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.

‡Corresponding author

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 $TB^{-1} \bmod N$ 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 $TB^{-1} \bmod N$ 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.

Pre-condition: $0 < N < B$ with $N \equiv 1 \pmod 2$, $0 < N' < B$ with $NN' + 1 \equiv 0 \pmod B$, and $0 \le T < BN$
Post-condition: $\mathrm{REDC}^{-}(N, N', T) \times B \equiv T \pmod N$
function $\mathrm{REDC}^{-}(N, N', T)$
  $m \leftarrow ((T \bmod B)N') \bmod B$
  $t \leftarrow (T + mN)/B$
  return t

Figure 1. Montgomery Reduction

Table I
VERIFYING FUNCTION REDC() WITH SELECTED TOOLS

Configuration	Output
CPA-SEQ (SV-COMP2019 version[1])	
`-default -heap 10000M`	TIMEOUT
`-svcomp19 -heap 10000M`	TIMEOUT
PESCO (SV-COMP2019 version)	
`-svcomp19-pesco -heap 10000M -stack 2048k`	TIMEOUT
`-svcomp19-pesco-linear -heap 10000M -stack 2048k`	TIMEOUT
UAUTOMIZER (SV-COMP2019 version)	
`--architecture 64bit`	TIMEOUT
SMACK (version 1.9.3)	
`--verifier boogie`	FALSE
`--bit-precise --verifier boogie`	TIMEOUT
`--verifier corral`	TIMEOUT
`--bit-precise --verifier corral`	TIMEOUT
`--verifier symbooglix`	FALSE
`--bit-precise --verifier symbooglix`	UNKNOWN[2]
`--verifier duality`	TIMEOUT
`--bit-precise --verifier duality`	TIMEOUT

```
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 + m * N) >> 32;
  return t;
}
```

The inputs N, Np and T are 64-bit integers. Given $\texttt{N} < 2^{31}$ and pre-conditions in Fig. 1, we would like to verify whether $\texttt{REDC(N,Np,T)} \times \texttt{B} \equiv \texttt{T} \pmod{\texttt{N}}$. 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.

[1]The SV-COMP2019 versions of CPA-SEQ, PESCO and UAUTOMIZER are downloadable at https://sv-comp.sosy-lab.org/2019/systems.php

[2]Due to a bug of the tool (see the issue at https://github.com/smackers/smack/issues/427), SMACK did claim that REDC() was verified. The real output is UNKNOWN.

In order to verify cryptographic C programs, new techniques are needed. In [13], the modeling language CRYPTOLINE and its tool for verifying cryptographic *assembly* programs are proposed. We leverage the work by translating LLVM IR programs to CRYPTOLINE 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 CRYPTOLINE program.
3) Specify properties about the C program in the generated CRYPTOLINE program.
4) Verify whether the CRYPTOLINE program conforms to the specification with the CRYPTOLINE 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 non-linear multivariate polynomial modulo equations over 45 (= $(5 + 4) \times 5$) 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[3]. 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 CRYPTOLINE program. CRYPTOLINE is

[3]https://github.com/openssl/openssl/commit/13fbce17fc9f02e2401fc3868f3f8e02d6647e5f

designed for cryptographic assembly programs, not LLVM IR. In particular, CRYPTOLINE does not allow pointer arithmetic found in LLVMCRYPTO. Such LLVMCRYPTO features are translated to CRYPTOLINE automatically. A soundness theorem is established for our translator. It guarantees that no bug will be missed if a program is verified by our approach.

Our case studies include 38 C functions from NIST P-224, P-256, P-521, and Curve25519 in OpenSSL. To the best of our knowledge, this is the first automated approach which can verify the correctness of these cryptographic C programs. We also expose a bug and two incorrect input assumptions in the NIST P-521 implementations.

