Verifying Arithmetic in Cryptographic C Programs

Jiaxiang Liu*, Xiaomu Shi*†, Ming-Hsien Tsai†, Bow-Yaw Wang† and Bo-Yin Yang†

*College of Computer Science and Software Engineering, Shenzhen University
†Institute of Information Science, Academia Sinica
{jiaxiang0924, xshi0811, mhtsai208}@gmail.com, {bywang, byyang}@iis.sinica.edu.tw

Abstract—Cryptographic primitives are ubiquitous for modern security. The correctness of their implementations is crucial to resist malicious attacks. Typical arithmetic computation of these C programs contains large numbers of non-linear operations, hence is challenging existing automatic C verification tools. We present an automated approach to verify cryptographic C programs. Our approach successfully verifies C implementations of various arithmetic operations used in NIST P-224, P-256, P-521 and Curve25519 in OpenSSL. During verification, we expose a bug and a few anomalies that have been existing for a long time. They have been reported to and confirmed by the OpenSSL community. Our results establish the functional correctness of these C implementations for the first time.

Keywords—program verification; cryptographic programs; functional correctness; OpenSSL

I. INTRODUCTION

Cryptographic primitives are the foundation of modern computer security. They are invoked for authentication, encryption, and key exchange protocols, among others. Unlike normal programs, typical cryptographic or security settings always assume an adversary who would take advantage of any mistakes and run out of his ways to induce errors so as to launch attacks. As illustrated in [1], even a tiny bug can have catastrophic impacts. Consequently, the correctness of cryptographic primitives is of the utmost importance.

Cryptography programming however is far from easy. Modern cryptography relies on complicated mathematical constructions. Consider, for instance, Elliptic Curve Cryptography (ECC) [2], [3]. Such cryptosystems are based on arithmetic over large finite fields. Take the elliptic curve Curve25519 [4] used in OpenSSH [5] as an example. It is defined over finite field \(\mathbb{Z}_{2^{255} - 19}\). Each field element hence belongs to the integer set \(\{0, 1, \ldots, 2^{255} - 20\}\); sums and products of two field elements are computed by addition and multiplication modulo \(2^{255} - 19\) respectively. A point on Curve25519 is a pair of field elements \((x, y)\) satisfying the curve equation \(y^2 = x^3 + 486662x^2 + x\), or the symbolic point at infinity. An operation on points called point addition can then be defined on top of those field operations. With point addition, a group is defined over points on Curve25519. Point multiplication further takes hundreds of point addition operations. And it is required by the public-key primitives over Curve25519, such as those in the default key exchange protocol in OpenSSH.

Reality is even more complicated than mathematics. Observe that a field element in \(\mathbb{Z}_{2^{255} - 19}\) can be represented by a 255-bit number. Yet there are no computers with 255-bit architectures available. In practice, a field element is represented by four 64- or five 51-bit numbers in 64-bit architectures. Arithmetic over the finite field has to be implemented on such representations. In such implementations, field multiplication requires several 64-bit multiplication and addition instructions. Carries must be propagated. Modular computation must be performed. Cryptography programming can be very challenging even for experienced programmers.

Curve25519 is but one elliptic curve in ECC. In the widely used security library OpenSSL [6], cryptographic primitives based on four different curves over different finite fields are provided. In addition to Curve25519, three NIST-recommended curves (P-224, P-256, and P-521) are used. Each curve is defined over its special finite field. Each finite field has its dedicated C functions for field arithmetic. One wonders if there might be errors in these building blocks of computer security. Indeed, the OpenSSL source code can only be modified by a chosen group of 12 developers for security purposes [7]. Restricting code commits can reduce the probability but not remove the possibility of bugs in the library. Concern about correctness of cryptographic C programs in OpenSSL thus has some justification.

Program verification is an active research field with numerous promising ideas. One naturally hopes that all such programs could be formally verified. Yet existing techniques do not appear to be able to verify cryptographic C programs.

Motivating Example. Montgomery reduction [8] is a widely used algorithm in cryptography programming. Let \(B = 2^{32}\). Given integer inputs \(N, N'\) and \(T\) with \(NN' + 1 \equiv 0 \pmod{B}\), Montgomery reduction is an efficient way to calculate \(TB^{-1} \pmod{N}\) without long division. Fig. 1 shows a simplified Montgomery reduction algorithm. Observe that division and modulo by \(B\) are bit shifting and masking operations respectively for \(B\) is a power of 2. The algorithm thus computes \(TB^{-1} \pmod{N}\) with addition, multiplication, bit shifting and masking operations. Long division by \(N\) is indeed not needed. A reference C implementation is as follows, where \(N < 2^{31}\) is assumed for simplicity.

†Corresponding author
Pre-condition: \(0 < N < B\) with \(N \equiv 1 \pmod{2}\), \(0 < N' < B\) with \(NN' + 1 \equiv 0 \pmod{B}\), and \(0 \leq T < BN\)

Post-condition: REDC\(^{-1}\)(\(N, N', T\)) \(\times B \equiv T \pmod{m}\) (mod \(N\))

function REDC\(^{-1}\)(\(N, N', T\))

\[
m \leftarrow ((T \mod B)N') \mod B
\]

\[
t \leftarrow (T + mN)/B
\]

return \(t\)

Figure 1. Montgomery Reduction

Table I

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPA-SEQ (SV-COMP2019 version)</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>-default -heap 10000M</td>
<td></td>
</tr>
<tr>
<td>-svcomp19 -heap 10000M</td>
<td></td>
</tr>
<tr>
<td>PeSCO (SV-COMP2019 version)</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>-svcomp19-pesco -heap 10000M</td>
<td></td>
</tr>
<tr>
<td>-svcomp19-pesco-linear -heap 10000M</td>
<td></td>
</tr>
<tr>
<td>-stack 2048k</td>
<td></td>
</tr>
<tr>
<td>UAUTOMIZER (SV-COMP2019 version)</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>--architecture 64bit</td>
<td></td>
</tr>
<tr>
<td>--verifier boogie</td>
<td>FALSE</td>
</tr>
<tr>
<td>--bit-precise --verifier boogie</td>
<td></td>
</tr>
<tr>
<td>--verifier corral</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>--bit-precise --verifier corral</td>
<td></td>
</tr>
<tr>
<td>--verifier symbooglix</td>
<td>FALSE</td>
</tr>
<tr>
<td>--bit-precise --verifier symbooglix</td>
<td></td>
</tr>
<tr>
<td>--verifier duality</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>--bit-precise --verifier duality</td>
<td></td>
</tr>
</tbody>
</table>

typedef uint64_t u64;
#define B ((u64)1 << 32)
u64 REDC(u64 N, u64 Np, u64 T) {
const u64 btm32bits = 0xFFFFFFFF;
u64 m = ((T & btm32bits) + Np) & btm32bits;
u64 t = (T + mN) >> 32;
return t;
}

The inputs \(N, Np\) and \(T\) are 64-bit integers. Given \(N < 2^{31}\) and pre-conditions in Fig. [1] we would like to verify whether REDC\((N, Np, T) \times B \equiv T \pmod{N}\). We have tried automatic C verification tools including CPA-SEQ [9], PeSCO [10], UAUTOMIZER [11], and SMACK [12] on a Linux machine with 2-core 3.60GHz CPUs and 16GB RAM. No tool can verify the 8-line C program in 15 minutes (Table [I]). Two FALSE's are reported, but the counterexamples turn out to be spurious. Real cryptographic C programs in OpenSSL implement operations on field elements with hundreds of bits. Using existing verification tools, it is very unlikely to verify these programs within a reasonable time limit.

1The SV-COMP2019 versions of CPA-SEQ, PeSCO and UAUTOMIZER are downloadable at https://sv-comp.sosy-lab.org/2019/systems.php.
2Due to a bug of the tool (see the issue at https://github.com/smackers/smack/issues/427), SMACK did claim that REDC\((\cdot)\) was verified. The real output is UNKNOWN.

