Verifying crypto

Many questions and the beginning of an answer

Peter Schwabe

Radboud University Nijmegen, The Netherlands

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

May 20, 2014

Brouwer Seminar

About me

- **2001-2006:** Studies of computer science at RWTH Aachen (Germany)
- **2006-2007:** Ph.D. student at RWTH Aachen
- **2008-2011:** Ph.D. student at TU Eindhoven
- **2011-2012:** Postdoc at Academia Sincia (Taiwan) and National Taiwan University
- **Since 2013:** UD in the Digital Security Group
- **Since 2014:** Work on VENI project "High-speed high-security cryptography"

Research topics

During Ph.D. time

- High-speed cryptography
  - Optimizing the Advanced Encryption Standard (AES)
  - Elliptic-curve cryptography (ECC)
  - Cryptographic pairings
  - NaCl (http://nacl.cr.yp.to)
- High-speed cryptanalysis
  - Attacking ECC (parallel Pollard rho algorithm)
  - Attacking code-based crypto (generalized birthday attack)

Research topics

During Ph.D. time

- High-speed cryptography
  - Optimizing the Advanced Encryption Standard (AES)
  - Elliptic-curve cryptography (ECC)
  - Cryptographic pairings
  - NaCl (http://nacl.cr.yp.to)
- High-speed cryptanalysis
  - Attacking ECC (parallel Pollard rho algorithm)
  - Attacking code-based crypto (generalized birthday attack)

As a Postdoc

- Focus on constructive side (NaCl)
- Starting to look into automated optimization

VENI project
"High-speed high-security crypto"

NaCl for embedded microcontrollers

- Very restricted environment (speed, memory, storage)
- Typically exposed to physical attacks

VENI project
"High-speed high-security crypto"

NaCl for embedded microcontrollers

- Very restricted environment (speed, memory, storage)
- Typically exposed to physical attacks

A finite-field compiler

- ECC needs operations in large finite fields
- Idea: compile sequence of field operations to superfast assembly

VENI project
"High-speed high-security crypto"

NaCl for embedded microcontrollers

- Very restricted environment (speed, memory, storage)
- Typically exposed to physical attacks

A finite-field compiler

- ECC needs operations in large finite fields
- Idea: compile sequence of field operations to superfast assembly

Verification of crypto software

- Started in the context of the finite-field compiler
- Generally important: ensure correctness of crypto software
- Additional: ensure security of crypto software
- Verification on the assembly level

High-speed crypto

- Crypto algorithms are typically small in software
- Example: AES, just a few lines of C
- Executed very often (AES encrypts terabytes each day)

High-speed crypto

- Crypto algorithms are typically small in software
- Example: AES, just a few lines of C
- Executed very often (AES encrypts terabytes each day)
- Crypto needs to work fast on busy servers
- Crypto needs to work fast on small embedded devices

High-speed crypto

- Crypto algorithms are typically small in software
- Example: AES, just a few lines of C
- Executed very often (AES encrypts terabytes each day)
- Crypto needs to work fast on busy servers
- Crypto needs to work fast on small embedded devices
- Serious optimization is feasible and worth the effort
- Typical high-speed crypto:
  - Optimize on the assembly level
  - Use instruction set to an extent that C does not allow
  - Inline, unroll, ...

High-speed crypto

- Crypto algorithms are typically small in software
- Example: AES, just a few lines of C
- Executed very often (AES encrypts terabytes each day)
- Crypto needs to work fast on busy servers
- Crypto needs to work fast on small embedded devices
- Serious optimization is feasible and worth the effort
- Typical high-speed crypto:
  - Optimize on the assembly level
  - Use instruction set to an extent that C does not allow
  - Inline, unroll, ...
- 10% speedup are typically a paper!