In the rest of the paper, after introducing LLVMCRYPTO in Section II, we review CRYPTOLINE in Section III. The translation is addressed in Section IV, while case studies are detailed in Section V. Conclusion comes in Section VII.

II. LLVMCRYPTO – A SUBSET OF LLVM IR

LLVM [14] is an open-source project for modular compilation and related technologies. It is based on a code representation called LLVM IR (for LLVM Intermediate Representation). Very roughly, any C program is represented in LLVM IR for code transformations and optimizations.

For cryptographic C programs, the full LLVM IR is not necessary. We examine the intermediate representations generated after architecturally independent optimizations in Clang, and identify a useful subset of LLVM IR for cryptographic C programs, called LLVMCRYPTO. In this section, we present LLVMCRYPTO and give its formal semantics.

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 $n \in \mathbb{N}^+$. $a \div b$ and $a \bmod b$ denote the quotient and non-negative remainder of a divided by b. That is, we have $a = b \times (a \div b) + (a \bmod b)$ with $0 \leq a \bmod b < b$. Let $f : A \rightarrow B$ be a function. For $a \in A$ and $b \in B$, define the function $f[a \leftarrow b] : A \rightarrow B$ by

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

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 ℓ ($\mathit{Argv}(\ell)$) can be a vector variable or a sequence of ℓ numbers. In LLVMCRYPTO, operands and the result of each instruction can be 64- or

$$
\begin{array}{rll}
\mathit{Num} & ::= 0 \mid 1 \mid 2 \mid \cdots & \mathit{Var} ::= x \mid y \mid z \mid \cdots \\
\mathit{Ptr} & ::= p \mid q \mid \cdots & \mathit{Width} ::= 64 \mid 128 \\
\mathit{Arg} & ::= \mathit{Var} \mid \mathit{Num} & \mathit{Argv}(\ell) ::= \mathit{Var} \mid \mathit{Num}^{\ell} \\
\mathit{Inst} & ::= \mathit{Var} = \texttt{add}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{addv}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Argv}(\ell)\ \mathit{Argv}(\ell) & \\
& \mid \mathit{Var} = \texttt{sub}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{subv}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Argv}(\ell)\ \mathit{Argv}(\ell) & \\
& \mid \mathit{Var} = \texttt{mul}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{mulv}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Argv}(\ell)\ \mathit{Argv}(\ell) & \\
& \mid \mathit{Var} = \texttt{shl}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Num} & \\
& \mid \mathit{Var} = \texttt{lshr}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Num} & \\
& \mid \mathit{Var} = \texttt{and}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{load}\ \mathit{Width}\ \mathit{Ptr} & \\
& \mid \mathit{Var} = \texttt{loadv}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Ptr} & \\
& \mid \texttt{store}\ \mathit{Width}\ \mathit{Arg}\ \mathit{Ptr} & \\
& \mid \texttt{storev}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Argv}(\ell)\ \mathit{Ptr} & \\
& \mid \mathit{Ptr} = \texttt{geteltptr}\ \mathit{Width}\ \mathit{Ptr}\ \mathit{Num} & \\
& \mid \mathit{Ptr} = \texttt{geteltptrv}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Ptr}\ \mathit{Num}\ \mathit{Num} & \\
& \mid \mathit{Var} = \texttt{trunc}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{zext}\ \mathit{Arg} & \\
& \mid \mathit{Var} = \texttt{insertelt}\ <\ell\,\texttt{x}\,\mathit{Width}>\ \mathit{Argv}(\ell)\ \mathit{Arg}\ \mathit{Num} & \\
\mathit{Prog} & ::= \mathit{Inst};\ \mid \mathit{Inst}; \mathit{Prog} &
\end{array}
$$

Figure 2. The Syntax of LLVMCRYPTO

128-bit values, specified by the instruction syntactically. For instance, the instruction $y = \texttt{add}\ 64\ a_1\ a_2$ adds the 64-bit operands a_1, a_2 together, and assigns the sum to the 64-bit variable y. On the other hand, $y = \texttt{add}\ 128\ a_1\ a_2$ has 128-bit operands and result.