In order to verify cryptographic C programs, new techniques are needed. In [13], the modeling language CRYPTO and its tool for verifying cryptographic assembly programs are proposed. We leverage the work by translating LLVM IR programs to CRYPTO and use its tool to verify cryptographic C programs. More specifically, the following steps are needed to verify cryptographic C programs:

1) Submit a cryptographic C program to Clang and generate a program in LLVM IR.
2) Use our translator to convert the LLVM IR program to a CRYPTO program.
3) Specify properties about the C program in the generated CRYPTO program.
4) Verify whether the CRYPTO program conforms to the specification with the CCRYPTO verification tool.

Using our translator, the 8-line reference C implementation for Montgomery reduction (Fig. [1]) is verified within 10 seconds. We then apply our approach to the cryptographic C programs for arithmetic operations over the four elliptic curves (NIST P-224, NIST P-256, NIST P-521, and Curve25519) in OpenSSL. 38 cryptographic C functions in OpenSSL are verified. The largest function (x25519_scalar_mult) has 1153 LLVM IR instructions and is verified within 50 minutes on a dedicated Linux server. The function implements the critical step in the group operation on Curve25519. It takes 5 255-bit field elements as inputs and returns 4 255-bit field elements as outputs. Its specification consists of three nonlinear multivariate polynomial modulo equations over 45 (= (5 + 4) × 5) 64-bit variables. We are not aware of any other similar technique at such a scale.

We would like to point out a bug found during verification. In the function felem_diff_128_64 for the NIST P-521 curve, our approach exposes an overflow error in the implementation. We have reported our findings to the OpenSSL developer community. The community confirmed the bug and released a fix\(^{[4]}\). To the credits of the community, we only found one bug and minor anomalies in 3 C functions out of 38. Yet programming errors did occur in this widely used and inspected security library. One can never be too careful about security libraries.

Our Contributions. We identify a useful subset of LLVM IR (called LLVMCRYPTO) to model intermediate representations of cryptographic programs emitted from Clang. LLVMCRYPTO contains the most common instructions used in implementations of arithmetic operations. These instructions however form the core of many public-key cryptographic programs. Using LLVMCRYPTO, a number of cryptographic programs are modeled.

Given an LLVMCRYPTO program, we develop a translator to translate it into a CRYPTO program. CRYPTO is

https://github.com/openssl/openssl/commit/13f8ce17fc5b02c240f1c38685f8e02d6647c51
We also expose a bug and two incorrect input assumptions.

Ell vector variable or a sequence of

B. Syntax

LLVM programs. The syntax of

masking and shifting are widely used. We consider the subset

define the function

is, we have

a

quotient and non-negative remainder of

and all integers, respectively. We use

1

theorem is established for our translator. It guarantees that no

translated to C

for variables and

A. Notations

Let \( \mathbb{N} \), \( \mathbb{N}^+ \) and \( \mathbb{Z} \) denote the set of non-negative, positive, and all integers, respectively. We use \([n]\) to denote the set \(\{0, 1, \ldots, n-1\}\) for \(a \in \mathbb{N}^+\), \(a \div b\) and \(a \mod b\) denote the quotient and non-negative remainder of \(a\) divided by \(b\). That is, we have \(a = b \times (a \div b) + (a \mod b)\) with \(0 \leq a \mod b < b\). Let \(f : A \to B\) be a function. For \(a \in A\) and \(b \in B\), define the function \(f[a \leftarrow b] : A \to B\) by

\[
\begin{align*}
    f[a \leftarrow b](x) &= b & \text{if } x = a \\
    f[a \leftarrow b](x) &= f(x) & \text{otherwise}.
\end{align*}
\]

B. Syntax

LLVM IR is a strongly typed language. In addition to variables and constants, it supports pointers and vectors. In cryptographic C programs, arithmetic computation, bitwise masking and shifting are widely used. We consider the subset LLVMCRYPTO that is useful to the compilation of these programs. The syntax of LLVMCRYPTO is shown in Fig. 2.

We use \(x, y, z, \ldots\) for variables and \(p, q, \ldots\) for pointers. An argument (Arg) can be a variable or a number. Let \(\ell \in \mathbb{N}^+\). An argument for a vector of size \(\ell\) (Arg(\(\ell\))) can be a vector variable or a sequence of \(\ell\) numbers. In LLVMCRYPTO, operands and the result of each instruction can be 64- or 128-bit values, specified by the instruction syntactically. For instance, the instruction \(y = \text{add} 64 a_1 a_2\) adds the 64-bit operands \(a_1, a_2\) together, and assigns the sum to the 64-bit variable \(y\). On the other hand, \(y = \text{add} 128 a_1 a_2\) has 128-bit operands and result.

Let \(w \in \{64, 128\}\). \(y = \text{addv} <\ell \times w> a_1 a_2\) computes the element-wise sum of the vectors \(a_1, a_2\), and assigns the result to the vector variable \(y\) whose \(\ell\) elements are of bit width \(w\). The instructions \(\text{sub}\) and \(\text{mul}\), as well as their vector versions \(\text{subv}\) and \(\text{mulv}\), work similarly.

Two bitwise shifting instructions are defined in LLVM-CRYPTO. \(y = \text{shl} w a n\) shifts the \(w\)-bit operand \(a\) to the left by \(n < w\) bits, and stores the result as a \(w\)-bit value in \(y\). Instruction \(\text{lsr}\) on the other hand shifts to the right. The bitwise AND instruction is \(y = \text{and} w a_1 a_2\).

The instruction \(y = \text{load} w p\) loads the \(w\)-bit value from pointer \(p\). To load a vector of \(w\)-bit values, \(y = \text{loadv} <\ell \times w> p\) is used. The instructions \(\text{store}\) and \(\text{storev}\) store values into the memory.

One can obtain the pointer to an element of a vector stored in memory. \(q = \text{geteltptr} w p n\) makes \(q\) point to the \(n\)-th \(w\)-bit element of the vector designated by \(p\). If \(p\) points to a vector whose elements are vectors of size \(\ell\), \(q = \text{geteltptrv} <\ell \times w> p n_1 n_2\) sets \(q\) to the pointer at the \(n_2\)-th element of the \(n_1\)-th vector designated by \(p\).

The instruction \(y = \text{trunc} a\) truncates the 128-bit value \(a\). 128
to the low 64 bits and stores the result in the 64-bit variable \( y \). \( y = \text{zext} \ a \) extends the 64-bit operand \( a \) to 128 bits.

Finally, the instruction \( y = \text{insertelt} < \ell \times w > a_1 a_2 k \) assigns to \( y \) the \( \ell \)-long vector identical to \( a_1 \) except that its \( k \)-th element is \( a_2 \) where \( k < \ell \). An LLVMCRYPTO program is simply a sequence of instructions separated by semicolons.

There are no control-flow instructions like branching in LLVMCRYPTO. Those are avoided in typical cryptographic programs for side-channel attack prevention.

**Example.** The file ecp_nistp521.c in OpenSSL implements the NIST P-521 elliptic curve over the prime \( p_{521} = 2^{521} - 1 \). In this implementation, a field element \( a \) is represented as \( a_0 + a_1 \times 2^{58 \times 1} + a_2 \times 2^{58 \times 2} + \cdots + a_8 \times 2^{58 \times 8} \) using nine 64-bit limbs \( a_i \)'s. The following LLVMCRYPTO fragment is extracted from the LLVM IR code of the C function `elem_diff64`. It subtracts a field element \( y \) represented by \( y_0, \ldots, y_8 \) from the field element \( x \) represented by \( x_0, \ldots, x_8 \). The result is then stored in the memory designated by \( p_{\text{out}} \).

```plaintext
1: \( v_0 = \text{sub} 64 4611686018427387872 y_0; \)
2: \( v_0' = \text{add} 64 v_0 x_0; \)
3: \( g_0 = \text{geteltptr} 64 p_{\text{out}} 0; \)
4: \( \text{store} 64 v_0' g_0; \)
```

The fragment only shows the operations on the least significant limb. \( y_0 \) is subtracted from a constant at line 1. The result is added to \( x_0 \) at line 2. Line 3 computes \( g_0 \) pointing to the 0-th 64-bit element of the memory designated by \( p_{\text{out}} \). The calculation result \( v_0' \) is stored to the memory cell pointed by \( g_0 \) at line 4.