High-security crypto

- Best known attacks take $\geq 2^{128}$ operations
- Attacks have been extensively studied

High-security crypto

- Best known attacks take $\geq 2^{128}$ operations
- Attacks have been extensively studied
- Implementations must not leak secret information
  - Execution time must be independent of secret data

High-security crypto

- Best known attacks take $\geq 2^{128}$ operations
- Attacks have been extensively studied
- Implementations must not leak secret information
  - Execution time must be independent of secret data
  - No data flow from secrets into branch conditions
  - No data flow from secrets into load/store addresses

High-security crypto

- Best known attacks take $\geq 2^{128}$ operations
- Attacks have been extensively studied
- Implementations must not leak secret information
  - Execution time must be independent of secret data
  - No data flow from secrets into branch conditions
  - No data flow from secrets into load/store addresses
  - *Timing attacks* are practical and efficient

High-security crypto

- Best known attacks take $\geq 2^{128}$ operations
- Attacks have been extensively studied
- Implementations must not leak secret information
  - Execution time must be independent of secret data
  - No data flow from secrets into branch conditions
  - No data flow from secrets into load/store addresses
  - *Timing attacks* are practical and efficient
- Implementations must be correct (bug attacks!)

Correct crypto?

"Are you actually sure that your implementations are correct?"
—Gerhard Woeginger, Jan. 24, 2011.

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

Correct crypto?

Audits

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

Correct crypto?

Formal verification

- Strongest guarantees of correctness
- Probably conflicts with performance

Correct crypto?

Formal verification

- Strongest guarantees of correctness
- Probably conflicts with performance
- **Should focus on cases where test and audits fail**

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$
- Points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ on E together with a "point at infinity" form a group $E(\mathbb{F}_q)$

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$
- Points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ on E together with a "point at infinity" form a group $E(\mathbb{F}_q)$
- Group addition can be computed with a few operations in $\mathbb{F}_q$

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$
- Points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ on E together with a "point at infinity" form a group $E(\mathbb{F}_q)$
- Group addition can be computed with a few operations in $\mathbb{F}_q$
- For $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}$, computing kP is easy ($\Theta(\log(k))$)

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

  $$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$
- Points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ on E together with a "point at infinity" form a group $E(\mathbb{F}_q)$
- Group addition can be computed with a few operations in $\mathbb{F}_q$
- For $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}$, computing kP is easy ($\Theta(\log(k))$)
- Given $Q \in \langle P \rangle$ and P, computing k with $kP = Q$ is hard ($\Theta(\sqrt{k})$)

Elliptic-curve cryptography

- Let $\mathbb{F}_q$ be a finite field
- For $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$, an equation of the form

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

  defines an elliptic curve E over $\mathbb{F}_q$
- Points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ on E together with a "point at infinity" form a group $E(\mathbb{F}_q)$
- Group addition can be computed with a few operations in $\mathbb{F}_q$
- For $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}$, computing kP is easy ($\Theta(\log(k))$)
- Given $Q \in \langle P \rangle$ and P, computing k with $kP = Q$ is hard ($\Theta(\sqrt{k})$)
- Use in crypto: choose random k, compute and publish kP

Curve25519 ECDH

- Diffie-Hellman key exchange protocol by Bernstein (2006)
- Uses curve $E : y^2 = x^3 + 486662x^2 + x$ defined over $\mathbb{F}_{2^{255}-19}$
- Conservative parameter choice, targeting high security
- Set speed records on a variety of platforms

Curve25519 ECDH

- Diffie-Hellman key exchange protocol by Bernstein (2006)
- Uses curve $E : y^2 = x^3 + 486662x^2 + x$ defined over $\mathbb{F}_{2^{255}-19}$
- Conservative parameter choice, targeting high security
- Set speed records on a variety of platforms
- High-level view:
  - Input: x-coordinate x_P of a point P, scalar k
  - Compute x-coordinate x_{kP} of kP as $x_{kP} = X_{kP}/Z_{kP}$
  - Invert Z_{kP}, multiply by X_{kP} to obtain x_{kP}
  - Inputs and outputs encoded as little-endian byte arrays of length 32