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

Two bitwise shifting instructions are defined in LLVMCRYPTO. $y = \texttt{shl}\ w\ a\ n$ 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 `lshr` on the other hand shifts to the right. The bitwise AND instruction is $y = \texttt{and}\ w\ a_1\ a_2$.

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

One can obtain the pointer to an element of a vector stored in memory. $q = \texttt{geteltptr}\ w\ p\ n$ 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 ℓ, $q = \texttt{geteltptrv}\ <\ell\,\texttt{x}\,w>\ p\ n_1\ n_2$ sets q to the pointer at the n_2-th element of the n_1-th vector designated by p.

The instruction $y = \texttt{trunc}\ a$ truncates the 128-bit value a

to the low 64 bits and stores the result in the 64-bit variable y. $y = \texttt{zext}\ a$ extends the 64-bit operand a to 128 bits.

Finally, the instruction $y = \texttt{insertelt}\ \texttt{<}\ell\,\texttt{x}\,w\texttt{>}\ a_1\ a_2\ k$ assigns to y the ℓ-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 `felem_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_{out}.

$$
\begin{array}{lll}
1: & v_0 & = \texttt{sub}\ 64\ 4611686018427387872\ y_0; \\
2: & v_0' & = \texttt{add}\ 64\ v_0\ x_0; \\
3: & q_0 & = \texttt{geteltptr}\ 64\ p_{out}\ 0; \\
4: & \multicolumn{2}{l}{\texttt{store}\ 64\ v_0'\ q_0;}
\end{array}
$$

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 q_0 pointing to the 0-th 64-bit element of the memory designated by p_{out}. The calculation result v_0' is stored to the memory cell pointed by q_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 $\sigma \in \mathcal{V} \triangleq (\mathit{Var} \cup \mathit{Ptr}) \to \mathbb{N}$ be a *valuation*, and $m \in \mathcal{M} \triangleq \mathbb{N} \to [2^{64}]$ a *memory state*. $\mathcal{S} \triangleq \mathcal{V} \times \mathcal{M}$ 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 $[\![\bullet]\!]_\sigma$ for numbers, variables and pointers as follows.

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

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

More notations are needed for vectors. For $\ell \in \mathbb{N}^+$ and $v \in \mathit{Num}^\ell$, $v[i]$ denotes the i-th element of v when $i \in [\ell]$. We also use the variable $x[i]$ for the i-th element of the vector variable $x \in \mathit{Var}$. Given a valuation σ and $n \in \mathbb{N}$, the notation $\sigma[a_i \leftarrow b_i]_{i=0}^n$ is short for $\sigma[a_0 \leftarrow b_0] \cdots [a_n \leftarrow b_n]$. The semantics of $y = \texttt{addv}\ \texttt{<}\ell\,\texttt{x}\,w\texttt{>}\ a_1\ a_2$ 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`, `lshr` 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:

$$
\begin{aligned}
m_{64}(n) &\triangleq m(n) \\
m_{64}[n \leftarrow v] &\triangleq m[n \leftarrow v \bmod 2^{64}]
\end{aligned}
$$

$m_{64}(n)$ 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:

$$
\begin{aligned}
m_{128}(n) &\triangleq m(n+1) \times 2^{64} + m(n) \\
m_{128}[n \leftarrow v] &\triangleq m[n \leftarrow v^L][n+1 \leftarrow v^H]
\end{aligned}
$$

where $v^L = v \bmod 2^{64}$ and $v^H = (v \div 2^{64}) \bmod 2^{64}$. 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 $y = \texttt{load}\ w\ p$ 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 $\mathit{size}(w) \triangleq w \div 64$ for the number of memory cells needed for w-bit values. By $y = \texttt{loadv}\ \texttt{<}\ell\,\texttt{x}\,w\texttt{>}\ p$, the vector variable y is updated with ℓ 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 $[\![p]\!]_\sigma + [\![n]\!]_\sigma \times \mathit{size}(w)$. This is exactly what $q = \texttt{geteltptr}\ w\ p\ n$ computes. `geteltptrv` is defined similarly when p points to a vector of vectors.