An LLVMCRYPTO program is in SSA form (Static Single Assignment) if its variables and pointers are defined at most once. Any LLVM IR program generated from Clang is in SSA form.

**C. Semantics**

Similar to its syntax, the semantics of LLVMCRYPTO is designed for cryptographic C programs. Observe that field elements in OpenSSL are represented by unsigned integers. Our semantics is hence defined over unsigned numbers. We moreover assume the underlying architecture is 64-bit for simplicity. Each memory cell represents a value in \( [2^{64}] \). It is straightforward to modify the semantics of LLVMCRYPTO for 32-bit architectures.

We give a small-step semantics for LLVMCRYPTO. Let \( \sigma \in \mathcal{V} \triangleq \{ \text{Var} \cup \text{Ptr} \} \rightarrow \mathbb{N} \) be a valuation, and \( m \in \mathcal{M} \triangleq \mathbb{N} \rightarrow [2^{64}] \) a memory state. \( S \triangleq \mathcal{V} \times \mathcal{M} \) is the set of states. A valuation formalizes the values of variables and pointers. The content of memory cells is modeled by a memory state. Our semantics specifies how a state transits to another by executing each instruction. Fig. 3 gives the semantics of LLVMCRYPTO.

Given \( \sigma \in \mathcal{V} \), we define the semantic function \([\cdot]_\sigma\) for numbers, variables and pointers as follows.

\[
[a]_\sigma = \begin{cases} 
    a & \text{if } a \in \text{Num} \\
    \sigma(a) & \text{if } a \in \text{Var} \cup \text{Ptr}
\end{cases}
\]

From the state \( (\sigma, m) \), the instruction \( y = \text{add} w a_1 a_2 \) moves to the state \( (\sigma', m) \) where \( \sigma' \) updates the value of \( y \) to \( ([a_1]_\sigma + [a_2]_\sigma) \mod 2^w \). The sum is truncated to \( w \) bits by modulo \( 2^w \). Other variables in \( \sigma \) remain unchanged in \( \sigma' \).

More notations are needed for vectors. For \( \ell \in \mathbb{N}^+ \) and \( v \in \text{Num}^\ell \), \( v[i] \) denotes the \( i \)-th element of \( v \) in \( \ell \). We also use the variable \( x[i] \) for the \( i \)-th element of the vector variable \( x \in \text{Var} \). Given a valuation \( \sigma \) and \( n \in \mathbb{N} \), the notation \( \sigma[a_i \leftarrow b_i] \) is short for \( \sigma[a_0 \leftarrow b_0] \cdots [a_n \leftarrow b_n] \). The semantics of \( y = \text{addv} < \ell \times w > a_1 a_2 \) should now be clear. It updates the vector variable \( y \) with the element-wise sum of vectors \( a_1, a_2 \); and each element sum is truncated to a \( w \)-bit value. The semantics for subtraction and multiplication is similar and omitted from Fig. 3 for clarity. The semantics for bitwise instructions `shl`, `ishr` and `and` is obvious.

In our memory model, addresses are natural numbers and memory cells are elements in \( [2^{64}] \) because we assume a 64-bit architecture. Let \( m \in \mathcal{M} \) be a memory state and \( n, v \in \mathbb{N} \), we use the following notations for convenience:

\[
m_{64}(n) \triangleq m(n)
\]

\[
m_{64}[n \leftarrow v] \triangleq m[n \leftarrow v \mod 2^{64}]
\]

\( m_{64}(n) \) reads the memory cell located at the address \( n \); \( m_{64}[n \leftarrow v] \) updates the cell located at \( n \) with the value \( v \).

In LLVMCRYPTO, we also need to interpret two consecutive memory cells as a 128-bit value. We choose the little-endian representation in our semantics. Define:

\[
m_{128}(n) \triangleq m(n + 1) \times 2^{64} + m(n)
\]

\[
m_{128}[n \leftarrow v] \triangleq m[n \leftarrow v][n + 1 \leftarrow v'][n + 2 \leftarrow v'][n + 3 \leftarrow v'][n + 4 \leftarrow v'][n + 5 \leftarrow v'][n + 6 \leftarrow v'][n + 7 \leftarrow v'][n + 8 \leftarrow v'][n + 9 \leftarrow v'][n + 10 \leftarrow v'][n + 11 \leftarrow v'][n + 12 \leftarrow v'][n + 13 \leftarrow v'][n + 14 \leftarrow v'][n + 15 \leftarrow v'][n + 16 \leftarrow v'][n + 17 \leftarrow v'][n + 18 \leftarrow v'][n + 19 \leftarrow v'][n + 20 \leftarrow v'][n + 21 \leftarrow v'][n + 22 \leftarrow v'][n + 23 \leftarrow v'][n + 24 \leftarrow v'][n + 25 \leftarrow v'][n + 26 \leftarrow v'][n + 27 \leftarrow v'][n + 28 \leftarrow v'][n + 29 \leftarrow v'][n + 30 \leftarrow v'][n + 31 \leftarrow v']
\]

where \( v^L = v \mod 2^{64} \) and \( v^H = (v \div 2^{64}) \mod 2^{64} \). Hence \( m_{128}(n) \) reads a 128-bit value from the memory cells located at \( n \); \( m_{128}[n \leftarrow v] \) updates the memory cells located at \( n \) with the 128-bit value \( v \).

The semantics of \( y = \text{load} w p \) can now be explained. It updates \( y \) by the \( w \)-bit value in the memory cell designated by \( p \). To load a vector of values, define \( \text{size}(w) \triangleq w \div 64 \) for the number of memory cells needed for \( w \)-bit values. By \( y = \text{loadv} < \ell \times w > p \), the vector variable \( y \) is updated with \( \ell \ w \)-bit values from the memory cells designated by \( p \). The instructions `store` and `storev` are defined similarly:

If \( p \) points to a vector of \( w \)-bit values in memory, the \( n \)-th element is located at \( [p]_\sigma + [n]_\sigma \times \text{size}(w) \). This is exactly what \( q = \text{geteltptr} w p \ n \) computes. \( \text{geteltptrv} \) is defined similarly when \( p \) points to a vector of vectors.

The semantics of instructions `trunc` and `zext` is straightforward. Finally, \( y = \text{insertelt} < \ell \times w > a_1 a_2 k \) copies
the vector $a_1$ of $\ell$ $w$-bit values to $y$ and then updates the
$k$-th element of $y$ with the $w$-bit value $a_2$.

### III. Domain-Specific Language CRYPTO LINE

**CRYPTO LINE** [13] is a domain-specific language for
 cryptographic assembly programs and their verification. It
 is equipped with an automatic verification tool. We briefly
 review the language and its verification in this section.

#### A. The Language

CRYPTO LINE serves as an abstraction for cryptographic
assembly programs across different architectures. Details such
as registers and address modes are ignored in the language.
For simplicity, it only considers variables, numbers and
flags. Typical arithmetic assembly instructions are modeled
in CRYPTO LINE. Fig. [4] gives the syntax of the language.

The semantics of CRYPTO LINE is parameterized by the bit
width of the underlying architecture. To be consistent with
LLVM CRYPTO, the semantics of CRYPTO LINE is explained
here for 64-bit architectures. All arguments (Arg) are hence
assumed 64-bit. The formal semantics can be found in [13].