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}, Z_{kP}) fulfilling $x_{kP} = X_{kP}/Z_{kP}$

$X_1 = x_P$; $X_2 = 1$; $Z_2 = 0$; $X_3 = x_P$; $Z_3 = 1$
for $i \leftarrow n-1$ downto 0 **do**
  if bit i of k is 1 **then**
    $(X3, Z3, X2, Z2) \leftarrow$ ladderstep$(X1, X3, Z3, X2, Z2)$
  else
    $(X2, Z2, X3, Z3) \leftarrow$ ladderstep$(X1, X2, Z2, X3, Z3)$
  end if
end for
return (X_2, Z_2)

One Montgomery "ladder step"

const $a24 = 121666$ (from the curve equation)
function ladderstep($X_{Q-P}, X_P, Z_P, X_Q, Z_Q$)

$t_1 \leftarrow X_P + Z_P$
$t_6 \leftarrow t_1^2$
$t_2 \leftarrow X_P - Z_P$
$t_7 \leftarrow t_2^2$
$t_5 \leftarrow t_6 - t_7$
$t_3 \leftarrow X_Q + Z_Q$
$t_4 \leftarrow X_Q - Z_Q$
$t_8 \leftarrow t_4 \cdot t_1$
$t_9 \leftarrow t_3 \cdot t_2$
$X_{P+Q} \leftarrow (t_8 + t_9)^2$
$Z_{P+Q} \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2$
$X_{2P} \leftarrow t_6 \cdot t_7$
$Z_{2P} \leftarrow t_5 \cdot (t_7 + a24 \cdot t_5)$
return $(X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q})$

end function

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

- Need arithmetic on 255-bit integers and reduction mod $2^{255} - 19$
- Speed typically determined by speed of multiplications
- Use fastest hardware multiplier
- On Intel Nehalem: $64 \times 64 \rightarrow 128$-bit integer multiply

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

- Need arithmetic on 255-bit integers and reduction mod $2^{255} - 19$
- Speed typically determined by speed of multiplications
- Use fastest hardware multiplier
- On Intel Nehalem: $64 \times 64 \rightarrow 128$-bit integer multiply
- Represent 256-bit integer A through 4 64-bit integers a_0, a_1, a_2, a_3
- Value of A is $\sum_{i=0}^{3} a_i 2^{64 \cdot i}$

```
typedef struct{
  unsigned long long v[4];
} fe25519;
```

Addition

```
int64 r0
int64 r1
int64 r2
int64 r3
int64 t0
int64 t1

enter fe25519_add

r0 = mem64[input_1 +  0]
r1 = mem64[input_1 +  8]
r2 = mem64[input_1 + 16]
r3 = mem64[input_1 + 24]

carry? r0 += mem64[input_2 +  0]
carry? r1 += mem64[input_2 +  8] + carry
carry? r2 += mem64[input_2 + 16] + carry
carry? r3 += mem64[input_2 + 24] + carry
```

```
t0 = 0
t1 = 38
t1 = t0 if !carry

carry? r0 += t1
carry? r1 += t0 + carry
carry? r2 += t0 + carry
carry? r3 += t0 + carry

t0 = t1 if carry
r0 += t0

mem64[input_0 +  0] = r0
mem64[input_0 +  8] = r1
mem64[input_0 + 16] = r2
mem64[input_0 + 24] = r3

return
```

Multiplication

```
x0 = mem64[input_1 + 0]
rax = mem64[input_2 + 0]
(uint128) rdx rax = rax * x0
r0 = rax
r1 = rdx

rax = mem64[input_2 + 8]
(uint128) rdx rax = rax * x0
carry? r1 += rax
r2 = 0
r2 += rdx + carry
```

```
rax = mem64[input_2 + 16]
(uint128) rdx rax = rax * x0
carry? r2 += rax
r3 = 0
r3 += rdx + carry

rax = mem64[input_2 + 24]
(uint128) rdx rax = rax * x0
carry? r3 += rax
r4 = 0
r4 += rdx + carry
```

