Signed Cryptographic Program Verification with Typed CryptoLine

Yu-Fu Fu[1], Jiaxiang Liu[2], Xiaomu Shi[2],
Ming-Hsien Tsai[1], Bow-Yaw Wang[1], Bo-Yin Yang[1]

[1] Academia Sinica

[2] Shenzhen University

ACM CCS 2019

Outline

Outline

Practical Cryptography

- Cryptographic program is written in C or ASM for efficiency.
- Computation over large finite field is not trivial in C and ASM.
- Split a large number into several smaller numbers (a.k.a. limbs).
  (e.g. 4 or 5 uint64_t/register to store 255-bit keys for Curve25519)
- Computation over limbs is error-prone.
- A simple bug can cause catastrophic damages.
  (e.g. a missing bound check in Heartbleed)

- Cryptographic program is written in C or ASM for efficiency.
- Computation over large finite field is not trivial in C and ASM.
- Split a large number into several smaller numbers (a.k.a. limbs).
  (e.g. 4 or 5 uint64_t/register to store 255-bit keys for Curve25519)
- Computation over limbs is error-prone.
- A simple bug can cause catastrophic damages.
  (e.g. a missing bound check in Heartbleed)

In this work, we focus on implementation written in C.

Functional Correctness

So.... How to achieve the functional correctness?

Functional Correctness

So.... How to achieve the functional correctness?
Test?

Functional Correctness

So.... How to achieve the functional correctness?
Test?
State space is too BIG, HARD to cover

Functional Correctness

So.... How to achieve the functional correctness?
Test?
State space is too BIG, HARD to cover

Verification

Outline

Previous Work

Previous Work

- Proof Assistant + SMT Solver (CHL+14)
  - can only verify some simple code in tolerable time.
  - many human-added annotations.

SMT: Satisfiability modulo theories

Previous Work

- Proof Assistant + SMT Solver (CHL+14)
  - can only verify some simple code in tolerable time.
  - many human-added annotations.
- Proof Assistant + SMT Solver + Algebra Solver (TWY17)
  - can deal with more complex operations like multiplication
    - SMT solver cannot deal with large integers multiplication well

SMT: Satisfiability modulo theories

Previous Work

- Proof Assistant + SMT Solver (CHL+14)
  - can only verify some simple code in tolerable time.
  - many human-added annotations.
- Proof Assistant + SMT Solver + Algebra Solver (TWY17)
  - can deal with more complex operations like multiplication
    - SMT solver cannot deal with large integers multiplication well
- DSL + SMT Solver + Algebra Solver (PTW+18)
  - Untyped CRYPTOLINE (only unsigned)
  - Target: ASM (some real-word examples in OpenSSL)
  - integer size is fixed (32/64 bit register)

SMT: Satisfiability modulo theories
DSL: Domain-specific language

Goal

- More real-world examples.
- Try to verify the C implementation once instead of ASM for every platforms.
  - most implementation now are still written in C instead of human-optimized ASM
- Less verification effort and friendly to normal cryptographic library developers.

Target Cryptographic Libraries

- OpenSSL: **UBIQUITOUS**
- BoringSSL: Chrome, Android
- NaCl: reference implementation
- wolfSSL: embedded systems
- Bitcoin's libsecp256k1: ECDSA used by **MANY** cryptocurrencies (Ethereum, Zcash, Ripple, $\cdots$)

What Curves We Verified

- OpenSSL: — 32/64: integer size
  - NIST P-224 : $2^{224} - 2^{96} + 1$ (unsigned 64)
  - NIST P-256 : $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (unsigned 64)
  - NIST P-521 : $2^{521} - 1$ (unsigned 64)
  - Curve25519 : $2^{255} - 19$ (unsigned 64, signed 32)
- BoringSSL: Curve25519 (unsigned 64)
- NaCl: Curve25519 (unsigned 64, signed 64)
- wolfSSL: Curve25519 (same as OpenSSL's) (signed 32)
- Bitcoin: Secp256k1 ($2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$)
  (unsigned, signed)

Contribution

- **Typed** CRYPTOLINE – unsigned and signed, arbitrary size integers
  - type system (type checking & type inference)
- A GCC plugin that translates GIMPLECRYPTOLINE into **Typed** CRYPTOLINE
- GIMPLECRYPTOLINE – a subset of GIMPLE
  - GIMPLE: a GCC IR used in machine-independent optimization
- Verify GIMPLE code after machine-independent optimization
- First to verify signed C implementation in cryptographic libraries used in industry
- Found a bug in NaCl's Curve25519 - Case study

Outline

Typed CryptoLine Program

- Program - instructions
- Specification
  - Assumption (Precondition)
  - Assertion (Postcondition)
  - Properties {algebra && range}
    - range: variables should be in a proper range (e.g. $a < 2^{51}$)
      checked by SMT solver (Boolector, MathSAT, Z3 $\cdots$)
    - algebra: mathematical properties (e.g. $c = a \times b$)
      checked by algebraic solver (Sage, Singular, Mathematica $\cdots$)
- Hoare triple: {assumption} program {assertion}

Typed CryptoLine Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

$$\text{limbs } 64\ [a_0, a_1, \cdots, a_n] = \sum_{i=0}^{n} a_i \times 2^{64 \times i}$$

