Two approaches to verifying high-speed ECC software

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org
April 29, 2017
“Cloudflare reported a carry bug in the P-256 implementation that they submitted for x86-64 in 7bacfc6. I can reproduce this via random testing against BoringSSL and, after applying the patch that they provided, can no longer do so, even after $2^{31}$ iterations.

This issue is not obviously exploitable, although we cannot rule out the possibility of someone managing to squeeze something through this hole. (It would be a cool paper.)”

—Adam Langley, Apr. 20, 2017
How to avoid cool papers

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org
April 29, 2017
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
- CVE-2015-3193 openssl: BN_mod_exp may produce incorrect results on x86_64
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
- CVE-2015-3193 openssl: BN_mod_exp may produce incorrect results on x86_64
- CVE-2014-3570 openssl: Bignum squaring may produce incorrect results
- CVE-2016-7055 openssl: Carry propagating bug in Montgomery multiplication
- CVE-2017-3732 openssl: BN_mod_exp may produce incorrect results on x86_64
- Ed25519 and X25519 "amd64-64" implementation (CHES 2011)
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
- CVE-2015-3193 openssl: BN_mod_exp may produce incorrect results on x86_64
- CVE-2014-3570 openssl: Bignum squaring may produce incorrect results
- CVE-2016-7055 openssl: Carry propagating bug in Montgomery multiplication
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
- CVE-2015-3193 openssl: BN_mod_exp may produce incorrect results on x86_64
- CVE-2014-3570 openssl: Bignum squaring may produce incorrect results
- CVE-2016-7055 openssl: Carry propagating bug in Montgomery multiplication
- CVE-2017-3732 openssl: BN_mod_exp may produce incorrect results on x86_64
Carry bugs

- Brumley, Barbosa, Page, Vercauteren, 2011: exploit carry bug in OpenSSL 0.9.8g
- 633 adaptive queries to obtain entire static ECDH key
- CVE-2015-3193 openssl: BN_mod_exp may produce incorrect results on x86_64
- CVE-2014-3570 openssl: Bignum squaring may produce incorrect results
- CVE-2016-7055 openssl: Carry propagating bug in Montgomery multiplication
- CVE-2017-3732 openssl: BN_mod_exp may produce incorrect results on x86_64
- Ed25519 and X25519 “amd64-64” implementation (CHES 2011)
• Bernstein 2006: X25519 Diffie-Hellman key exchange (originally: “Curve25519”)

• Secret keys: 32-byte little-endian scalars

• Public keys: 32-byte arrays, encoding $x$-coordinate of a point on

$$E : y^2 = x^3 + 486662x^2 + x$$

over $\mathbb{F}_{2^{255} - 19}$

• Base point: $(9, 0, \ldots, 0)$
Example for today: X25519

- Bernstein 2006: X25519 Diffie-Hellman key exchange (originally: “Curve25519”)
- Secret keys: 32-byte little-endian scalars
- Public keys: 32-byte arrays, encoding $x$-coordinate of a point on

$$E : y^2 = x^3 + 486662x^2 + x$$

over $\mathbb{F}_{2^{255}−19}$
- Base point: $(9, 0, \ldots, 0)$
- Given secret key $s$ and public key (or base point) $P$:
  - Copy $s$ to $s'$
  - Set least significant 3 bits of $s'$ to zero
  - Set most significant bit of $s'$ to zero
  - Set second-most significant bit of $s'$ to one
  - Compute $x$-coordinate of $s'P$
The Montgomery ladder

**Require:** A scalar \( 0 \leq k \in \mathbb{Z} \) and the \( x \)-coordinate \( x_P \) of some point \( P \)

**Ensure:** \( x_{kP} \)

\[
X_1 = x_P; \quad X_2 = 1; \quad Z_2 = 0; \quad X_3 = x_P; \quad Z_3 = 1
\]

for \( i \leftarrow n - 1 \) downto 0 do

if bit \( i \) of \( k \) is 1 then

\((X_3, Z_3, X_2, Z_2) \leftarrow \text{ladderstep}(X_1, X_3, Z_3, X_2, Z_2)\)

else

\((X_2, Z_2, X_3, Z_3) \leftarrow \text{ladderstep}(X_1, X_2, Z_2, X_3, Z_3)\)

end if

end for

return \( X_2 \cdot Z_2^{-1} \)
One Montgomery “ladder step”

\[ \text{const } a_{24} = (A + 2)/4 \text{ (A from the curve equation)} \]