Multiplication

```
x1 = mem64[input_1 + 8]
rax = mem64[input_2 + 0]
(uint128) rdx rax = rax * x1
carry? r1 += rax
c = 0
c += rdx + carry

rax = mem64[input_2 + 8]
(uint128) rdx rax = rax * x1
carry? r2 += rax
rdx += 0 + carry
carry? r2 += c
c = 0
c += rdx + carry
```

```
rax = mem64[input_2 + 16]
(uint128) rdx rax = rax * x1
carry? r3 += rax
rdx += 0 + carry
carry? r3 += c
c = 0
c += rdx + carry

rax = mem64[input_2 + 24]
(uint128) rdx rax = rax * x1
carry? r4 += rax
rdx += 0 + carry
carry? r4 += c
r5 = 0
r5 += rdx + carry

...
```

Multiplication

```
x3 = mem64[input_1 + 24]
rax = mem64[input_2 + 0]
(uint128) rdx rax = rax * x3
carry? r3 += rax
c = 0
c += rdx + carry

rax = mem64[input_2 + 8]
(uint128) rdx rax = rax * x3
carry? r4 += rax
rdx += 0 + carry
carry? r4 += c
c = 0
c += rdx + carry
```

```
rax = mem64[input_2 + 16]
(uint128) rdx rax = rax * x3
carry? r5 += rax
rdx += 0 + carry
carry? r5 += c
c = 0
c += rdx + carry

rax = mem64[input_2 + 24]
(uint128) rdx rax = rax * x3
carry? r6 += rax
rdx += 0 + carry
carry? r6 += c
r7 = 0
r7 += rdx + carry
```

Reduction mod $2^{255} - 19$

- "Lazy" reduction modulo $2^{256} - 38$: multiply upper half by 38, add to lower half

Reduction mod $2^{255} - 19$

- "Lazy" reduction modulo $2^{256} - 38$: multiply upper half by 38, add to lower half
- In assembly:

```
rax = r4
(uint128) rdx rax = rax * mem64[&const_38]
r4 = rax
rax = r5
r5 = rdx
(uint128) rdx rax = rax * mem64[&const_38]
carry? r5 += rax
rax = r6
r6 = 0
r6 += rdx + carry
...

(uint128) rdx rax = rax * mem64[&const_38]
carry? r7 += rax
r8 = 0
r8 += rdx + carry
```

Reduction mod $2^{255} - 19$

- "Lazy" reduction modulo $2^{256} - 38$: multiply upper half by 38, add to lower half
- In assembly:

```
carry? r0 += r4
carry? r1 += r5 + carry
carry? r2 += r6 + carry
carry? r3 += r7 + carry

zero = 0
r8 += zero + carry
r8 *= 38
carry? r0 += r8
carry? r1 += zero + carry
carry? r2 += zero + carry
carry? r3 += zero + carry
zero += zero + carry
zero *= 38
r0 += zero
```

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$$

- This is called radix-2^{51} representation

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$$

- This is called radix-2^{51} representation
- Multiple ways to write the same integer A, for example $A = 2^{52}$:
  - $(2^{52}, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0)$

Changing the radix

- Radix-2^{64} representation works and is sometimes a good choice
- Highly depends on the efficiency of handling carries
- Example: Intel Nehalem can do 3 additions every cycle, but only 1 addition with carry every two cycles (carries cost a factor of 6!)
- Let's get rid of the carries, represent A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{51 \cdot i}$$

- This is called radix-2^{51} representation
- Multiple ways to write the same integer A, for example $A = 2^{52}$:
  - $(2^{52}, 0, 0, 0, 0)$
  - $(0, 2, 0, 0, 0)$
- Call a representation $(a_0, a_1, a_2, a_3, a_4)$ *reduced*, if all $a_i \in [0, \dots, 2^{52} - 1]$

