constant-time what needs to be constant-time.
- Use other implementation.
  - Reference implementations on benchmarking toolkit SUPERCOP[3].
    - `ref`
    - `ref10`
    - `x86-specific amd64-64-24k`
    - `x86-specific amd64-51-30k`
  - donna implementation, portable, 32-bit and 64-bit.
  - Google's BoringSSL uses `fiat-crypto` for field arithmetic.
  - `Monocypher` targets small IoT devices.
  - OpenSSL
  - Mozilla's NSS
  - Many more...

[3] `https://bench.cr.yp.to/supercop.html`

What now?

- Undo optimizations and use reference implementation.
- Make constant-time what needs to be constant-time.
- Use other implementation.
  - Reference implementations on benchmarking toolkit SUPERCOP[3].
    - `ref`
    - `ref10`
    - x86-specific `amd64-64-24k`
    - x86-specific `amd64-51-30k`
  - `donna` implementation, portable, 32-bit and 64-bit.
  - Google's BoringSSL uses `fiat-crypto` for field arithmetic.
  - `Monocypher` targets small IoT devices.
  - OpenSSL
  - Mozilla's NSS
  - Many more...

[3] `https://bench.cr.yp.to/supercop.html`

Contents

ECCKiila[5]

"ECCKiila allows to dynamically create portable C-code (supporting both 64-bit and 32-bit architectures, no alignment or endianness assumptions) underlying Elliptic Curve Cryptography (ECC) cryptosystems."

- Computer-aided cryptography tool by Belyavsky et al. [1].
- Supports Weierstrass and Edwards curves.
- Generates Galois Field layer using `fiat-crypto`[4].
- Implements constant-time scalar multiplication algorithms.

[4] `https://github.com/mit-plv/fiat-crypto`

[5] `https://gitlab.com/nisec/ecckiila`

ECCKiila[5]

"ECCKiila allows to dynamically create portable C-code (supporting both 64-bit and 32-bit architectures, no alignment or endianness assumptions) underlying Elliptic Curve Cryptography (ECC) cryptosystems."

- Computer-aided cryptography tool by Belyavsky et al. [1].
- Supports Weierstrass and Edwards curves.
- Generates Galois Field layer using `fiat-crypto`[4].
- Implements constant-time scalar multiplication algorithms.

[4] `https://github.com/mit-plv/fiat-crypto`
[5] `https://gitlab.com/nisec/ecckiila`

ECCKiila-generated Ed25519

- ECCKiila: Field arithmetic + EC arithmetic.
- SUPERCOP: Ed25519 arithmetic $\rightarrow$ Point decompression, multiply and add, and modular reduction by the order of the base point.

- `ecckiila-no-precomp`
  - Made for 32-bits architectures (uses 10-limb integers).
  - Minimal precomputed table $\sim$ 2.5 KB.
  - Uses constant-time variable-point scalar multiplication with regular-NAF.
  - Uses double-scalar multiplication only for verification.
  - Roughly 40 kb in size.
- `ecckiila-precomp`
  - Made for both 32-bits and 64-bits architectures.
  - Precomputed table $\sim$ 30 KB.
  - Constant-time comb method fixed-point scalar multiplication.
  - Double-scalar multiplication only for verification.
  - Roughly 70 kb in size.

ECCKiila-generated Ed25519

- ECCKiila: Field arithmetic + EC arithmetic.
- SUPERCOP: Ed25519 arithmetic $\rightarrow$ Point decompression, multiply and add, and modular reduction by the order of the base point.

- `ecckiila-no-precomp`
  - Made for 32-bits architectures (uses 10-limb integers).
  - Minimal precomputed table $\sim$ 2.5 KB.
  - Uses constant-time variable-point scalar multiplication with regular-NAF.
  - Uses double-scalar multiplication only for verification.
  - Roughly 40 kb in size.

- `ecckiila-precomp`
  - Made for both 32-bits and 64-bits architectures.
  - Precomputed table $\sim$ 30 KB.
  - Constant-time comb method fixed-point scalar multiplication.
  - Double-scalar multiplication only for verification.
  - Roughly 70 kb in size.

ECCKiila-generated Ed25519

- ECCKiila: Field arithmetic $+$ EC arithmetic.
- SUPERCOP: Ed25519 arithmetic $\rightarrow$ Point decompression, multiply and add, and modular reduction by the order of the base point.

- `ecckiila-no-precomp`
  - Made for 32-bits architectures (uses 10-limb integers).
  - Minimal precomputed table ~ 2.5 KB.
  - Uses constant-time variable-point scalar multiplication with regular-NAF.
  - Uses double-scalar multiplication only for verification.
  - Roughly 40 kb in size.