\textbf{function} ladderstep(}X_{Q-P}, X_P, Z_P, X_Q, Z_Q) \text{) }

\begin{align*}
  t_1 & \leftarrow X_P + Z_P \\
  t_6 & \leftarrow t_1^2 \\
  t_2 & \leftarrow X_P - Z_P \\
  t_7 & \leftarrow t_2^2 \\
  t_5 & \leftarrow t_6 - t_7 \\
  t_3 & \leftarrow X_Q + Z_Q \\
  t_4 & \leftarrow X_Q - Z_Q \\
  t_8 & \leftarrow t_4 \cdot t_1 \\
  t_9 & \leftarrow t_3 \cdot t_2 \\
  X_{P+Q} & \leftarrow (t_8 + t_9)^2 \\
  Z_{P+Q} & \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2 \\
  X_{2P} & \leftarrow t_6 \cdot t_7 \\
  Z_{2P} & \leftarrow t_5 \cdot (t_7 + a_{24} \cdot t_5) \\
  \text{return } (X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q}) \\
\end{align*}

\textbf{end function}
Curve25519 implementations

- Bernstein, 2006: X25519 for various 32-bit Intel and AMD processors
- Gaudry, Thomé, 2007: X25519 for 64-bit Intel and AMD processors
- Costigan, Schwabe, 2009: X25519 for Cell Broadband Engine
- Bernstein, Duif, Lange, Schwabe, Yang, 2011: X25519 for Intel Nehalem/Westmire
- Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez, Schwabe, 2015: X25519 for AVR ATmega, TI MSP430, and ARM Cortex-M0
- Chou, 2015: The fastest Curve25519 software ever
- Many more implementations, most without scientific papers
Curve25519 implementations

- Bernstein, 2006: X25519 for various 32-bit Intel and AMD processors
- Gaudry, Thomé, 2007: X25519 for 64-bit Intel and AMD processors
- Costigan, Schwabe, 2009: X25519 for Cell Broadband Engine
- Bernstein, Duif, Lange, Schwabe, Yang, 2011: X25519 for Intel Nehalem/Westmere
- Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez, Schwabe, 2015: X25519 for AVR ATmega, TI MSP430, and ARM Cortex-M0
- Chou, 2015: The fastest Curve25519 software ever
- Many more implementations, most without scientific papers
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
Arithmetic in $\mathbb{F}_{2^{255}-19}$ for AMD64

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 $2^{51}$
- Can delay carry operations; carry “en bloc”
- Schoolbook multiplication now 25 64-bit integer multiplications
- Easy to merge multiplication with reduction (multiplies by 19)
- Better performance on Westmere/Nehalem, worse on 65 nm Core 2 and AMD processors
Bug in the radix-64 reduction

```
mulq crypto_sign_ed25519_amd64_64_38
add %rax,%r13
adc %rdx,%r14
adc $0,%r14
mov %r9,%rax
mulq crypto_sign_ed25519_amd64_64_38
add %rax,%r14
adc %rdx,%r15
adc $0,%r15
mov %r10,%rax
mulq crypto_sign_ed25519_amd64_64_38
add %rax,%r15
adc %rdx,%rbx
adc $0,%rbx
mov %r11,%rax
mulq crypto_sign_ed25519_amd64_64_38
add %rax,%rbx
mov $0,%rsi
adc %rdx,%rsi
```
Bug in the radix-64 reduction

```
(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r0 += mulrax
carry? r1 += mulrdx + carry
r1 += 0 + carry
mulrax = mulr5

(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r1 += mulrax
carry? r2 += mulrdx + carry
r2 += 0 + carry
mulrax = mulr6

(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r2 += mulrax
carry? r3 += mulrdx + carry
r3 += 0 + carry
mulrax = mulr7

(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r3 += mulrax
mulr4 = 0
mulr4 += mulrdx + carry
```
Bug in the radix-64 reduction

```
(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r0 += mulrax
carry? r1 += mulrdx + carry
r1 += 0 + carry
mulrax = mulr5
(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r1 += mulrax
carry? r2 += mulrdx + carry
r2 += 0 + carry
mulrax = mulr6
(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r2 += mulrax
carry? r3 += mulrdx + carry
r3 += 0 + carry
mulrax = mulr7
(uint128) mulrdx mulrax = mulrax * *(uint64 *)&crypto_sign_ed25519_amd64_64_38
carry? r3 += mulrax
mulr4 = 0
mulr4 += mulrdx + carry
```

Full software package contains 8912 lines of qhasm code!
Directions to correct crypto