The semantics of instructions `trunc` and `zext` is straightforward. Finally, $y = \texttt{insertelt}\ \texttt{<}\ell\,\texttt{x}\,w\texttt{>}\ a_1\ a_2\ k$ copies

$$
\begin{array}{lcll}
(\sigma, m) & \xrightarrow{y\ \texttt{= add}\ w\ a_1\ a_2} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow ([\![a_1]\!]_\sigma + [\![a_2]\!]_\sigma) \bmod 2^w] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= addv <}\ell \,\texttt{x}\, w\texttt{>}\ a_1\ a_2} & (\sigma', m) & \text{where } \sigma' = \sigma[y[i] \leftarrow ([\![a_1[i]]\!]_\sigma + [\![a_2[i]]\!]_\sigma) \bmod 2^w]_{i=0}^{\ell-1} \\
(\sigma, m) & \xrightarrow{y\ \texttt{= shl}\ w\ a\ n} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow ([\![a]\!]_\sigma \times 2^{[\![n]\!]_\sigma}) \bmod 2^w] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= lshr}\ w\ a\ n} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow [\![a]\!]_\sigma \div 2^{[\![n]\!]_\sigma}] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= and}\ w\ a_1\ a_2} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow [\![a_1]\!]_\sigma \text{ band } [\![a_2]\!]_\sigma] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= load}\ w\ p} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow m_w([\![p]\!]_\sigma)] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= loadv <}\ell \,\texttt{x}\, w\texttt{>}\ p} & (\sigma', m) & \text{where } \sigma' = \sigma[y[i] \leftarrow m_w([\![p]\!]_\sigma + i \times \mathit{size}(w))]_{i=0}^{\ell-1} \\
(\sigma, m) & \xrightarrow{\texttt{store}\ w\ a\ p} & (\sigma, m') & \text{where } m' = m_w[[\![p]\!]_\sigma \leftarrow [\![a]\!]_\sigma] \\
(\sigma, m) & \xrightarrow{\texttt{storev <}\ell \,\texttt{x}\, w\texttt{>}\ a\ p} & (\sigma, m') & \text{where } m' = m_w[[\![p]\!]_\sigma + i \times \mathit{size}(w) \leftarrow [\![a[i]]\!]_\sigma]_{i=0}^{\ell-1} \\
(\sigma, m) & \xrightarrow{q\ \texttt{= geteltptr}\ w\ p\ n} & (\sigma', m) & \text{where } \sigma' = \sigma[q \leftarrow [\![p]\!]_\sigma + [\![n]\!]_\sigma \times \mathit{size}(w)] \\
(\sigma, m) & \xrightarrow{q\ \texttt{= geteltptrv <}\ell \,\texttt{x}\, w\texttt{>}\ p\ n_1\ n_2} & (\sigma', m) & \text{where } \sigma' = \sigma[q \leftarrow [\![p]\!]_\sigma + [\![n_1]\!]_\sigma \times \ell \times \mathit{size}(w) + [\![n_2]\!]_\sigma \times \mathit{size}(w)] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= trunc}\ a} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow [\![a]\!]_\sigma \bmod 2^{64}] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= zext}\ a} & (\sigma', m) & \text{where } \sigma' = \sigma[y \leftarrow [\![a]\!]_\sigma] \\
(\sigma, m) & \xrightarrow{y\ \texttt{= insertelt <}\ell \,\texttt{x}\, w\texttt{>}\ a_1\ a_2\ k} & (\sigma', m) & \text{where } \sigma' = \sigma[y[i] \leftarrow [\![a_1[i]]\!]_\sigma \bmod 2^w]_{i=0}^{\ell-1}[y[k] \leftarrow [\![a_2]\!]_\sigma \bmod 2^w]
\end{array}
$$

Figure 3. Semantics of LLVMCRYPTO

the vector a_1 of ℓ 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 CRYPTOLINE

CRYPTOLINE [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

CRYPTOLINE 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 CRYPTOLINE. Fig. 4 gives the syntax of the language.