Typed CRYPTOLINE Program Example - Naive Addition

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no assumption
  &&
  and [  // range prop
    a0 <u (2**63)@64, a1 <u (2**63)@64,
    b0 <u (2**63)@64, b1 <u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

$$2^{63} - 1 + 2^{63} - 1 = 2^{64} - 2 \leq 2^{64} - 1$$

Typed CRYPTOLINE Program Example - Overflow

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no restriction
  &&
  and [  // range prop
    a0 <=u (2**63)@64, a1 <=u (2**63)@64,
    b0 <=u (2**63)@64, b1 <=u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Overflow

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no restriction
  &&
  and [  // range prop
    a0 <=u (2**63)@64, a1 <=u (2**63)@64,
    b0 <=u (2**63)@64, b1 <=u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Overflow

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no restriction
  &&
  and [  // range prop
    a0 <=u (2**63)@64, a1 <=u (2**63)@64,
    b0 <=u (2**63)@64, b1 <=u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Overflow

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no restriction
  &&
  and [  // range prop
    a0 <=u (2**63)@64, a1 <=u (2**63)@64,
    b0 <=u (2**63)@64, b1 <=u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

Typed CRYPTOLINE Program Example - Overflow

```
proc main (uint64 a0, uint64 a1, uint64 b0, uint64 b1) =
{
  true // algebraic prop; true means no restriction
  &&
  and [  // range prop
    a0 <=u (2**63)@64, a1 <=u (2**63)@64,
    b0 <=u (2**63)@64, b1 <=u (2**63)@64
  ]
}
add c0 a0 b0; // c0 = a0 + b0
add c1 a1 b1; // c1 = a1 + b1
{
  limbs 64 [c0, c1]
  =
  limbs 64 [a0, a1] + limbs 64 [b0, b1]
  &&
  and [
    c0 >=u a0, c1 >=u a1 // true iff not overflow
  ]
}
```

$$2^{63} + 2^{63} = 2^{64} \not\leq 2^{64} - 1$$
$$2^{64} = 0 \; (mod \; 2^{64})$$

Program Safety Check by SMT Solver

Safety in our context means that following kinds of errors do not exist.

- Overflow / Underflow

Program Safety Check by SMT Solver

Safety in our context means that following kinds of errors do not exist.

- Overflow / Underflow
- Cast between types
  (uint64 $\leftrightarrow$ int64, uint64 $\leftrightarrow$ uint32)
  Value preserving casting (vpc)

2's complement representation for signed integers

uint4 $\leftrightarrow$ int4

$(0111)_2$ =	7	(unsigned) =	7	(signed)
$(1111)_2$ =	15	(unsigned) =	-1	(signed)

Program Safety Check by SMT Solver

Safety in our context means that following kinds of errors do not exist.

- Overflow / Underflow
- Cast between types
  (uint64 $\leftrightarrow$ int64, uint64 $\leftrightarrow$ uint32)
  Value preserving casting (vpc)

2's complement representation for signed integers

uint4 $\leftrightarrow$ int4

$(0111)_2$ =	7	(unsigned) =	7	(signed)	✓
$(1111)_2$ =	15	(unsigned) =	-1	(signed)	✗

BUG (vpc) or on purpose (cast)

Counterexample by SMT solver

Typed CRYPTOLINE Program Example - Cast v.s. VPC

```
proc main (uint64 a ,uint64 b)=
{
 true
 &&
 and [
     a <u (2**63), b <u (2**63)
  ]
}
cast wa@int64 a;
cast wb@int64 b;
mul c wa wb;
{ ... }
```

✓

```
proc main (uint64 a ,uint64 b)=
{
 true
 &&
 and [
     a <u (2**63), b <u (2**63)
  ]
}
vpc wa@int64 a;
vpc wb@int64 b;
mul c wa wb;
{ ... }
```

✓

Figure: cast = vpc in some cases

under the assumption, sign bit will never be 1.

Typed CRYPTOLINE Program Example - VPC Error

```
proc main (uint64 a ,uint64 b)=
{
 true
 &&
 and [
     a <=u (2**63), b <=u (2**63)
   ]
}
cast wa@int64 a;
cast wb@int64 b;
mul c wa wb;
{ ... }
```

✓

```
proc main (uint64 a ,uint64 b)=
{
 true
 &&
 and [
     a <=u (2**63), b <=u (2**63)
   ]
}
vpc wa@int64 a;
vpc wb@int64 b;
mul c wa wb;
{ ... }
```