Addition

```
enter fe25519_add
r0 = mem64[input_1 +  0]
r1 = mem64[input_1 +  8]
r2 = mem64[input_1 + 16]
r3 = mem64[input_1 + 24]
r4 = mem64[input_1 + 32]

r0 += mem64[input_2 +  0]
r1 += mem64[input_2 +  8]
r2 += mem64[input_2 + 16]
r3 += mem64[input_2 + 24]
r4 += mem64[input_2 + 32]

mem64[input_0 +  0] = r0
mem64[input_0 +  8] = r1
mem64[input_0 + 16] = r2
mem64[input_0 + 24] = r3
mem64[input_0 + 32] = r4
return
```

Multiplication

```
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 0]
r0 = rax
r0h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 8]
r1 = rax
r1h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 16]
r2 = rax
r2h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 24]
r3 = rax
r3h = rdx
rax = mem64[input_1 + 0]
(int128) rdx rax = rax * mem64[input_2 + 32]
r4 = rax
r4h = rdx
```

Multiplication

```
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 0]
carry? r1 += rax
r1h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 8]
carry? r2 += rax
r2h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 16]
carry? r3 += rax
r3h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 24]
carry? r4 += rax
r4h += rdx + carry
rax = mem64[input_1 + 8]
(int128) rdx rax = rax * mem64[input_2 + 32]
r5 = rax
r5h = rdx
```

Multiplication

```
...

mem64[input_0 +  0] = r0
mem64[input_0 +  8] = r0h
mem64[input_0 + 16] = r1
mem64[input_0 + 24] = r1h
mem64[input_0 + 32] = r2
mem64[input_0 + 40] = r2h

...

mem64[input_0 + 128] = r8
mem64[input_0 + 136] = r8h
```

Reduction mod p

- We now have $r_0, \ldots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \ldots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

Reduction mod p

- We now have $r_0, \ldots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \ldots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255} - 19}$$

- We can reduce modulo p as

  $r_0 \leftarrow r_0 + 19 r_5$

Reduction mod p

- We now have $r_0, \ldots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \ldots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

- We can reduce modulo p as

$$\begin{aligned}
r_0 &\leftarrow r_0 + 19r_5\\
r_1 &\leftarrow r_1 + 19r_6\\
r_2 &\leftarrow r_2 + 19r_7\\
r_3 &\leftarrow r_3 + 19r_8
\end{aligned}$$

Reduction mod p

- We now have $r_0, \ldots, r_8$, such that

$$\sum_{i=0}^{8} r_i X^i = \left(\sum_{i=0}^{4} a_i X^i\right)\left(\sum_{i=0}^{4} b_i X^i\right)$$

- We want to have $r_0, \ldots, r_4$, such that

$$\sum_{i=0}^{4} r_i 2^{51\cdot i} \equiv \left(\sum_{i=0}^{4} a_i 2^{51\cdot i}\right)\left(\sum_{i=0}^{4} b_i 2^{51\cdot i}\right) \pmod{2^{255}-19}$$

- We can reduce modulo p as

$$r_0 \leftarrow r_0 + 19r_5$$
$$r_1 \leftarrow r_1 + 19r_6$$
$$r_2 \leftarrow r_2 + 19r_7$$
$$r_3 \leftarrow r_3 + 19r_8$$

- Can even merge this reduction with multiplication:
  - Precompute $19a_1, 19a_2, 19a_3, 19a_4$
  - Multiply b_j by $19a_i$ if $i + j > 4$

Carrying after multiplication

- Coefficients r_i are way too large
- Need to carry. In pseudocode:

```
carry = (r0h.r0) >> 51
(r1h.r1) += carry
carry <<= 51
(r0h.r0) -= carry
```

Carrying after multiplication

- Coefficients r_i are way too large
- Need to carry. In pseudocode:

```
carry = (r0h.r0) >> 51
(r1h.r1) += carry
carry <<= 51
(r0h.r0) -= carry
```