A CRYPTO LINE state models the current values of vari-
ables and flags (cIFlag). Set is the assignment statement
and Cset is the conditional assignment. Carry and borrow flags
are explicit in CRYPTO LINE. Add $b$ $x$ $u$ $v$ sets the sum of $u$ and $v$
to $x$ with carry in $b$. Adec is the addition-with-carry statement.
Sub and Sbb are subtraction and subtraction-with-borrow
statements, respectively. Full multiplication Mul $x$ $y$ $u$ $v$
updates $x$ and $y$ with the high and low 64 bits of the product
of $u$ and $v$, respectively. And is the bitwise AND statement.
Shl $x$ $u$ $n$ shifts the value of $u$ to the left by $n$ bits and
assigns the result to $x$ if the high $n$ bits of $u$ are all zero;
otherwise the CRYPTO LINE program goes into the error
state.

$$\text{cIFlag ::= b } | \text{ c } | \text{ d } | \ldots$$

$$\text{clExp ::= Arg } | \text{ clExp } + \text{ clExp } | \text{ clExp } - \text{ clExp }$$

$$| \text{ clExp } * \text{ clExp }$$

$$\text{clPred ::= clExp } = \text{ clExp } | \text{ clExp } \equiv \text{ clExp } \mod \text{ clExp }$$

$$| \text{ clExp } < \text{ clExp } | \text{ clExp } \leq \text{ clExp } | \text{ clPred } \land \text{ clPred }$$

$$\text{clStmt ::= Set Var Arg } | \text{ Cset Var cIFlag Arg Arg }$$

$$| \text{ Add clFlag Var Arg Arg }$$

$$| \text{ Adec clFlag Var Arg cFlag Arg }$$

$$| \text{ Sub clFlag Var Arg Arg }$$

$$| \text{ Sbb clFlag Var Arg cFlag Arg }$$

$$| \text{ Mul Var Arg Arg } | \text{ And Var Arg Arg }$$

$$| \text{ Shl Var Arg Num } | \text{ Split Var Var Arg Num }$$

$$| \text{ Assert clPred } | \text{ Assume clPred }$$

$$\text{clProg ::= e } | \text{ clStmt;clProg }$$

### Figure 3. Semantics of LLVM CRYPTO LINE

### Figure 4. CRYPTO LINE Statements and Programs

Split is provided to model common patterns of assembly
code in cryptographic programs. The statement **Split** $x$ $y$ $u$ $n$
splits the value of $u$ into two parts: the low $n$ bits are moved
to $y$ and the remaining high bits are moved to $x$.

For verification purposes, CRYPTO LINE supports assertions
and assumptions. Predicates (clPred) $e_1 = e_2$ and
$e_1 \equiv e_2 \mod e_3$ are algebraic properties, $e_1 < e_2$ and
$e_1 \leq e_2$ are range properties. **Assert pred** checks if the
predicate **pred** holds in the current state. If so, the execution
continues with the same state. Otherwise, it enters the error
state. **Assume pred** on the other hand assumes **pred** holds
at the current program location, thus the execution continues
with states satisfying **pred**. No predicate is satisfied in the
error state. A common usage for assertions and assumptions
is to add external information for verification. Let us assume,
say, $answer = 42$ at some program location but this predicate is obscure. In CRYPTO LINE, a human verifier can assert the predicate and then assume it to pass the predicate to the verification tool. The assertion ensures the predicate indeed holds at the location; the assumption then adds the predicate as a lemma for verification.

A CRYPTO LINE program is simply a sequence of CRYPTO LINE statements followed by semicolons.

B. Verification with Specifications

In addition to programs, CRYPTO LINE allows to specify pre- and post-conditions using predicates ($\Pi$Pred). Pre- and post-conditions together compose specifications. Given a CRYPTO LINE program with its specification, we would like to know if the program will end in a state satisfying the post-condition whenever it starts from a state satisfying the pre-condition. The CRYPTO LINE verification tool checks if a CRYPTO LINE program conforms to its specification automatically.

Example (continued). According to the comments of felem diff64 in ecp nistp521.c, the pre-condition of the program is the range property $\bigwedge_{i=0}^{8} y_i < 2^{59} + 2^{14}$. Assume that the nine consecutive memory cells designated by $p_{out}$ are represented by variables $addr_{p_0}, addr_{p_1}, \ldots, addr_{p_8}$ (explained later). The post-condition of the CRYPTO LINE program generated from felem diff64 is

\[
((addr_{p_0} < x_0 + 2^{62}) \land \cdots \land (addr_{p_8} < x_8 + 2^{62}))
\land (radix 58(x_0, x_1, \ldots, x_8) - \text{radix} 58(y_0, y_1, \ldots, y_8))
\equiv \text{radix} 58(addr_{p_0}, \ldots, addr_{p_8}) \mod p_{521}
\]

where $\text{radix} 58(a_0, a_1, \ldots, a_8) = a_0 + a_1 \times 2^{58 \times 1} + a_2 \times 2^{58 \times 2} + \cdots + a_8 \times 2^{58 \times 8}$ denotes the field element represented by $a_i$'s. The first part of the post-condition is a range property. The second part is an algebraic property stating that the result element is a difference between the inputs over the prime $p_{521}$.

IV. TRANSLATING LLVMCRYPTO TO CRYPTO LINE

Given an LLVMCRYPTO program, we first translate it into a CRYPTO LINE program in order to verify with the CRYPTO LINE verification tool. The soundness property of the translation means that the generated program captures all behaviors of the input one, being an over-approximation of it. We introduce the translation and discuss its soundness in this section.

A. Symbolic Memory Addresses

In CRYPTO LINE and its semantics model, there are no pointers or memory. But it is different in LLVMCRYPTO. Both pointers and memory are considered, which is closer to reality. We bridge this by representing memory addresses symbolically then using these symbols to translate pointers.

Assume $\mathcal{S}Var = \{p, q_1, \ldots\}$ is a set of symbolic variables. $A \triangleq \mathcal{S}Var \times \mathbb{Z}$ is called the set of symbolic (memory) addresses. A symbolic address $(p, o) \in A$ represents the memory address having an offset $o$ from the memory address represented by $(p, 0) \in A$. In LLVMCRYPTO, a pointer is only allowed to be added with a constant offset. Hence we define the addition $+_{A} : A \times \mathbb{Z} \rightarrow A$ as $(p, o_1) +_{A} o_2 \triangleq (p, o_1 + o_2)$, where $(p, o_1) \in A$ and $o_2 \in \mathbb{Z}$. This is sufficient to model pointer calculations in LLVMCRYPTO. For instance, the following two instructions are commonly used for accessing the $i$-th element of the array $a$ designated by pointer $p$:

\[
q = \text{geteltptr} 64 p i;
\]
\[
y = \text{load} 64 q;
\]

where $y$ is the value of $a[i]$. If $p$ refers to the memory address $n_p$, $q$ will have the value $n_q = n_p + i$ according to the semantics. If $n_p$ is symbolically represented by $(p, o_p)$, our translation algorithm is sufficient to calculate $n_q$'s symbolic address as $(p, o_q) = (p, o_p) +_{A} i$. Even though symbolic addresses cannot capture complete information about memory addresses, they reflect the relationships between the absolute memory addresses. For example, the offset $i$ between $n_p$ and $n_q$ is preserved for their symbolic representations.

We define a pointer table $pt$ as a mapping from pointers $Ptr$ to symbolic addresses $A$. It models the valuation of pointers and helps alias analysis in the translation algorithm. $pt(p)$ models the memory address represented by $p$.

B. Translation Algorithm

Assume the pointer table $pt$ models the valuation of pointers before executing an LLVMCRYPTO instruction $s$. The function $\text{INSTToClProg}(pt, s)$ translates $s$ into a sequence $cp$ of CRYPTO LINE statements. It returns a pair $(pt', cp)$, where $pt'$ models the valuation of pointers after executing $s$. $pt'$ reflects the effect on pointers when executing $s$ with respect to the LLVMCRYPTO semantics. We then translate a given LLVMCRYPTO program by sequentially applying $\text{INSTToClProg}()$ to each instruction. We summarize the translation for 64-bit LLVMCRYPTO instructions as in Table I.

The translation of arithmetic instructions is straightforward. For example, the instruction $\text{add}$ is translated to the statement $\text{Add}$. Note that carries are not present in LLVMCRYPTO. Hence the introduced carry flag is discarded using a fresh carry flag.