✗

Figure: cast $\neq$ vpc in some cases

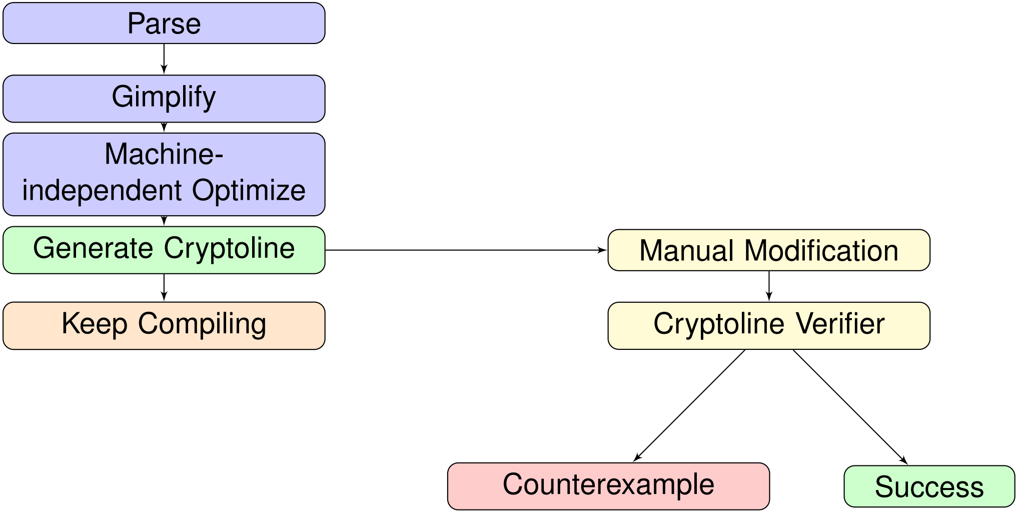
$$2^{63} = (100\ldots0)_2$$

Outline

GCC Plugin

- Introduced in GCC 4.5.0
- Let us add custom optimization passes
- Able to access AST (abstract syntax tree)
    - No need to write parser by yourself!

Verification Workflow Using GCC Plugin



GIMPLE Example

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t*)f_2(D)+4B];
    ...
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t*)g_13(D)+4B];
    ...
h0_24 = f0_3 - g0_14;
h1_25 = f1_4 - g1_15;
    ...
*h_34(D) = h0_24;
MEM[(int32_t*)h_34(D)+4B] = h1_25;
    ...
```

?LHS = MEM[?RHS] $\Rightarrow$ Load from RHS to LHS

MEM[?LHS] = ?RHS $\Rightarrow$ Store RHS to LHS

GIMPLE Example

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t*)f_2(D)+4B];
    ...
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t*)g_13(D)+4B];
    ...
h0_24 = f0_3 - g0_14;
h1_25 = f1_4 - g1_15;
    ...
*h_34(D) = h0_24;
MEM[(int32_t*)h_34(D)+4B] = h1_25;
    ...
```

?LHS = MEM[?RHS] $\Rightarrow$ Load from RHS to LHS

MEM[?LHS] = ?RHS $\Rightarrow$ Store RHS to LHS

GIMPLE Example

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t*)f_2(D)+4B];
     ...
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t*)g_13(D)+4B];
     ...
h0_24 = f0_3 - g0_14;
h1_25 = f1_4 - g1_15;
     ...
*h_34(D) = h0_24;
MEM[(int32_t*)h_34(D)+4B] = h1_25;
     ...
```

?LHS = MEM[?RHS]	$\Rightarrow$	Load from RHS to LHS
MEM[?LHS] = ?RHS	$\Rightarrow$	Store RHS to LHS

GIMPLE Example

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t*)f_2(D)+4B];
    ...
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t*)g_13(D)+4B];
    ...
h0_24 = f0_3 - g0_14;
h1_25 = f1_4 - g1_15;
    ...
*h_34(D) = h0_24;
MEM[(int32_t*)h_34(D)+4B] = h1_25;
    ...
```

?LHS = MEM[?RHS] $\Rightarrow$ Load from RHS to LHS

MEM[?LHS] = ?RHS $\Rightarrow$ Store RHS to LHS

GIMPLE Example

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t*)f_2(D)+4B];
    ...
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t*)g_13(D)+4B];
    ...
h0_24 = f0_3 - g0_14;
h1_25 = f1_4 - g1_15;
    ...
*h_34(D) = h0_24;
MEM[(int32_t*)h_34(D)+4B] = h1_25;
    ...
```

?LHS = MEM[?RHS] $\Rightarrow$ Load from RHS to LHS
MEM[?LHS] = ?RHS $\Rightarrow$ Store RHS to LHS

GimpleCryptoLine Example

generated by the plugin automatically. later manually add assumption / assertion.

```
proc main () =
{ true && true }
mov f03 f2_0; // f0_3 = *f_2
mov f14 f2_4; // f1_4 = MEM[(...) f_2 + 4]
        ...
mov g014 g13_0;
mov g115 g13_4;
        ...
sub h024 f03 g014; // h0_24 = f0_3 - g0_14
sub h125 f14 g115;
        ...
mov h34_0 h024; // *h_34 = h0_24
mov h34_4 h125; // MEM[(...) h_34 + 4]
{ true && true }
```

use pointer name and offset to represent the variable

GIMPLECRYPTOLINE Example

generated by the plugin automatically. later manually add assumption / assertion.

```
proc main () =
{ true && true }
mov f03 f2_0; // f0_3 = *f_2
mov f14 f2_4; // f1_4 = MEM[(...) f_2 + 4]
        ...
mov g014 g13_0;
mov g115 g13_4;
        ...
sub h024 f03 g014; // h0_24 = f0_3 - g0_14
sub h125 f14 g115;
        ...
mov h34_0 h024; // *h_34 = h0_24
mov h34_4 h125; // MEM[(...) h_34 + 4]
{ true && true }
```

use pointer name and offset to represent the variable

GIMPLECRYPTOLINE Example

generated by the plugin automatically. later manually add assumption / assertion.

```
proc main () =
{ true && true }
mov f03 f2_0; // f0_3 = *f_2
mov f14 f2_4; // f1_4 = MEM[(...) f_2 + 4]
        ...
mov g014 g13_0;
mov g115 g13_4;
        ...
sub h024 f03 g014; // h0_24 = f0_3 - g0_14
sub h125 f14 g115;
        ...
mov h34_0 h024; // *h_34 = h0_24
mov h34_4 h125; // MEM[(...) h_34 + 4]
{ true && true }
```

use pointer name and offset to represent the variable

GIMPLECRYPTOLINE Example

generated by the plugin automatically. later manually add assumption / assertion.