- `ecckiila-precomp`
  - Made for both 32-bits and 64-bits architectures.
  - Precomputed table ~ 30 KB.
  - Constant-time comb method fixed-point scalar multiplication.
  - Double-scalar multiplication only for verification.
  - Roughly 70 kb in size.

Contents

Benchmarking

- SUPERCOP as benchmarking framework.
- Several implementations: `ref*`, `donna`, `monocypher`, `ecckiila-*`, `overoptimized`
- Compiled with `gcc 7.5`.
- Setup: Raspberry Pi 3B and Intel Xeon E5-1650 running Ubuntu 18.04 LTS.

Results - (59 bytes) Intel

Architecture	Implementation	Sign	Verify	KeyGen
x86_64	ref10	140 (□ base)	455 (□ base)	135 (□ base)
	ref	1560 (▽11.1x)	5218 (▽11.4x)	1531 (▽11.3x)
	amd64-64-24k	64 (▲2.18x)	225 (▲2.02x)	60 (▲2.25x)
	amd64-51-30k	66 (▲2.12x)	210 (▲2.16x)	62 (▲2.17x)
	donna	64 (▲2.18x)	217 (▲2.09x)	59 (▲2.28x)
	monocypher	230 (▽1.64x)	525 (▽1.15x)	210 (▽1.55x)
	overoptimized	264 (▽1.88x)	455 (▽1.00x)	227 (▽1.68x)
	ecckiila-precomp	101 (▲1.38x)	280 (▲1.62x)	96 (▲1.4x)
x86	ref10	399 (□ base)	1155 (□ base)	374 (□ base)
	ref	4137 (▽10.3x)	14105 (▽12.2x)	4086 (▽10.9x)
	amd64-64-24k	—	—	—
	amd64-51-30k	—	—	—
	donna	310 (▲1.28x)	962 (▲1.20x)	291 (▲1.28x)
	monocypher	533 (▽1.33x)	1347 (▽1.16x)	471 (▽1.25x)
	overoptimized	958 (▽2.40x)	1155 (▽1.00x)	914 (▽2.44x)
	ecckiila-no-precomp	1133 (▽2.83x)	1231 (▽1.06x)	1075 (▽2.87x)
	ecckiila-precomp	427 (▽1.07x)	1228 (▽1.06x)	368 (▲1.01x)

Comparison of timings on Intel architecture. □ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

Results - (59 bytes) ARM

Architecture	Implementation		Sign		Verify		KeyGen
aarch64	ref10		245 (☐ base)		688 (☐ base)		238 (☐ base)
	ref		2924 (▽11.9x)		9579 (▽13.9x)		2425 (▽10.1x)
	amd64-64-24k		—		—		—
	amd64-51-30k		—		—		—
	donna		196 (▲1.25x)		638 (▲1.07x)		162 (▲1.46x)
	monocypher		422 (▽1.72x)		812 (▽1.18x)		366 (▽1.53x)
	overoptimized		726 (▽2.96x)		688 (▽1.00x)		635 (▽2.66x)
	ecckiila-precomp		270 (▽1.10x)		808 (▽1.17x)		261 (▽1.09x)
armv7l	ref10		597 (☐ base)		1755 (☐ base)		582 (☐ base)
	ref		9933 (▽16.6x)		28642 (▽16.3x)		8442 (▽14.5x)
	amd64-64-24k		—		—		—
	amd64-51-30k		—		—		—
	donna		508 (▲1.17x)		1508 (▲1.16x)		495 (▲1.17x)
	monocypher		983 (▽1.64x)		2505 (▽1.42x)		987 (▽1.69x)
	overoptimized		1622 (▽2.71x)		1800 (▽1.02x)		1534 (▽2.63x)
	ecckiila-no-precomp		2134 (▽3.57x)		2237 (▽1.27x)		2050 (▽3.52x)
	ecckiila-precomp		815 (▽1.36x)		2213 (▽1.26x)		732 (▽1.25x)

Comparison of timings on ARM architecture. ☐ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

Contents

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Conclusions

- Cryptography engineering is hard — the devil is in the details.
- Computer-aided cryptographic tools help prevent common side-channel flaws.
- ECCKiila generates fast, secure, and portable code.
- Ed25519 reference implementations are around 10 years old.
- Some improvements are possible, e.g., GCD [2].
- Monocypher provides a small, secure, AND fast implementation for IoT devices.
- No partial leakage side-channel attacks on Ed25519 to date.

Thank you for listening.

All questions welcomed!

[1] Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. Set it and forget it! turnkey ECC for instant integration. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Au stin, TX, USA, 7-11 December, 2020*, pages 760–771. ACM, 2020. doi: 10.1145/3427228.3427291. URL `https://doi.org/10.1145/3427228.3427291`.

[2] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.

[3] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012. doi: 10.1007/s13389-012-0027-1. URL `https://doi.org/10.1007/s13389-012-0027-1`.