- Carry from r_0 to r_1; from r_1 to r_2, and so on
- Multiply carry from r_4 by 19 and add to r_0

Carrying after multiplication

- Coefficients r_i are way too large
- Need to carry. In pseudocode:

```
carry = (r0h.r0) >> 51
(r1h.r1) += carry
carry <<= 51
(r0h.r0) -= carry
```

- Carry from r_0 to r_1; from r_1 to r_2, and so on
- Multiply carry from r_4 by 19 and add to r_0
- After one round of carries we have signed 64-bit integers
- Perform another round of carries to obtain reduced coefficients

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

Nice for formal verification

- Code is completely branch-free
- Can even write down branch-free Montgomery ladder (unrolling)

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

Nice for formal verification

- Code is completely branch-free
- Can even write down branch-free Montgomery ladder (unrolling)
- No dynamic memory allocations
- No function calls
- No side effects (except for flags)

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

Nice for formal verification

- Code is completely branch-free
- Can even write down branch-free Montgomery ladder (unrolling)
- No dynamic memory allocations
- No function calls
- No side effects (except for flags)
- "abnormally straight line code" —Adam Langley

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

. . . not so nice

- Only very high-level and very low-level description
  - Pseudocode – sequence of operations in $\mathbb{F}_{2^{255}-19}$
  - Hand-optimized assembly (2 versions with different radices)

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

. . . not so nice

- Only very high-level and very low-level description
  - Pseudocode – sequence of operations in $\mathbb{F}_{2^{255}-19}$
  - Hand-optimized assembly (2 versions with different radices)
- Non-linear operations on non-native data types

Ladderstep observations

Ladderstep

- Two versions, fully inlined sequence of $\mathbb{F}_{2^{255}-19}$ operations:
  - One using radix-2^{64} representation
  - One using radix-2^{51} representation

... not so nice

- Only very high-level and very low-level description
  - Pseudocode – sequence of operations in $\mathbb{F}_{2^{255}-19}$
  - Hand-optimized assembly (2 versions with different radices)
- Non-linear operations on non-native data types
- 1419 LOC in radix 2^{64}
- 1533 LOC in radix 2^{51}

Assembly?

- The code I showed you is not native assembly
- It's qhasm code:
  - High-level ("portable") assembler by Bernstein
  - Unified syntax across architectures
  - Efficient register allocation (linear-scan like)
  - All freedom of assembly but faster development time

Annotated qhasm

Idea for proof of correctness

- Annotate qhasm code with pre- and post-conditions
- Automatically translate to boolector
- Use `boolector -minisat` to prove correctness

Annotated qhasm

Idea for proof of correctness

- Annotate qhasm code with pre- and post-conditions
- Automatically translate to boolector
- Use `boolector -minisat` to prove correctness

Experience so far

- Don't verify ladderstep "en bloc", chop in pieces, use composition of Hoare logic

Annotated qhasm

Idea for proof of correctness

- Annotate qhasm code with pre- and post-conditions
- Automatically translate to boolector
- Use `boolector -minisat` to prove correctness

Experience so far

- Don't verify ladderstep "en bloc", chop in pieces, use composition of Hoare logic
- Extensive annotation needed, in particular for multiplication

Annotated qhasm

Idea for proof of correctness

- Annotate qhasm code with pre- and post-conditions
- Automatically translate to boolector
- Use `boolector -minisat` to prove correctness

Experience so far

- Don't verify ladderstep "en bloc", chop in pieces, use composition of Hoare logic
- Extensive annotation needed, in particular for multiplication
- Carries cause trouble (verification of radix-2^{51} implementation is easier)

Annotated qhasm

Idea for proof of correctness

- Annotate qhasm code with pre- and post-conditions
- Automatically translate to boolector
- Use `boolector -minisat` to prove correctness