```
proc main () =
{ true && true }
mov f03 f2_0; // f0_3 = *f_2
mov f14 f2_4; // f1_4 = MEM[(...) f_2 + 4]
        ...
mov g014 g13_0;
mov g115 g13_4;
        ...
sub h024 f03 g014; // h0_24 = f0_3 - g0_14
sub h125 f14 g115;
        ...
mov h34_0 h024; // *h_34 = h0_24
mov h34_4 h125; // MEM[(...) h_34 + 4]
{ true && true }
```

use pointer name and offset to represent the variable

Outline

Case Study - Field Subtraction in NaCl Curve25519

```c
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

Case Study - Field Subtraction in NaCl Curve25519

```
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

Case Study - Field Subtraction in NaCl Curve25519

```c
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

Case Study - Field Subtraction in NaCl Curve25519

```
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

Case Study - Field Subtraction in NaCl Curve25519

```c
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

Case Study - Field Subtraction in NaCl Curve25519

```
typedef uint64_t felem;
/* Find the difference of two numbers: output = in - output
 * (note the order of the arguments!)
 */
static void fdifference_backwards(felem *ioutput, const felem *iin) {
  static const int64_t twotothe51 = (1l << 51);
  const int64_t *in = (const int64_t *) iin;
  int64_t *out = (int64_t *) ioutput;

  out[0] = in[0] - out[0]; out[1] = in[1] - out[1];
  out[2] = in[2] - out[2]; out[3] = in[3] - out[3];
  out[4] = in[4] - out[4];

  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1); NEGCHAIN(1, 2);
  NEGCHAIN(2, 3); NEGCHAIN(3, 4);
```

5 uint64 limbs and use signed computation

```c
int64_t t;

#define NEGCHAIN(a,b) \
  t = out[a] >> 63; \
  out[a] += twotothe51 & t; \
  out[b] -= 1 & t;

#define NEGCHAIN19(a,b) \
  t = out[a] >> 63; \
  out[a] += twotothe51 & t; \
  out[b] -= 19 & t;
```

```c
  NEGCHAIN(0, 1);
  NEGCHAIN(1, 2);
  NEGCHAIN(2, 3);
  NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1);
  NEGCHAIN(1, 2);
  NEGCHAIN(2, 3);
  NEGCHAIN(3, 4);
}
```

Figure: Bitwise tricks (signed right shift) & Reduction chain

```c
int64_t t;

#define NEGCHAIN(a,b) \
  t = out[a] >> 63; \
  out[a] += twotothe51 & t; \
  out[b] -= 1 & t;

#define NEGCHAIN19(a,b) \
  t = out[a] >> 63; \
  out[a] += twotothe51 & t; \
  out[b] -= 19 & t;
```

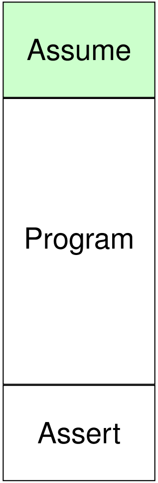
```c
  NEGCHAIN(0, 1);
  NEGCHAIN(1, 2);
  NEGCHAIN(2, 3);
  NEGCHAIN(3, 4);
  NEGCHAIN19(4, 0);
  NEGCHAIN(0, 1);
  NEGCHAIN(1, 2);
  NEGCHAIN(2, 3);
  NEGCHAIN(3, 4);
}
```

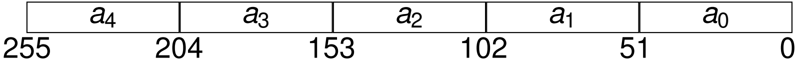
Figure: Bitwise tricks (signed right shift) & Reduction chain

sign_bit(out[a]) == 1/0 $\leftrightarrow$ t is all 1/0

Verify by CRYPTOLINE - Pre-condition

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3, uint64 a4,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3, uint64 b4) =
{
  true
  &&
  and [
    a0 <u (2**51)@64,
    a1 <u (2**51)@64,
    a2 <u (2**51)@64,
    a3 <u (2**51)@64,
    a4 <u (2**51)@64,
    b0 <u (2**51)@64,
    b1 <u (2**51)@64,
    b2 <u (2**51)@64,
    b3 <u (2**51)@64,
    b4 <u (2**51)@64
  ]
}
```





Verify by CryptoLine - Pre-condition

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3, uint64 a4,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3, uint64 b4) =
{
  true
  &&
  and [
    a0 <u (2**51)@64,
    a1 <u (2**51)@64,
    a2 <u (2**51)@64,
    a3 <u (2**51)@64,
    a4 <u (2**51)@64,
    b0 <u (2**51)@64,
    b1 <u (2**51)@64,
    b2 <u (2**51)@64,
    b3 <u (2**51)@64,
    b4 <u (2**51)@64
  ]
}
```