The translation of bitwise shifting is a little subtle. The instruction $\text{shl}$ has similar semantics as $\text{Shl}$, except that $\text{Shl}$ may cause an error that is undesired by $\text{Shl}$. To avoid that,
the high \( n \) bits of \( a \) are discarded first by \texttt{Split} using the fresh variable \( z^d \). Then the remaining low \( 64 - n \) bits stored in the temporary variable \( t \) can be safely shifted to the left by \( n \) bits. Similar translation is applied for \texttt{lshr}. The translation for and is trivial.

To translate an instruction involving memory, the memory cell is referred to using a CRYPTOINE variable. Assume that we have a one-to-one function \( \rho : A \rightarrow Var \). It converts each symbolic address into a CRYPTOINE variable. For example, \( \rho(p, 1) \) is straightforward.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Output ((pt', cp))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = \text{addv} &lt;\times 64&gt; a )</td>
<td>((pt, \text{Add } y a a2; \text{Addr } y a2))</td>
</tr>
<tr>
<td>( y = \text{addv} &lt;\times 64&gt; a )</td>
<td>((pt, \text{Add } y a a2; \text{Addr } y a2))</td>
</tr>
<tr>
<td>( y = \text{shl} 64 a n )</td>
<td>((pt, \text{Split } z^d t a (64 - n); \text{Shl } y n t))</td>
</tr>
<tr>
<td>( y = \text{lsrr} 64 a n )</td>
<td>((pt, \text{Split } y z^d a n; \text{Shr } y n t))</td>
</tr>
<tr>
<td>( y = \text{shl} 64 a n )</td>
<td>((pt, \text{Split } y z^d a n; \text{Shr } y n t))</td>
</tr>
<tr>
<td>( y = \text{load} &lt;\times 64&gt; p n )</td>
<td>((pt[\text{Load } y p n] = \text{Addr } y a2))</td>
</tr>
<tr>
<td>( y = \text{trunc} a )</td>
<td>((pt, \text{Set } y a H n))</td>
</tr>
<tr>
<td>( y = \text{insertelt} &lt;\times 64&gt; a )</td>
<td>((pt, \text{sequence of Set's}))</td>
</tr>
</tbody>
</table>

In Table II, we only show the translation algorithm, which is trivial. An updated Set is translated similarly, as treated in Table III.

When translating \( q = \text{getelt} \text{ptr} 64 p n \), we obtain the symbolic address \( pt(p) \) and add it with offset \( n \times \text{size}(64) = n \). Then the value of \( q \) in \( pt \) is updated with the result. No CRYPTOINE statement is required since it does not modify variables or memory, hence \( cp = \epsilon \). The translation for \texttt{getelt}ptrv is similar and hence omitted.

Given any 128-bit LLVMCRYPTO variable \( y \), two CRYPTOINE variables \( y^L \) and \( y^H \) are used to represent its low and high 64 bits, respectively, in the translation. The same applies to any number, thus any argument. Then it is straightforward to translate instructions \texttt{trunc} and \texttt{zext} with Set statements. As well, Set statements are used to translate \texttt{insertelt} by copying each element of \( a \) to \( y \) with the \( k \)-th element \( y[k] \) assigned by \( a2 \).

For 128-bit instructions, the idea is the same but more technical. For instance, when translating a 128-bit \texttt{add}, two 64-bit \texttt{Add’s} are required to mimic 128-bit addition. One 128-bit \texttt{load} needs two \texttt{Set’s} to copy two consecutive cells to \( y^L \) and \( y^H \). The technicalities are not detailed here.

Given an LLVMCRYPTO program \( prog \), a variable or a pointer is undefined if it is not assigned by any instructions in \( prog \). Undefined variables (denoted by \( Var_U \)) and pointers (by \( PTr_U \)) are usually the input variables and pointers of the program.

Now the LLVMCRYPTO program translation is straightforward with ISToCLE program. The algorithm is depicted in Fig. 5. We first construct the initial pointer table \( pt_0 \) as a mapping that maps each \( p_j \) to \( PTr_U \) in \( prog \), to the symbolic address \( \rho(p_j, 0) \). All \( p_j \)'s are distinct. With \( pt_0 \), the algorithm starts from the first instruction \( s_1 \). An updated pointer table \( pt_1 \) and a fragment of CRYPTOINE program \( cp_1 \) are obtained. It then continues to translate the next instruction \( s_2 \) with \( pt_1 \). \( pt_i \) is obtained at the \( i \)-th iteration. It actually models the valuation of pointers after executing \( i \) instructions of \( prog \). Finally, all \( cp_i \) 's are combined in order as the output CRYPTOINE program.

**Example** (continued). Given the whole LLVM program, we know that \( p_{out} \) is an undefined pointer. We let \( pt_0(p_{out}) = (p, 0) \) when constructing \( pt_0 \). According to the translation algorithm, \( pt_2 = pt_1 = pt_0 \) after translating lines 1 and 2. Line 3 is translated into no CRYPTOINE statements, but updates \( q_0 \) in \( pt_3 \) with \( pt_4(q_0) = pt_2(p_{out}) +_A 0 = (p, 0) \). Let \( \rho(p, 0) = \text{Addr } y a H n \). The LLVMCRYPTO program fragment is translated into the following CRYPTOINE program fragment by our algorithm:

\[
\begin{align*}
1a : & \quad \text{Sub } d_0 v_0 4611686018427387872 y_0; \\
2a : & \quad \text{Add } d_1 v_0 v_0 x_0; \\
4a : & \quad \text{Set } addr \_ p \_0 v_0; \\
\end{align*}
\]

**C. Soundness**

Given an initial state \( (s_0, m_0) \), an LLVMCRYPTO program is well-formed if (1) all its undefined variables and undefined pointers have their values in \( s_0 \), and (2) it is in SSA form.

We make an assumption on how programs access the memory.

**Separation Assumption.** The memory is divided into several isolated segments. Each segment \( \pi_j \) contains one base address designated by an undefined pointer \( p_j \in PTr_U \). Let \( pt_0(p_j) = (p_j, 0) \). Every address in \( \pi_j \) is uniquely represented by the symbolic address \( \rho(p_j, o) \) for some \( o \in Z \) during translation.

This assumption is indeed common in cryptographic programs. Assume that a cryptographic arithmetic function
has two arrays \(a\) and \(b\) as parameters. Pointers \(p_a\) and \(p_b\) are the inputs pointing to their base addresses, i.e. the addresses of \(a[0]\) and \(b[0]\). Then the address of \(a[i]\) can be, and is always, calculated via \(p_a\). No one will do this via \(p_b\). The separation assumption is inspired by separation logic [13].

Given an LLVM program \(\text{prog}\) with initial state \((\sigma_0, m_0)\), we use \((\sigma_i, m_i)\) to denote the state after executing the first \(i\) instructions. That is, for the \(i\)-th instruction \(s_i\) of \(\text{prog}\), we have \((\sigma_{i-1}, m_{i-1}) \xrightarrow{s_i} (\sigma_i, m_i)\). We define a simulation relation \(\preceq\) between LLVM states \((\sigma, m)\) and CRYPTO states \(\rho\). \((\sigma, m) \preceq \rho\) reads as \((\sigma, m)\) is simulated by \(\rho\). \((\sigma, m) \preceq \rho\) holds if the values of variables and the content of memory in \((\sigma, m)\) are correctly projected into \(\rho\). For example, given a 128-bit LLVM variable \(x\), its corresponding 64-bit representations \(x^L\) and \(x^H\) in LLVM should satisfy \(\rho(x^L) + \rho(x^H) \times 2^{64} = \sigma(x)\).

We prove the soundness property of our translation via the following theorem:

**Theorem 1.** Given Separation Assumption and a well-formed LLVM program \(\text{prog}\) with initial state \((\sigma_0, m_0)\). The generated CRYPTO program \(\text{clprog} = \text{PROGToCLPROG}(\text{prog}) = (c_1 \cdots c_n)\) satisfies:

1. for all \(i \in [0, n]\), there exists a CRYPTO state \(\rho_i\) of \(\text{clprog}\) such that \((\sigma_i, m_i) \preceq \rho_i\);
2. for all \(i \in [1, n]\) and \(\rho\) with \((\sigma_{i-1}, m_{i-1}) \xrightarrow{c_i} (\sigma_i, m_i)\) and \((\sigma_{i-1}, m_{i-1}) \preceq \rho\), there exists \(\rho'\) such that \(\rho \preceq \rho'\) and \((\sigma_i, m_i) \preceq \rho'\).