Experience so far

- Don't verify ladderstep "en bloc", chop in pieces, use composition of Hoare logic
- Extensive annotation needed, in particular for multiplication
- Carries cause trouble (verification of radix-2^{51} implementation is easier)
- Cannot prove everything with boolector, need 2 proofs in Coq (not automated)

Results

- Fully verified ladderstep (code matches annotations)

Results

- Fully verified ladderstep (code matches annotations)
- Most costly to verify: radix-2^{51} multiplication:
  - 27 intermediate conditions/annotations
  - 5658 minutes, ≈ 4 days
  - Out of this, 2723 minutes for delayed carry
  - Two-phase carry is only 264 minutes

Results

- Fully verified ladderstep (code matches annotations)
- Most costly to verify: radix-2^{51} multiplication:
  - 27 intermediate conditions/annotations
  - 5658 minutes, ≈ 4 days
  - Out of this, 2723 minutes for delayed carry
  - Two-phase carry is only 264 minutes
- Finding a known bug in early radix-2^{64} multiplication is fast: < 9 seconds

Questions

- Is annotated assembly/qhasm the right approach?

Questions

- Is annotated assembly/qhasm the right approach?
- Is translation to boolector the right approach?

Questions

- Is annotated assembly/qhasm the right approach?
- Is translation to boolector the right approach?
- How can we reduce the amount of annotations?
- How can we automate the whole process (incl. Coq)?

Questions

- Is annotated assembly/qhasm the right approach?
- Is translation to boolector the right approach?
- How can we reduce the amount of annotations?
- How can we automate the whole process (incl. Coq)?
- Will this scale to less friendly cases
  - Highly interleaved operations
  - Arithmetic using floats
  - Vector instructions

Questions

- Is annotated assembly/qhasm the right approach?
- Is translation to boolector the right approach?
- How can we reduce the amount of annotations?
- How can we automate the whole process (incl. Coq)?
- Will this scale to less friendly cases
  - Highly interleaved operations
  - Arithmetic using floats
  - Vector instructions
- How about proofs of timing-attack resistance?

Questions

- Is annotated assembly/qhasm the right approach?
- Is translation to boolector the right approach?
- How can we reduce the amount of annotations?
- How can we automate the whole process (incl. Coq)?
- Will this scale to less friendly cases
  - Highly interleaved operations
  - Arithmetic using floats
  - Vector instructions
- How about proofs of timing-attack resistance?
- Can we prove equivalence with a reference implementation?

An equivalent(?) Curve25519 implementation

TweetNaCl

- Joint work with Bernstein, Janssen, and Lange
- Re-implementation of NaCl in just 100 Tweets
- Aims at auditability
- Contains Curve25519, Ed25519 signatures, Salsa20 stream cipher, Poly1305 authenticator, SHA-512 hash
- All written in portable ISO C

An equivalent(?) Curve25519 implementation

TweetNaCl

- Joint work with Bernstein, Janssen, and Lange
- Re-implementation of NaCl in just 100 Tweets
- Aims at auditability
- Contains Curve25519, Ed25519 signatures, Salsa20 stream cipher, Poly1305 authenticator, SHA-512 hash
- All written in portable ISO C
- Curve25519 is $> 10\times$ slower on Ivy Bridge than speed-optimized software

An equivalent(?) Curve25519 implementation

TweetNaCl

- Joint work with Bernstein, Janssen, and Lange
- Re-implementation of NaCl in just 100 Tweets
- Aims at auditability
- Contains Curve25519, Ed25519 signatures, Salsa20 stream cipher, Poly1305 authenticator, SHA-512 hash
- All written in portable ISO C
- Curve25519 is $> 10\times$ slower on Ivy Bridge than speed-optimized software
- Code available at `http://tweetnacl.cr.yp.to`

Resources online

- Paper:
  http://cryptojedi.org/papers/#verify25519
- Translator, proofs:
  http://cryptojedi.org/crypto/#verify25519
- qhasm:
  http://cr.yp.to/qhasm.html