Assume
Program
Assert

a_4	a_3	a_2	a_1	a_0

255 204 153 102 51 0

Verify by CRYPTOLINE - Pre-condition

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3, uint64 a4,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3, uint64 b4) =
{
  true
  &&
  and [
    a0 <u (2**51)@64,
    a1 <u (2**51)@64,
    a2 <u (2**51)@64,
    a3 <u (2**51)@64,
    a4 <u (2**51)@64,
    b0 <u (2**51)@64,
    b1 <u (2**51)@64,
    b2 <u (2**51)@64,
    b3 <u (2**51)@64,
    b4 <u (2**51)@64
  ]
}
```

Assume
Program
Assert

a_4	a_3	a_2	a_1	a_0

255 204 153 102 51 0

Verify by CRYPTOLINE - Pre-condition

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3, uint64 a4,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3, uint64 b4) =
{
  true
  &&
  and [
    a0 <u (2**51)@64,
    a1 <u (2**51)@64,
    a2 <u (2**51)@64,
    a3 <u (2**51)@64,
    a4 <u (2**51)@64,
    b0 <u (2**51)@64,
    b1 <u (2**51)@64,
    b2 <u (2**51)@64,
    b3 <u (2**51)@64,
    b4 <u (2**51)@64
  ]
}
```

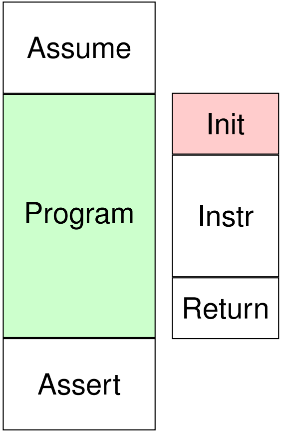
Assume
Program
Assert

a_4	a_3	a_2	a_1	a_0

255 204 153 102 51 0

Verify by CryptoLine - Init type casting

```
vpc iin52_0@int64 a0;
vpc iin52_8@int64 a1;
vpc iin52_16@int64 a2;
vpc iin52_24@int64 a3;
vpc iin52_32@int64 a4;
vpc ioutput53_0@int64 b0;
vpc ioutput53_8@int64 b1;
vpc ioutput53_16@int64 b2;
vpc ioutput53_24@int64 b3;
vpc ioutput53_32@int64 b4;
```



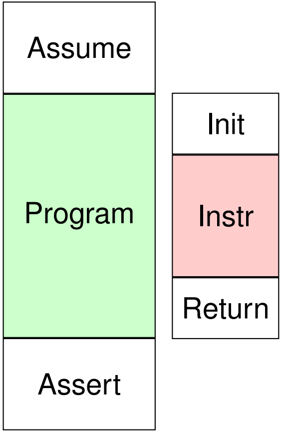
uint64 $\rightarrow$ int64

bridge assumption and program

vpc: value preserve casting (will do safety check)

Verify by CRYPTOLINE - Instructions

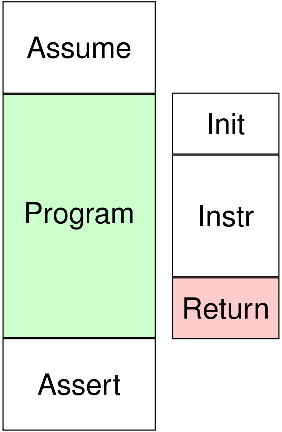
```
(* _1 = MEM[(const int64_t * )iin_52(D)]; *)
mov v1 iin52_0;
(* _2 = MEM[(int64_t * )ioutput_53(D)]; *)
mov v2 ioutput53_0;
(* _3 = _1 - _2; *)
ssub v3 v1 v2;
(* MEM[(int64_t * )ioutput_53(D)] = _3; *)
mov ioutput53_0 v3;
(* _4 = MEM[(const int64_t * )iin_52(D) + 8B]; *)
mov v4 iin52_8;
(* _5 = MEM[(int64_t * )ioutput_53(D) + 8B]; *)
mov v5 ioutput53_8;
(* _6 = _4 - _5; *)
ssub v6 v4 v5;
```



ssub: signed subtraction (usub/ssub explicitly ⇒ type checking, sub ⇒ type inference)

Verify by CryptoLine - Return type casting

```
vpc c0@uint64 ioutput53_0@int64;
vpc c1@uint64 ioutput53_8@int64;
vpc c2@uint64 ioutput53_16@int64;
vpc c3@uint64 ioutput53_24@int64;
vpc c4@uint64 ioutput53_32@int64;
```



int64 $\rightarrow$ uint64

bridge program and assertion

vpc: value preserve casting (will do safety check)

Verify by CRYPTOLINE - Post-condition

```
{
  (limbs 51 [c0, c1, c2, c3, c4])
   =
   (
     (limbs 51 [a0, a1, a2, a3, a4])
     -
     (limbs 51 [b0, b1, b2, b3, b4])
   )
   (mod (2**255 - 19))
 &&
 and [
   c0 <u (2**51)@64,
   c1 <u (2**51)@64,
   c2 <u (2**51)@64,
   c3 <u (2**51)@64,
   c4 <u (2**51)@64
 ]
}
```