Theorem 1 guarantees that after translation, (1) each state of the input LLVM program has its corresponding simulation CRYPTO state(s) of the generated program; (2) each execution trace of the input LLVM program has its corresponding simulation trace(s) of the generated CRYPTO program. Therefore, all the behaviors of the input program are captured by the generated one.

With Theorem 1, assume that the given LLVM program \(\text{prog}\) has \(n\) instructions. If a property \(P_{\text{llvm}}\) does not hold in the final state \((\sigma_n, m_n)\), then there must exist a CRYPTO state \(\rho_n\) (with \((\sigma_n, m_n) \preceq \rho_n\)) of the generated program and a trace to \(\rho_n\), such that the corresponding property \(P_{\text{cl}}\) does not hold in \(\rho_n\) either. In other words, if the verification tool fails during execution, then \(P_{\text{cl}}\) holds in all possible final states \(\rho_n\)’s (even if \((\sigma_n, m_n) \not\preceq \rho_n\)) along all possible execution traces of the generated CRYPTO program, then \(P_{\text{llvm}}\) is guaranteed to hold in \((\sigma_n, m_n)\) of the input \(\text{prog}\). Therefore, if our verification result shows that the generated CRYPTO program is correct with respect to the specification, it implies that the input LLVM program is also correct. Hence the input C program is correct.

**D. Implementation Heuristics**

Although our translator is fully automatic, extra human effort is sometimes needed to get the generated CRYPTO programs verified due to limitations of the CRYPTO verification tool. In our first implementation of the translator, we found that it took large amounts of human work to verify the generated CRYPTO programs. However, most of the work was repetitive and tedious. We develop four kinds of heuristics to reduce human efforts in the current implementation, including heuristics for special bitwise shifting, for special and, for overflow/underflow, and for the and-\text{after}-\text{lshr} pattern. The former two apply specialized translation when the arguments of the input instruction have specific values. The latter two are detailed as follows. We stress that all the heuristics retain the soundness property of our verification results.

1) **Heuristics for Overflow/Underflow**: In the translation of the add, sub, and mul instructions, we introduce carry/borrow flags that indicate the presence of overflow/underflow in the input LLVM instructions. For example, when translating \(y = \text{add} a_1 a_2\), the statement \(\text{Add } d \ y a_1 a_2\) is used. A new flag \(d\) is introduced that indicates the presence of overflow in \(\text{add}\). But in most cases in cryptographic programs, such an overflow will not happen thanks to the careful range assumptions on inputs. Nevertheless, it is difficult for the CRYPTO tool to deduce \(d = 0\) automatically and use this information to verify specified properties. Our heuristic hence automatically inserts the following two CRYPTO statements for each flag \(d\) of such a kind during translation:

\[
\text{Ass} \ = \ d = 0; \quad \text{Ass} \ = \ d = 0;
\]

The Assert statement tells CRYPTO to check whether \(d\) is 0. If it is, then \text{Ass} utilizes this information to ease the verification. Only if it is not, an overflow may arise. Human efforts hence are needed to investigate the problem.

This heuristic also applies to the translation of \text{shl} for the same reason, to check whether the value of \(z^d\) equals 0.

2) **Heuristics for and-\text{after}-\text{lshr}**: In cryptographic programs, a masking and instruction often follows an \text{lshr} instruction to perform a splitting together. For instance, the following pattern is common:

\[
\begin{align*}
y_1 &= \text{lshr} 64 a 51; \\
y_2 &= \text{and} 64 a 0x7FFFFFFF;
\end{align*}
\]

\(y_1\) and \(y_2\) get the high 13 bits and the low 51 bits, respectively, of \(a\). By our translation algorithm, they are translated to:

\[
\begin{align*}
\text{Split } y_1 \ &= \ z^d \ a \ 51; \\
\text{And } y_2 \ &= \ 0x7FFFFFFF; \\
\end{align*}
\]

But CRYPTO requires the extra information \(z^d = y_2\) to pass the verification. We implement heuristics to insert the following statements automatically to help the verification:

\[
\text{Ass} \ = \ z^d = y_2; \quad \text{Ass} \ = \ z^d = y_2;
\]

Note that in practice, the instructions \text{lshr} and \text{and} may not be adjacent. They may not have exactly the same \(a\) as operands. And several pairs of \text{lshr} and \text{and} may even interleave. It makes this and-\text{after}-\text{lshr} pattern more
complicated. The implemented simple heuristic only relates and to the previous lshl. However, it works in most of the scenarios we have encountered. A more precise analysis of the pattern can further improve the automation of our technique.

V. Evaluation

We have implemented our translator on LLVM 3.7.0 and successfully applied our approach to 38 C implementations of arithmetic operations in cryptographic primitives of OpenSSL 1.1.1. Among them, 35 are verified. One bug and several anomalies are exposed and confirmed in the remaining 3 functions.

A. Experiment Setup

The verification proceeded in the following way. Given a C implementation of an arithmetic operation, first we compiled it into LLVM IR using Clang 3.7.0. Then the LLVM IR code was translated by our translator to Cryptol. The very few instructions not supported by LLVMCRYPTO required manual translation. The generated Cryptol program was then verified by the verification tool. Some of them required human efforts to annotate the program, like adding Assert’s. The verification was performed on two machines respectively: a Mac laptop M1 running OS X 10.11.6 with a 2-core 2.6GHz CPU and 8GB RAM, and a Linux machine M2 running Ubuntu 16.04.5 with two 6-core 3.47GHz CPUs and 128GB RAM. Boolector 3.0.0 and Singular 4.1.1 were utilized as SMT solver and ideal memory tool.

B. Verification Tasks

The verified implementations include fundamental arithmetic operations in Curve25519 and three NIST elliptic curves (P-224, P-256 and P-521). Each curve has its own special finite field \( \mathbb{F}_p \): for Curve25519, \( p = 2^{255} - 19 \) for Curve25519; \( p = 2^{224} - 2^{96} + 1 \) for NIST P-224, \( p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \) for NIST P-256, and \( p = 2^{521} - 1 \) for NIST P-521. The field elements of different bit widths over different curves hence have different representations. Their arithmetic implementations thus differ.

Most of the specifications of these arithmetic operations came from the comments in OpenSSL source code. For those without specifications written in the comments, we determined their specifications from the context of their usage inside OpenSSL. The pre-conditions of the verified specifications are range properties on inputs. And the post-conditions contain both algebraic and range properties relating outputs to the inputs. See the example in Section III-B.

The most complicated algebraic post-condition we have verified is part of the Montgomery Ladderstep [16] in Curve25519: \( X_1 \times X_5 \times (X_2 \times Z_3 - Z_2 \times X_3)^2 \equiv Z_5 \times (X_2 \times Z_3 - Z_2 \times Z_3)^2 \mod (2^{255} - 19) \). Each of \( X_1, X_2, X_3, X_5, Z_2, Z_3 \) and \( Z_5 \) is a field element represented by five 64-bit limbs. Note that Montgomery Ladderstep is a crucial step to compute point multiplication over Curve25519 efficiently and securely. It requires 18 field operations that are implemented as individual functions. Such complicated algebraic properties involving large numbers cannot even be specified in existing general-purpose C verification tools, let alone be verified by them.

C. Experiment Results

The results of the experiment are summarized in Table III. In the table, the “loc-ir” column displays the number of lines of LLVM IR code for each function, and “loc-cl” for their Cryptol code. The “diff-*” columns show the percentage of manual modifications in each Cryptol program. “diff-0” is for our translator with no heuristics implemented and “diff-h” is with all four heuristics. Note that specifying pre- and post-conditions does not count, but modifying one line (i.e. one statement) counts two: one deletion and one addition. Finally, \( T_1 \) and \( T_2 \) are the verification time in seconds on M1 and M2, respectively. They do not contain translation time from LLVMCRYPTO to Cryptol. The translation for the largest target with 1153 instructions only took less than 5 seconds. The others took less than 2 seconds. We highlight the results as follows:

1) For functions felem_diff_128_64, felem_mul and felem_square in ecp_nistp521 (marked with “.”), our approach shows that they do not conform to the specifications given in the OpenSSL source code comments. More details are given in Section V-D.