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

A CRYPTOLINE *state* models the current values of variables and flags (*clFlag*). Set is the assignment statement and Cset is the conditional assignment. Carry and borrow flags are explicit in CRYPTOLINE. Add $b\ x\ u\ v$ sets the sum of u and v to x with carry in b. Adc is the addition-with-carry statement. Sub and Sbb are subtraction and subtraction-with-borrow statements, respectively. Full multiplication Mulf $x\ y\ u\ v$ 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 CRYPTOLINE program goes into the *error state*.

$$
\begin{array}{rcll}
\mathit{clFlag} & ::= & b \mid c \mid d \mid \cdots & \\
\mathit{clExp} & ::= & \mathit{Arg} \mid \mathit{clExp} + \mathit{clExp} \mid \mathit{clExp} - \mathit{clExp} & \\
& & \mid \mathit{clExp} * \mathit{clExp} & \\
\mathit{clPred} & ::= & \mathit{clExp} = \mathit{clExp} \mid \mathit{clExp} \equiv \mathit{clExp} \ \mathsf{mod}\ \mathit{clExp} & \\
& & \mid \mathit{clExp} < \mathit{clExp} \mid \mathit{clExp} \leq \mathit{clExp} \mid \mathit{clPred} \wedge \mathit{clPred} & \\
\mathit{clStmt} & ::= & \mathsf{Set}\ \mathit{Var}\ \mathit{Arg} & \mid \mathsf{Cset}\ \mathit{Var}\ \mathit{clFlag}\ \mathit{Arg}\ \mathit{Arg} \\
& & \mid \mathsf{Add}\ \mathit{clFlag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} & \\
& & \mid \mathsf{Adc}\ \mathit{clFlag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg}\ \mathit{clFlag} & \\
& & \mid \mathsf{Sub}\ \mathit{clFlag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} & \\
& & \mid \mathsf{Sbb}\ \mathit{clFlag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg}\ \mathit{clFlag} & \\
& & \mid \mathsf{Mulf}\ \mathit{Var}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} & \mid \mathsf{And}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} \\
& & \mid \mathsf{Shl}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Num} & \mid \mathsf{Split}\ \mathit{Var}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Num} \\
& & \mid \mathsf{Assert}\ \mathit{clPred} & \mid \mathsf{Assume}\ \mathit{clPred} \\
\mathit{clProg} & ::= & \epsilon \mid \mathit{clStmt}; \mathit{clProg} &
\end{array}
$$

Figure 4. CRYPTOLINE 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, CRYPTOLINE 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 CRYPTOLINE, 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 CRYPTOLINE program is simply a sequence of CRYPTOLINE statements followed by semicolons.

B. Verification with Specifications

In addition to programs, CRYPTOLINE allows to specify pre- and post-conditions using predicates (*clPred*). Pre- and post-conditions together compose *specifications*. Given a CRYPTOLINE 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 CRYPTOLINE verification tool checks if a CRYPTOLINE 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 CRYPTOLINE program generated from `felem_diff64` is

$$
\begin{aligned}
&((addr\_p\_0 < x_0 + 2^{62}) \wedge \cdots \wedge (addr\_p\_8 < x_8 + 2^{62}))\\
&\wedge\ (radix58(x_0, x_1, \ldots, x_8) - radix58(y_0, y_1, \ldots, y_8)\\
&\qquad \equiv radix58(addr\_p\_0, \ldots, addr\_p\_8) \ \mathsf{mod}\ p_{521})
\end{aligned}
$$

where $radix58(a_0, a_1, \ldots, a_8) = a_0 + a_1 \times 2^{58\times1} + a_2 \times 2^{58\times2} + \cdots + a_8 \times 2^{58\times8}$ 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 CRYPTOLINE