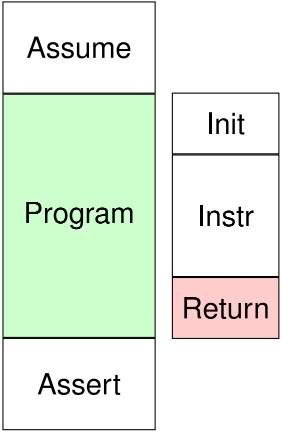
Assume
Program
Assert

$$\text{limbs}\,51\,[a_0, a_1, \cdots, a_n] = \sum_{i=0}^{n} a_i \times 2^{51\times i}$$

mod m: under modulo m

Verify by CRYPTOLINE - Return type casting - Revisit

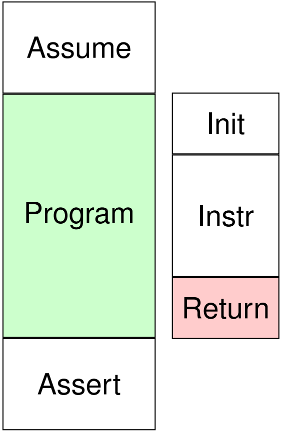
```
vpc c0@uint64 ioutput53_0@int64;
vpc c1@uint64 ioutput53_8@int64;
vpc c2@uint64 ioutput53_16@int64;
vpc c3@uint64 ioutput53_24@int64;
vpc c4@uint64 ioutput53_32@int64;
```



vpc: value preserve casting (will do safety check)

Verify by CRYPTOLINE - Return type casting - Revisit

```
vpc c0@uint64 ioutput53_0@int64;
vpc c1@uint64 ioutput53_8@int64;
vpc c2@uint64 ioutput53_16@int64;
vpc c3@uint64 ioutput53_24@int64;
vpc c4@uint64 ioutput53_32@int64;
```



vpc: value preserve casting (will do safety check)

Safety Check Failed

Counterexample generated by SMT solvers

```
(b4_0 (_ bv0 64))
(b3_0 (_ bv0 64))
(b2_0 (_ bv2251799813685250 64))
(b1_0 (_ bv0 64))
(b0_0 (_ bv2 64))
(a4_0 (_ bv0 64))
(a3_0 (_ bv1 64))
(a2_0 (_ bv0 64))
(a1_0 (_ bv1 64))
(a0_0 (_ bv2 64)) )
```

Figure: output by MathSAT

Counterexample generated by SMT solvers

```
(b4_0 (_ bv0 64))
(b3_0 (_ bv0 64))
(b2_0 (_ bv2251799813685250 64))
(b1_0 (_ bv0 64))
(b0_0 (_ bv2 64))
(a4_0 (_ bv0 64))
(a3_0 (_ bv1 64))
(a2_0 (_ bv0 64))
(a1_0 (_ bv1 64))
(a0_0 (_ bv2 64)) )
```

Figure: output by MathSAT

Found Counterexample translated to C language

```c
int main()
{
    felem in[5] = { 2, 1, 0, 1, 0 };
    felem out[5] = { 2, 0, 2251799813685250, 0, 0 };
    fdifference_backwards(out, in);
    for (int i = 0; i < 5; i++) {
        printf(" out%d: 0x%llx \n", i, out[i]);
    }
}
```

Check whether the program result is correct !

```
out0: 0x0
out1: 0x1
out2: 0x7ffffffffffe
out3: 0x7fffffffffff
out4: 0xffffffffffffffff
```

Check whether the program result is correct !

```
out0: 0x0
out1: 0x1
out2: 0x7fffffffffffe
out3: 0x7fffffffffffff
out4: 0xffffffffffffffff
```

- 0x0
- 0x1
- 0x5f6080e
- 0x0
- 0x0

Underflow and not in proper range

$$\texttt{0xffffffffffffffff} = 2^{64} - 1$$

Outline

Glimpse Evaluation Result

- 82 C functions (when paper is submitted)
- Evaluated on two different machines
  - much more range properties and safety check by SMT solver $\Rightarrow$ done in parallel
  - a few algebraic properties (most have only 1)
    - field operation
    - group operation