2) Most of the verified tasks (71.4%, 25 of 35) are finished within only 5 seconds even on M1. 32 (91.4%) of them take less than 20 seconds. Two of those left require around 1 minute. The largest target in the experiment, the Montgomery Ladderstep (the last row) makes M1 out-of-memory (marked with “OM”). It requires around 47 minutes on M2. A further experiment showed that this can be accelerated by parallelization supported by Cryptol. The verification time on M2 is then reduced to 1288 seconds (around 21 minutes) by using option “--jobs 6” to parallelize with 6 threads. This improves the scalability of our approach.

3) The “diff-*” columns show that the automation of our approach is greatly improved by our heuristics. With these heuristics, most of the tasks (65.8%, 25 of 38) are verified fully automatically. Almost all (92.1%, 35 of 38) need only less than 10% of manual modifications. We believe that more heuristics can further reduce these efforts and improve the usability of our tool.

D. Bug and Anomalies in ecp_nistp521

The functions felem_diff_128_64, felem_mul and felem_square in ecp_nistp521.c implement subtraction, multiplication and squaring on field elements respectively.
### Table III

<table>
<thead>
<tr>
<th>function</th>
<th>loc-ir</th>
<th>loc-cl</th>
<th>diff-0 (%)</th>
<th>diff-h (%)</th>
<th>$T_1$ (s)</th>
<th>$T_2$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>felem_diff</td>
<td>30</td>
<td>40</td>
<td>40.0</td>
<td>0.0</td>
<td>0.40</td>
<td>0.18</td>
</tr>
<tr>
<td>felem_diff_128_64</td>
<td>30</td>
<td>60</td>
<td>26.7</td>
<td>0.0</td>
<td>0.85</td>
<td>0.73</td>
</tr>
<tr>
<td>felem_mul</td>
<td>60</td>
<td>298</td>
<td>49.0</td>
<td>0.0</td>
<td>9.64</td>
<td>8.80</td>
</tr>
<tr>
<td>felem_scalar</td>
<td>15</td>
<td>20</td>
<td>40.0</td>
<td>0.0</td>
<td>0.16</td>
<td>0.08</td>
</tr>
<tr>
<td>felem_square</td>
<td>43</td>
<td>193</td>
<td>47.7</td>
<td>0.0</td>
<td>1.96</td>
<td>1.20</td>
</tr>
<tr>
<td>felem_sum</td>
<td>22</td>
<td>24</td>
<td>33.3</td>
<td>0.0</td>
<td>0.23</td>
<td>0.10</td>
</tr>
<tr>
<td>widefelem_diff</td>
<td>54</td>
<td>112</td>
<td>25.0</td>
<td>0.0</td>
<td>2.75</td>
<td>2.67</td>
</tr>
<tr>
<td>felem_mul_reduce</td>
<td>99</td>
<td>493</td>
<td>58.4</td>
<td>9.3</td>
<td>73.47</td>
<td>71.02</td>
</tr>
<tr>
<td>felem_neg</td>
<td>47</td>
<td>145</td>
<td>49.0</td>
<td>9.0</td>
<td>1.07</td>
<td>0.57</td>
</tr>
<tr>
<td>felem_reduce</td>
<td>75</td>
<td>246</td>
<td>50.0</td>
<td>7.7</td>
<td>4.56</td>
<td>3.94</td>
</tr>
<tr>
<td>felem_square_reduce</td>
<td>82</td>
<td>388</td>
<td>60.3</td>
<td>11.9</td>
<td>64.83</td>
<td>61.45</td>
</tr>
<tr>
<td>widefelem_scalar</td>
<td>31</td>
<td>136</td>
<td>45.6</td>
<td>2.9</td>
<td>5.28</td>
<td>4.72</td>
</tr>
</tbody>
</table>

### ecp_nistp224.c

<table>
<thead>
<tr>
<th>function</th>
<th>loc-ir</th>
<th>loc-cl</th>
<th>diff-0 (%)</th>
<th>diff-h (%)</th>
<th>$T_1$ (s)</th>
<th>$T_2$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>felem_diff</td>
<td>30</td>
<td>64</td>
<td>25.0</td>
<td>0.0</td>
<td>2.09</td>
<td>2.11</td>
</tr>
<tr>
<td>felem_mul</td>
<td>16</td>
<td>74</td>
<td>43.2</td>
<td>0.0</td>
<td>0.55</td>
<td>0.31</td>
</tr>
<tr>
<td>felem_small_sum</td>
<td>26</td>
<td>44</td>
<td>18.2</td>
<td>0.0</td>
<td>0.39</td>
<td>0.31</td>
</tr>
<tr>
<td>felem_sum</td>
<td>22</td>
<td>40</td>
<td>20.0</td>
<td>0.0</td>
<td>0.34</td>
<td>0.27</td>
</tr>
<tr>
<td>smallfelem_mul</td>
<td>109</td>
<td>488</td>
<td>49.2</td>
<td>0.0</td>
<td>8.58</td>
<td>6.59</td>
</tr>
<tr>
<td>smallfelem_neg</td>
<td>22</td>
<td>36</td>
<td>22.0</td>
<td>0.0</td>
<td>0.25</td>
<td>0.12</td>
</tr>
<tr>
<td>felem_shrink</td>
<td>65</td>
<td>160</td>
<td>63.8</td>
<td>26.3</td>
<td>4.55</td>
<td>4.55</td>
</tr>
<tr>
<td>felem_small_mul</td>
<td>175</td>
<td>672</td>
<td>51.9</td>
<td>6.7</td>
<td>15.06</td>
<td>12.88</td>
</tr>
<tr>
<td>smallfelem_square</td>
<td>74</td>
<td>330</td>
<td>51.5</td>
<td>1.2</td>
<td>4.55</td>
<td>3.25</td>
</tr>
</tbody>
</table>

### ecp_nistp256.c

<table>
<thead>
<tr>
<th>function</th>
<th>loc-ir</th>
<th>loc-cl</th>
<th>diff-0 (%)</th>
<th>diff-h (%)</th>
<th>$T_1$ (s)</th>
<th>$T_2$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>felem_diff</td>
<td>61</td>
<td>81</td>
<td>44.4</td>
<td>0.0</td>
<td>0.84</td>
<td>0.49</td>
</tr>
<tr>
<td>felem_diff_128</td>
<td>61</td>
<td>126</td>
<td>28.6</td>
<td>0.0</td>
<td>17.59</td>
<td>18.30</td>
</tr>
<tr>
<td>felem_neg</td>
<td>43</td>
<td>45</td>
<td>40.0</td>
<td>0.0</td>
<td>0.50</td>
<td>0.24</td>
</tr>
<tr>
<td>felem_scalar</td>
<td>43</td>
<td>45</td>
<td>40.0</td>
<td>0.0</td>
<td>0.97</td>
<td>0.95</td>
</tr>
<tr>
<td>felem_scalar_64</td>
<td>35</td>
<td>45</td>
<td>40.0</td>
<td>0.0</td>
<td>1.12</td>
<td>1.15</td>
</tr>
<tr>
<td>felem_scalar_128</td>
<td>36</td>
<td>162</td>
<td>44.4</td>
<td>0.0</td>
<td>3.61</td>
<td>3.61</td>
</tr>
<tr>
<td>felem_sum_64</td>
<td>52</td>
<td>54</td>
<td>33.3</td>
<td>0.0</td>
<td>0.28</td>
<td>0.13</td>
</tr>
<tr>
<td>felem_reduce</td>
<td>145</td>
<td>317</td>
<td>53.0</td>
<td>17.0</td>
<td>2.29</td>
<td>1.36</td>
</tr>
<tr>
<td>felem_diff_128_64</td>
<td>70</td>
<td>126</td>
<td>28.6</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>felem_mul</td>
<td>289</td>
<td>1618</td>
<td>49.9</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>felem_square</td>
<td>158</td>
<td>892</td>
<td>51.2</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### curve25519.c