Testing

• Is cheap, catches many bugs
• Does not conflict with performance
• Provides very high confidence in correctness for some crypto algorithms
• Typically fails to catch very rarely triggered bugs
Audits

- Expensive (time and/or money)
- Conflicts with performance
- Standard approach to ensure correctness and quality of crypto software
Formal verification

- Strongest guarantees of correctness
- Probably conflicts with performance
Directions to correct crypto

Formal verification

- Strongest guarantees of correctness
- Probably conflicts with performance
- Should focus on cases where tests fail
• C or assembly programmer adds high-level annotations
• More specifically, for example:
  • Limbs $a_0, \ldots, a_n$ compose a field element $A$
  • Limbs $b_0, \ldots, b_n$ compose a field element $B$
  • Limbs $r_0, \ldots, r_n$ compose a field element $R$
  • $R = A \cdot B$
Verification: the vision

• C or assembly programmer adds high-level annotations
• More specifically, for example:
  • Limbs $a_0, \ldots, a_n$ compose a field element $A$
  • Limbs $b_0, \ldots, b_n$ compose a field element $B$
  • Limbs $r_0, \ldots, r_n$ compose a field element $R$
  • $R = A \cdot B$
• Annotated code gets fed to verification tool
• Verification ensures that operation on limbs corresponds to high-level arithmetic
• Audits look at high-level annotations
Verification: the vision

• C or assembly programmer adds high-level annotations

• More specifically, for example:
  • Limbs $a_0, \ldots, a_n$ compose a field element $A$
  • Limbs $b_0, \ldots, b_n$ compose a field element $B$
  • Limbs $r_0, \ldots, r_n$ compose a field element $R$
  • $R = A \cdot B$

• Annotated code gets fed to verification tool

• Verification ensures that operation on limbs corresponds to high-level arithmetic

• Audits look at high-level annotations

• Even better: feed to even higher level verification

• Verify that the sequence of field operations accomplishes EC arithmetic
Verification approach I

Joint work with Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang.
Verification approach I

Joint work with Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang.

- Nehalem Curve25519 software is written in \texttt{qhasm}
- \texttt{qhasm} is a portable assembly language by Bernstein
Verification approach I

Joint work with Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang.

- Nehalem Curve25519 software is written in qhasm
- qhasm is a portable assembly language by Bernstein
- Idea for verification:
  - Annotate qhasm code
  - Translate annotated qhasm automatically to SMT-solver boolector
  - Use boolector to verify software
Verification approach I

Joint work with Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang.

- Nehalem Curve25519 software is written in qhasm
- qhasm is a portable assembly language by Bernstein
- Idea for verification:
  - Annotate qhasm code
  - Translate annotated qhasm automatically to SMT-solver boolector
  - Use boolector to verify software
- Verification target: Montgomery ladder step of X25519:
  - 5 multiplications in $\mathbb{F}_{2^{255}-19}$
  - 4 squarings in $\mathbb{F}_{2^{255}-19}$
  - 1 multiplication by 121666
  - Several additions and subtractions
// assume 0 \leq u x0, x1, x2, x3, x4 \leq u 2^{51} + 2^{15}
// assume 0 \leq u y0, y1, y2, y3, y4 \leq u 2^{51} + 2^{15}
r0 = x0
r1 = x1
r2 = x2
r3 = x3
r4 = x4
r0 += y0
r1 += y1
r2 += y2
r3 += y3
r4 += y4

// var sum_x = x0@u320 + x1@u320 \times 2^{51} + x2@u320 \times 2^{102} \\
// + x3@u320 \times 2^{153} + x4@u320 \times 2^{204}
// var sum_y = y0@u320 + y1@u320 \times 2^{51} + y2@u320 \times 2^{102} \\
// + y3@u320 \times 2^{153} + y4@u320 \times 2^{204}
// var sum_r = r0@u320 + r1@u320 \times 2^{51} + r2@u320 \times 2^{102} \\
// + r3@u320 \times 2^{153} + r4@u320 \times 2^{204}
// assert (sum_r - (sum_x + sum_y)) \% (2^{255} - 19) = 0 &&
// 0 \leq u r0, r1, r2, r3, r4 < u 2^{53}
How about multiplication?

- Again, express input field elements and output field elements
- Again, express bounds on the “limb size”
- Again, express algebraic relation of a modular multiplication
- Overall slightly more annotations for an auditor to look at
How about multiplication?