M1: Macbook 13”	2C/4T	16GB
M2: Ubuntu Server	18C/36T	1024GB

Evaluation Table - all functions

Table 2: Experimental Result

Function	L_{IR}	L_{CL}	D	P	TR_{M1}	MR_{M1}	TA_{M1}	MA_{M1}	TR_{M2}	MR_{M2}	TA_{M2}	MA_{M2}
nacl/curve25519/donna_c64/curve25519.c (MathSAT, SMT-LIB2 format)												
fdifference_backwards	69	69	66	0	-	-	0.23	6.3	-	-	0.14	9.1
fmul	91	127	10	14	12.51	452.2	0.20	6.3	4.05	486.6	0.14	9.3
fscalar_product	38	38	7	10	2.75	104.4	0.20	5.6	0.95	108.4	0.12	8.6
fsquare	68	116	10	12	7.44	288.1	0.22	6.3	2.61	301.0	0.13	9.3
fsum	20	20	0	0	0.48	5.6	0.15	4.8	0.22	10.0	0.10	8.2
fmonty	1147	1493	361	127	-	-	OOM	OOM	-	-	353.66	32764
wolfssl/fe_operations.c (Boolector with Lingeling, BTOR format)												
fe_add	40	40	0	0	1.48	6.5	0.19	5.6	0.61	9.5	0.11	8.6
fe_mul	305	305	20	24	OOT	OOT	0.32	7.0	13178	883.3	0.15	9.9
fe_mul121666	91	91	20	20	19.68	17.9	0.26	6.4	3.75	13.8	0.13	9.4
fe_neg	30	30	0	0	1.24	6.5	0.18	5.3	0.63	9.3	0.10	8.3
fe_sq	204	204	20	24	13411.84	351.9	0.33	6.7	2033	355.6	0.14	9.6
fe_sq2	214	214	20	24	18252.02	388.9	0.30	6.8	2763	385.5	0.14	9.6
fe_sub	40	40	0	0	1.31	6.5	0.16	5.7	0.64	9.4	0.11	8.6
curve25519[1]	2770	2770	200	236	OOT	OOT	12.06	385.6	68140	796.7	8.26	382.1
bitcoin/field_5x52_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_fe_add	13	20	0	0	0.33	5.3	0.14	4.8	0.22	10.0	0.09	8.3
secp256k1_fe_cmov	29	49	13	20	1.35	28.7	0.29	6.4	0.46	29.6	0.17	9.3
secp256k1_fe_from_storage	24	32	6	14	0.53	6.4	0.15	5.2	0.31	10.7	0.09	8.4
secp256k1_fe_mul_int	16	16	2	0	0.52	26.1	0.14	4.7	0.28	28.0	0.10	8.4
secp256k1_fe_negate	20	20	2	0	0.52	5.7	0.18	4.9	0.27	9.9	0.11	8.6
bitcoin/field_5x52_impl.h (Boolector with Lingeling, BTOR format)												
secp256k1_fe_normalize	52	60	21	0	117.18	45.3	0.12	5.3	91.89	31.5	0.08	8.3
secp256k1_fe_normalize_var	63	63	29	0	120.80	47.1	0.12	5.4	95.65	34.1	0.08	8.3
secp256k1_fe_normalize_weak	26	26	15	0	63.85	40.0	0.25	5.3	51.51	28.3	0.13	8.8
secp256k1_fe_normalizes_to_zero	34	39	10	0	203.12	60.3	0.16	5.2	151.03	42.9	0.08	8.2
bitcoin/field_5x52_int128_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_fe_mul_inner	111	137	17	24	16.09	461.0	0.22	6.5	4.00	489.1	0.14	9.5
secp256k1_fe_sqr_inner	90	116	21	22	9.91	284.5	0.20	6.4	2.72	303.2	0.14	9.3
bitcoin/scalar_4x64_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_scalar_add	81	102	55	22	2.03	10.1	0.21	6.5	1.11	14.1	0.13	9.4
secp256k1_scalar_eq	17	17	23	0	0.29	9.2	0.10	4.7	0.26	14.5	0.07	7.6
secp256k1_scalar_mul_512	273	384	136	90	13.75	263.3	0.26	7.1	4.96	280.0	0.16	9.9
secp256k1_scalar_mul	652	947	379	228	128.19	453.9	0.84	19.8	741.35	2219	0.43	16.3
secp256k1_scalar_negate	41	55	4	1	28.50	132.4	0.10	5.0	40.31	135.5	0.08	8.0
secp256k1_scalar_reduce_512	379	563	243	138	31.84	127.5	0.37	8.7	8.25	128.2	0.23	11.7
secp256k1_scalar_reduce	34	32	11	8	1.52	11.7	0.18	6.4	0.88	15.2	0.14	9.3
secp256k1_scalar_sqr_512	235	333	145	88	23.75	212.9	0.26	7.2	7.39	204.8	0.17	10.1
secp256k1_scalar_sqr	614	896	388	226	234.87	349.1	0.82	19.8	26.69	341.5	0.45	16.5
bitcoin/group_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_ge_from_storage	48	65	12	28	0.93	6.5	0.19	6.3	0.48	10.7	0.12	9.2
secp256k1_ge_neg	33	31	0	10	0.76	6.6	0.19	5.4	0.44	11.2	0.13	8.7
secp256k1_gej_add_ge_var	2109	2457	371	396	574.39	3166.9	OOM	OOM	75	3354	9363	70156
secp256k1_gej_double_var	899	1042	154	160	163.30	1703.0	0.77	18.4	25.27	1806	0.57	22.7
openssl/curve25519.c (MathSAT, SMT-LIB2 format)												
fe51_add	20	20	0	0	0.85	6.0	0.19	4.9	0.36	10.0	0.10	8.3
fe51_mul	96	105	11	20	17.95	381.2	0.26	6.4	3.69	409.3	0.13	9.2
fe51_mul121666	44	44	11	14	1.3	17.3	0.25	5.8	0.63	20.2	0.12	8.7
fe51_sq	73	82	11	20	8.07	227.0	0.23	6.3	2.22	247.6	0.14	9.2
fe51_sub	25	25	10	10	0.37	6.8	0.24	5.4	0.26	11.4	0.13	8.9
x25519_scalar_mult[1]	923	1047	110	194	558.56	1419.8	187.40	5538	119.89	1472	145.12	5511
openssl/ecp_nistp224.c (MathSAT, SMT-LIB2 format)												
felem_diff_128_64	24	36	0	0	0.56	6.4	0.23	5.1	0.32	10.7	0.14	8.6
felem_diff	24	24	0	0	0.55	5.8	0.19	4.9	0.33	10.4	0.11	8.8
felem_mul	40	40	0	0	2.24	83.2	0.15	5.2	0.65	88	0.09	8.2
felem_mul_reduce	82	121	15	16	10.65	321.8	0.20	6.4	3.11	322.5	0.13	9.1
felem_neg	47	58	5	10	0.95	6.8	0.19	5.8	0.55	11.1	0.12	8.7
felem_reduce	56	95	6	18	1.67	13.7	0.20	6.3	0.88	17.3	0.13	9.3
felem_scalar	12	12	0	0	0.48	26.7	0.14	4.6	0.24	28.9	0.09	8.1
felem_square	27	27	0	0	1.11	45.1	0.15	4.9	0.43	47.6	0.10	8.2
felem_square_reduce	69	108	14	18	6.36	195.8	0.21	6.4	1.81	198.8	0.13	9.2
felem_sum	16	16	0	0	0.41	5.4	0.15	4.7	0.26	10.0	0.10	8.3
widefelem_diff	41	63	0	0	0.90	6.5	0.19	5.7	0.46	10.6	0.12	8.7
widfefelem_scalar	21	21	0	0	2.58	87.7	0.14	4.8	0.70	88.3	0.10	8.4
openssl/ecp_nistp256.c (MathSAT, SMT-LIB2 format)												
felem_diff	24	36	0	0	0.59	7.6	0.18	5.1	0.35	11.7	0.12	8.6
felem_scalar	13	13	0	0	0.70	47.7	0.17	4.6	0.31	48.8	0.10	8.2
felem_shrink	65	95	18	16	1.78	14.0	0.20	6.4	0.95	17.1	0.13	9.3
felem_small_mul	145	95	17	46	4.75	123.0	0.23	7.0	2.29	123.2	0.14	9.8
felem_small_sum	20	20	0	0	0.41	5.8	0.14	4.8	0.25	10.2	0.10	8.4
felem_sum	16	16	0	0	0.41	5.6	0.14	4.7	0.24	10.3	0.09	8.2
smallfelem_mul	88	136	0	30	2.80	91.9	0.17	6.4	1.22	95.4	0.11	9.4
smallfelem_neg	26	28	0	0	0.1	5.4	0.19	4.9	0.27	9.7	0.12	8.6
smallfelem_square	60	108	0	20	1.92	55.8	0.15	6.3	0.85	55.5	0.10	9.2
openssl/ecp_nistp521.c[2] (MathSAT, SMT-LIB2 format)												
felem_diff64	45	45	18	18	0.81	6.9	0.20	6.4	0.48	11.4	0.13	9.3
felem_diff128	45	72	18	18	1.13	7.9	0.21	6.4	0.47	11.9	0.12	9.2
felem_neg	27	27	0	0	0.77	6.4	0.18	5.3	0.48	10.0	0.12	8.6
felem_reduce	122	155	74	72	4.10	7.8	0.24	6.7	2.06	10.8	0.14	9.6
felem_scalar	27	27	0	0	0.80	28.4	0.14	5.0	0.36	29.0	0.09	8.3
felem_scalar64	27	27	0	0	0.82	28.2	0.15	4.9	0.35	28.9	0.09	8.3
felem_scalar128	27	27	0	0	1.26	48.4	0.14	5.0	0.41	48.8	0.09	8.4
felem_sum64	36	36	0	0	0.49	6.0	0.14	5.2	0.29	10.0	0.10	8.3
felem_diff_128_64	54	54	0	0	1.34	7.2	0.29	6.0	0.68	11.4	0.15	8.7
felem_mul	188	188	0	0	23.92	187.0	0.22	6.6	3.13	182.5	0.13	9.5
felem_square	111	111	0	0	7.38	95.5	0.21	6.4	0.99	103.9	0.13	9.3
boringssl/fiat/curve25519.c (MathSAT, SMT-LIB2 format)												
fe_add	11	20	0	0	0.33	5.3	0.14	4.8	0.20	10.0	0.10	8.2
fe_mul_impl	96	108	9	22	18.39	452.9	0.21	6.4	5.11	473.9	0.13	9.2
fe_mul121666	43	43	9	14	1.12	18.4	0.20	5.7	0.62	21.2	0.11	8.6
fe_sqr_impl	73	85	9	22	10.59	278.7	0.26	6.3	3.11	293.0	0.12	9.2
fe_sub	15	25	0	0	0.51	5.9	0.19	5.0	0.28	10.4	0.11	8.8
x25519_scalar_mult_generic[1]	927	1073	161	212	470.68	1489.0	120.33	5726	118.95	1579	91.99	5766