Given an LLVMCRYPTO program, we first translate it into a CRYPTOLINE program in order to verify with the CRYPTOLINE 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 CRYPTOLINE 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 gap by representing memory addresses symbolically then using these symbols to translate pointers.

Assume $\mathcal{SV}ar = \{\mathbb{p}, \mathbb{q}, \ldots\}$ is a set of *symbolic variables*. $\mathbb{A} \triangleq \mathcal{SV}ar \times \mathbb{Z}$ is called the set of *symbolic (memory) addresses*. A symbolic address $(\mathbb{p}, o) \in \mathbb{A}$ represents the memory address having an offset o from the memory address represented by $(\mathbb{p}, 0) \in \mathbb{A}$. In LLVMCRYPTO, a pointer is only allowed to be added with a constant offset. Hence we define the addition $+_{\mathbb{A}} : \mathbb{A} \times \mathbb{Z} \to \mathbb{A}$ as $(\mathbb{p}, o_1) +_{\mathbb{A}} o_2 \triangleq (\mathbb{p}, o_1 + o_2)$, where $(\mathbb{p}, o_1) \in \mathbb{A}$ 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:

$$
\begin{aligned}
q &= \texttt{geteltptr}\ 64\ p\ i;\\
y &= \texttt{load}\ 64\ q;
\end{aligned}
$$

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 $(\mathbb{p}, o_p)$, our translation algorithm is sufficient to calculate n_q's symbolic address as $(\mathbb{p}, o_q) = (\mathbb{p}, o_p) +_{\mathbb{A}} i$. 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 $\mathbb{A}$. 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 INSTTOCLPROG(pt, s) translates s into a sequence cp of CRYPTOLINE statements. It returns a pair $\langle pt', cp\rangle$, 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 INSTTOCLPROG() to each instruction. We summarize the translation for 64-bit LLVMCRYPTO instructions as in Table II.

The translation of arithmetic instructions is straightforward. For example, the instruction `add` is translated to the statement Add. Note that carries are not present in LLVMCRYPTO. Hence the introduced carry flag is discarded using a fresh name d. But the value of d does indicate the presence of overflow when executing the `add` instruction. Since the instruction `add` does not change pointers, pt remains the same after executing `add`. The instruction y = `addv` $<\ell \times 64>$ a_1 a_2 adds two vectors a_1 and a_2 of length ℓ. It is equivalent to ℓ `add` instructions on each pair of $a_1[i]$ and $a_2[i]$. Hence we have it translated to ℓ Add statements.

The translation of bitwise shifting is a little subtle. The instruction `shl` has similar semantics as Shl, except that Shl may cause an error that is undesired by `shl`. To avoid that,

Table II
SUMMARY OF INSTTOCLPROG(pt, s), THE 64-BIT CASE

Instruction s	**Output** $\langle pt', cp\rangle$
y = `add` 64 a_1 a_2	$\langle pt,$ Add $d\ y\ a_1\ a_2;\rangle$
y = `addv` < ℓ x 64 > a_1 a_2	$\langle pt,$ sequence of Add's$\rangle$
y = `shl` 64 a n	$\langle pt,$ Split $z^d\ t\ a\ (64-n);$ Shl $y\ t\ n;\rangle$
y = `lshr` 64 a n	$\langle pt,$ Split $y\ z^d\ a\ n;\rangle$
y = `and` 64 a_1 a_2	$\langle pt,$ And $y\ a_1\ a_2;\rangle$
y = `load` 64 p	$\langle pt,$ Set $y\ (\!|pt(p)|\!)_V;\rangle$
y = `loadv` < ℓ x 64 > p	$\langle pt,$ sequence of Set's$\rangle$
q = `geteltptr` 64 p n	$\langle pt[q \leftarrow pt(p) +_{\mathbb{A}} n],\ \epsilon\rangle$
y = `trunc` a	$\langle pt,$ Set $y\ a^L;\rangle$
y = `zext` a	$\langle pt,$ Set $y^L\ a;$ Set $y^H\ 0;\rangle$
y = `insertelt` < ℓ x 64 > a_1 a_2 k	$\langle pt,$ sequence of Set's$\rangle$