<table>
<thead>
<tr>
<th>function</th>
<th>loc-ir</th>
<th>loc-cl</th>
<th>diff-0 (%)</th>
<th>diff-h (%)</th>
<th>$T_1$ (s)</th>
<th>$T_2$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe51_add</td>
<td>32</td>
<td>30</td>
<td>33.3</td>
<td>0.0</td>
<td>0.18</td>
<td>0.07</td>
</tr>
<tr>
<td>fe51_sub</td>
<td>37</td>
<td>45</td>
<td>44.4</td>
<td>0.0</td>
<td>0.35</td>
<td>0.15</td>
</tr>
<tr>
<td>fe51_mul</td>
<td>124</td>
<td>617</td>
<td>51.5</td>
<td>2.7</td>
<td>17.67</td>
<td>14.52</td>
</tr>
<tr>
<td>fe51_mul_121666</td>
<td>57</td>
<td>166</td>
<td>44.6</td>
<td>4.8</td>
<td>1.42</td>
<td>0.88</td>
</tr>
<tr>
<td>fe51_sq</td>
<td>94</td>
<td>432</td>
<td>51.4</td>
<td>3.2</td>
<td>9.67</td>
<td>7.57</td>
</tr>
<tr>
<td>x25519_scalar_mul</td>
<td>1153</td>
<td>5280</td>
<td>50.6</td>
<td>2.5</td>
<td>OM</td>
<td>2815.16</td>
</tr>
</tbody>
</table>

Note: Only the Montgomery Ladderstep part is verified.

They all have input range assumptions given in the comments as pre-conditions.

The verification of these three functions failed with these given pre-conditions. It turns out that the given range assumptions may cause unexpected overflows in the implementations. These overflows then result in wrong returned values. Using the output of the CRYPTO LINE verification tool, we succeeded in locating the instructions with unexpected overflows. Counterexamples were also constructed. A counterexample of `felem_diff_128_64` shows that the unexpected overflow really happens when it is invoked by `point_double`

We reported our findings with the counterexamples to the OpenSSL developer community. The community confirmed that the overflow in `felem_diff_128_64` is a bug. They then fixed it in the commit 13fbc1. Besides OpenSSL 1.1.1, this bug is hidden in various releases including 1.0.0, 1.0.2 etc. For `felem_mul` and `felem_square`, the community confirmed that the range assumptions written in the comments were wrong. New range assumptions were also given from the community.

The new implementation of `felem_diff_128_64` and the new range assumptions of `felem_mul` and `felem_square` have been verified by our approach. The verification of new `felem_diff_128_64` takes less than 5 seconds on both M1 and M2. `felem_mul` and `felem_square` with new assumptions take around 320 and 80 seconds respectively on both machines.

### E. Remark on Compiler Optimization

In the experiment, we found that there are vectorized instructions in the assembly output of `x25519_scalar_mult` from Clang, even though the source code is sequential. It turns out that compilers like Clang are able to perform surprisingly non-trivial optimizations. It can vectorize a fragment of sequential code. In the case of `x25519_scalar_mult`, two sequential additions $a_1 + b_1$ and $a_2 + b_2$ in C code are optimized to a vector addition $a+b$ on vectors $a$ and $b$, where $a$ contains $a_1, a_2$ and $b$ similarly. The vector addition is further assembled to a vectorized assembly instruction if the underlying architecture supports. This means the two sequential C statements will be executed simultaneously in the binary executable.

### VI. RELATED WORK

To the best of our knowledge, this work presents the first attempt to verify existing cryptographic C code automatically. We compare our approach with others in three categories.

### A. General-Purpose C Verification

Numerous techniques and well-developed automatic tools such as [9]–[12], [17]–[21] are available for verifying C code. The annual Competition on Software Verification (SV-COMP) is a showcase for them. We have tried CPA-SEQ [9], PESCO [10] and UAUTOMIZER [11] from the top three winning in Overall category in SV-COMP 2019 [22] to verify our 8-line motivating example. As shown, these general-purpose verification tools are not very suitable for verifying bit-precise non-linear algebraic properties in cryptographic programs. We specially mention SMACK [12] since it works similarly as our approach. It converts LLVM IR programs into Boogie programs [23], then chooses various verifiers for Boogie to perform verification. However, Boogie is an industrial-strength verification tool, we succeeded in locating the instructions with unexpected overflows. Counterexamples were also constructed. A counterexample of

[https://sv-comp.sosy-lab.org/](https://sv-comp.sosy-lab.org/)
not designed for cryptographic programs and there is no verifier developed for that purpose. FRAMA-C [24] allows to verify algebraic properties by combining SMT-solving and interactive theorem proving. However, the complicated algebraic properties involving large numbers are difficult or even impossible to be specified in these general-purpose C verification tools.

B. Verifying Cryptographic C Code

gfverif [25] is an automatic tool used to verify a C implementation of the Montgomery Ladderstep in Curve25519. It needs to re-implement existing C programs using its constructs before verification. gfverif verifies fewer programs than our approach because its constructs are more limited. For example, it does not support 128-bit integers or algebraic properties involving variable modulus. It cannot verify our motivating example in Section I either. Cryptol/SAW [26] automatically verifies several cryptographic implementations in C and Java against their reference implementations. However, the reference implementations are not proven correct. F* [27] and Vale [28] implement arithmetic operations in their languages, and verify the results using SMT solving and manual proofs. Fiat-Crypto [29] is a project that tries to synthesize correct-by-construction C code for cryptographic primitives. But its verification relies on manual proofs using Coq [30]. A collection of hash functions, random number generators and other operations [31]–[38] are formally and manually verified using proof assistants like Coq. Note that interactive theorem proving costs much more human efforts than our approach. And our approach is able to construct counterexamples when verification fails, while manual approaches cannot.

C. Verifying Cryptographic Assembly Code

In [39], the authors verified a hand-optimized assembly implementation of the Montgomery Ladderstep in Curve25519 using SMT solvers and Coq. They have to annotate programs extensively and manually. If SMT solving fails, human verifiers have to use Coq to manually fill the gap. The work [40] models cryptographic assembly programs with a domain-specific language BVCRYPTOLINE. Programs in BVCryptoline can be verified automatically by a certified approach. Extending BVCryptolINE, CRYPTO LINE [13] is equipped with automated tools for translation from assembly code to CRYPTO LINE. Although the verification is not certified, it is much faster. Our approach is based on CRYPTO LINE. Compared to them, our approach works at a higher level and supports features like pointer arithmetic.

VII. Conclusion

We have presented an automated approach to translation and verification of arithmetic functions in cryptographic C programs. The case studies on real-world implementations in OpenSSL suggest the applicability and scalability of our approach. We were assisted greatly by the useful comments of OpenSSL developers in our experiments.

There are three obvious future directions. First, more translation heuristics can be developed to ease the verification process. Second, specifications are written at the CRYPTO LINE level for the moment. It requires human verifiers to have knowledge about the translation. Another direction is to design a specification language that allows verifiers to write pre- and post-conditions at C source code. Finally, we have only worked up to one iteration of an innermost loop (the Montgomery Ladderstep). We could elevate the verification to a higher level of the cryptographic primitive (here, a point multiplication on a curve) by checking loop invariants.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their valuable comments and suggestions. This work is supported by Academia Sinica under the Grant Numbers AS-IA-104-M01 and AS-TP-106-M06; the Guangdong Science and Technology Department under the Grant Number 2018B010107004; the Ministry of Science and Technology of Taiwan under Grant Numbers 105-2221-E-001-014-MY3, 107-2221-E-001-004, 108-2221-E-001-009-MY2, 108-2221-E-001-010-MY3; and the National Natural Science Foundation of China under the Grant Numbers 61802259 and 61836005.

REFERENCES