- Again, express input field elements and output field elements
- Again, express bounds on the “limb size”
- Again, express algebraic relation of a modular multiplication
- Overall slightly more annotations for an auditor to look at
- *Huge amount* of intermediate annotations
- SMT solver cannot simply verify from inputs to outputs
How about multiplication?

- Again, express input field elements and output field elements
- Again, express bounds on the “limb size”
- Again, express algebraic relation of a modular multiplication
- Overall slightly more annotations for an auditor to look at
- *Huge amount* of intermediate annotations
- SMT solver cannot simply verify from inputs to outputs
- Overall:
  - 217 lines of qhasm, including variable declarations
  - 589 lines of annotations
How about multiplication?

- Again, express input field elements and output field elements
- Again, express bounds on the “limb size”
- Again, express algebraic relation of a modular multiplication
- Overall slightly more annotations for an auditor to look at
- *Huge amount* of intermediate annotations
- SMT solver cannot simply verify from inputs to outputs
- Overall:
  - 217 lines of `qhasm`, including variable declarations
  - 589 lines of annotations
- Large amount of manual work on top of assembly optimization
- Writing verifiable code requires expert knowledge from two domains!
- Verification of just multiplication takes $> 90$ hours
Overall results

- Formally verified Montgomery ladderstep
  - “Redundant” radix-$2^{51}$ representation
  - Non-redundant radix-$2^{64}$ representation
  - Reproduced bug in original version of the software
- Most verification used automatic qhasm $\rightarrow$ boolector translation
- Tiny bit of code in radix-$2^{64}$ needed proof assistant Coq
Verification approach II

- 2 problems with SMT approach:
  - Huge amount of (manual) annotations
  - Long verification time

- Idea: automagically translate to input for computer-algebra system
- Accept failures to prove correctness
- Work in progress with Bernstein
- Annotate C code (later: also qhasm)
- (Currently) use C++ compiler and operator overloading to
  - Keep track of computation graph
  - Keep track of worst-case ranges of limbs
- Output polynomial relations to Sage
- Use Sage to verify correctness of C code
Verification approach II

- 2 problems with SMT approach:
  - Huge amount of (manual) annotations
  - Long verification time
- Idea: automagically translate to input for computer-algebra system
- Accept failures to prove correctness
Verification approach II

- 2 problems with SMT approach:
  - Huge amount of (manual) annotations
  - Long verification time
- Idea: automagically translate to input for computer-algebra system
- Accept failures to prove correctness

Work in progress with Bernstein

- Annotate C code (later: also qhasm)
- (Currently) use C++ compiler and operator overloading to
  - Keep track of computation graph
  - Keep track of worst-case ranges of limbs
  - Output polynomial relations to Sage
  - Use Sage to verify correctness of C code
crypto_int32 f[10];
crypto_int32 g[10];
crypto_int32 h[10];

verifier_bigint vf;
verifier_addlimbs_10_255(&vf,f);
verifier_bigint vg;
verifier_addlimbs_10_255(&vg,g);

fe_add(h,f,g);

verifier_bigint vh;
verifier_addlimbs_10_255(&vh,h);
verifier_assertsum(&vh,&vf,&vg);
Example: multiplication

crypto_int32 f[10];
crypto_int32 g[10];
crypto_int32 h[10];

verifier_bigint vf;
verifier_addlimbs_10_255(&vf,f);
verifier_bigint vg;
verifier_addlimbs_10_255(&vg,g);

fe_mul(h,f,g);

verifier_bigint vh;
verifier_addlimbs_10_255(&vh,h);
verifier_assertprodmod(&vh,&vf,&vg,"2^255-19");
• Consider computation of $x^{2^{100}}$ in $\mathbb{F}_{2^{127}}$.
• Input is little-endian byte array.
• Convert to internal representation in radix $2^{26}$.
A small demo

- Consider computation of $x^{2^{100}}$ in $\mathbb{F}_{2^{127}} - 1$
- Input is little-endian byte array
- Convert to internal representation in radix $2^{26}$
- Verify a single squaring
A small demo

- Consider computation of $x^{2^{100}}$ in $\mathbb{F}_{2^{127}-1}$
- Input is little-endian byte array
- Convert to internal representation in radix $2^{26}$
- Verify a single squaring
- Put a loop around it
A small demo