Some comparisons

Montgomery Ladder step* involves 4 add, 4 sub, 4 square, 6 mul
(Curve25519) (field operations)

F	U/S	L_{IR}	L_{CL}	TR_{M1}	TA_{M1}	TR_{M2}	TA_{M2}	TH
openSSL	5 * U64	923	1047	9.3m	0.93s	2m	0.61s	
boringSSL	5 * U64	927	1073	7.8m	0.89s	2m	0.56s	
boringSSL	10 * U32	2715	3419	27.5m	59s	6.3m	42s	2h
wolfSSL	10 * S32	2770	2770	OOT	12s	18.9h	8s	

L_{IR}: lines of IR

L_{CL}: lines of CRYPTOLINE

TR(range, safety), TA(algebra): used time

OOT: used time $>$ 1day

TH: human effort (one person)

Montgomery Ladder is used for scalar multiplication of elliptic curve point

$$Q = aP$$

Outline

Conclusion

- A lightweight and easy to use method to verify cryptographic software involving both unsigned/signed operations.
- A GCC Plugin reducing human effort
- Verify several functions in well-known cryptographic libraries.
  - OpenSSL
  - BoringSSL
  - NaCl
  - wolfSSL
  - Bitcoin's libsecp256k1

CryptoLine Verifier

GCC Plugin

This Slide[1]

github.com/fmlab-iis

Signed Cryptographic Program Verification with Typed CRYPTOLINE

Open Access

[1] twleo.com/slides/ccs19-slide.pdf