- Consider computation of $x^{2^{100}}$ in $\mathbb{F}_{2^{127} - 1}$
- Input is little-endian byte array
- Convert to internal representation in radix $2^{26}$
- Verify a single squaring
- Put a loop around it
- Still too slow for big chunks of code
  - Problem: verification always goes back to the beginning
  - Idea: Declare that we trust already verified results
  - This is known as “cutting” the verification
Let’s “cut some limbs”
Let’s call it a draw
fe_sub(tmp0,x3,z3);

  verifier_bigint D;
  verifier_limbs_10_255(&D,tmp0);
  verifier_assertdiff(&D,&X3,&Z3);
  verifier_cutlimbs(tmp0,10);

fe_sub(tmp1,x2,z2);

  verifier_bigint B;
  verifier_limbs_10_255(&B,tmp1);
  verifier_assertdiff(&B,&X2,&Z2);
  verifier_cutlimbs(tmp1,10);

fe_add(x2,x2,z2);

  verifier_bigint A;
  verifier_limbs_10_255(&A,x2);
  verifier_assertsum(&A,&X2,&Z2);
  verifier_cutlimbs(x2,10);

fe_add(z2,x3,z3);

  verifier_bigint C;
  verifier_limbs_10_255(&C,z2);
  verifier_assertsum(&C,&X3,&Z3);
  verifier_cutlimbs(z2,10);

fe_mul(z3,tmp0,x2);

  verifier_bigint DA;
  verifier_limbs_10_255(&DA,z3);
  verifier_assertprodmod(&DA,&D,&A,vp);
  verifier_cutlimbs(z3,10);

fe_mul(z2,z2,tmp1);

  verifier_bigint CB;
  verifier_limbs_10_255(&CB,z2);
  verifier_assertprodmod(&CB,&C,&B,vp);
  verifier_cutlimbs(z2,10);
• Input conversion from byte array (see $\mathbb{F}_{2^{127} - 1}$ example)
Beyond the ladder step

- Input conversion from byte array (see $\mathbb{F}_{2^{127}-1}$ example)
- “Clamping” of scalar: currently not covered
Beyond the ladder step

- Input conversion from byte array (see $\mathbb{F}_{2^{127} - 1}$ example)
- “Clamping” of scalar: currently not covered
- Final inversion: exponentiation by $p - 2$
Beyond the ladder step

- Input conversion from byte array (see $\mathbb{F}_{2^{127} - 1}$ example)
- “Clamping” of scalar: currently not covered
- Final inversion: exponentiation by $p - 2$
- “Freezing” of elements:
  - Carry to produce result in $\{0, \ldots, 2^{255} - 1\}$
  - Conditionally subtract $p$
  - Use fork to verify both cases
# feed through: ./unroll x1 n

\[ p = 2^{**255}-19 \]

\[ A = 486662 \]

\[ x_2, z_2, x_3, z_3 = 1, 0, x_1, 1 \]

for i in reversed(range(255)):
    ni = bit(n,i)
    x2, x3 = cswap(x2, x3, ni)
    z2, z3 = cswap(z2, z3, ni)
    x3, z3 = \((4*(x_2*x_3-z_2*z_3)**2, 4*x_1*(x_2*z_3-z_2*x_3)**2)\)
    x2, z2 = \((x_2**2-z_2**2)**2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2)**2)\)
    x3, z3 = \((x_3%p, z_3%p)\)
    x2, z2 = \((x_2%p, z_2%p)\)
    cut(x2)
    cut(x3)
    cut(z2)
    cut(z3)
    x2, x3 = cswap(x2, x3, ni)
    z2, z3 = cswap(z2, z3, ni)
    cut(x2)
    cut(z2)
return x2*pow(z2,p-2,p)
First results and TODOs

Results

• Verification of modular multiplication in a few seconds
• Verification of full X25519 Montgomery ladder in \( \approx 1:10 \) minutes
• Verification against high-level code

TODOs

• Support assembly/qhasm
• Get rid of C++ compiler
• Support “non-redundant” arithmetic
• Support window methods
• Test, test, test
First results and TODOs

Results

• Verification of modular multiplication in a few seconds
• Verification of full X25519 Montgomery ladder in \( \approx 1:10 \) minutes
• Verification against high-level code

TODOs

• Support assembly/qhasm
• Get rid of C++ compiler
• Support “non-redundant” arithmetic
• Support window methods
• Test, test, test
Papers and Software

  https://cryptojedi.org/papers/#verify25519

- Many X25519 implementations in SUPERCOP (crypto_scalarmult/curve25519)
  https://bench.cr.yp.to/supercop.html

- Verification using boolector:
  https://cryptojedi.org/crypto/#verify25519

- Verification with gfverif:
  https://cryptojedi.org/crypto/#gfverif