the high n bits of a are discarded first by 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 `lshr`. The translation for `and` is trivial.

To translate an instruction involving memory, the memory cell is referred to using a CRYPTOLINE variable. Assume that we have a one-to-one function $(\!|\bullet|\!)_V : \mathbb{A} \rightarrow \mathit{Var}$. It converts each symbolic address into a CRYPTOLINE variable. For example, $(\!|(\mathbb{p}, 1)|\!)_V = \mathit{addr\_p\_1}$ in our implementation. Now the translation for y = `load` 64 p is straightforward. y is assigned with the variable $(\!|pt(p)|\!)_V$, which represents the value in the memory cell indexed by p. The vector version `loadv` is translated in the same way as `addv`. Again, `load` does not change pointers, pt remaining unchanged. The instructions `store` and `storev` are treated similarly, omitted in Table II.

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

Given any 128-bit LLVMCRYPTO variable y, two CRYPTOLINE 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 `trunc` and `zext` with Set statements. As well, Set statements are used to translate `insertelt` by copying each element of a_1 to y with the k-th element $y[k]$ assigned by a_2.

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

function PROGTOCLPROG($prog$)
  Construct pt_0 with $prog$
  Let $prog = s_1; s_2; \cdots; s_n$;
  for $i \leftarrow 1$ **to** n **do**
    $\langle pt_i, cp_i\rangle \leftarrow$ INSTTOCLPROG(pt_{i-1}, s_i)
  return $(cp_1\ cp_2\ \cdots\ cp_n)$

Figure 5. Translation of LLVMCRYPTO Programs

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 INSTTOCLPROG(). The algorithm is depicted in Fig. 5. We first construct the initial pointer table pt_0 as a mapping that maps each $p_j \in \mathit{Ptr}_U$ in $prog$, to the symbolic address $(\mathbb{p}_j, 0)$. All $\mathbb{p}_j$'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 CRYPTOLINE 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 CRYPTOLINE program.

Example (continued)**.** Given the whole LLVMCRYPTO program, we know that p_{out} is an undefined pointer. We let $pt_0(p_{out}) = (\mathbb{p}, 0)$ 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 CRYPTOLINE statements, but updates q_0 in pt_3 with $pt_3(q_0) = pt_2(p_{out}) +_{\mathbb{A}} 0 = (\mathbb{p}, 0)$. Let $(\!|(\mathbb{p}, 0)|\!)_V = \mathit{addr\_p\_0}$. The LLVMCRYPTO program fragment is translated into the following CRYPTOLINE program fragment by our algorithm:

$1a:$ Sub $d_0\ v_0\ 4611686018427387872\ y_0;$
$2a:$ Add $d_1\ v_0'\ v_0\ x_0;$
$4a:$ Set $\mathit{addr\_p\_0}\ v_0';$

C. Soundness

Given an initial state (σ_0, m_0), an LLVMCRYPTO program is *well-formed* if (1) all its undefined variables and undefined pointers have their values in σ_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 π_j contains one* base address *designated by an undefined pointer $p_j \in \mathit{Ptr}_U$. Let $pt_0(p_j) = (\mathbb{p}_j, 0)$. Every address in π_j is* uniquely *represented by the symbolic address $(\mathbb{p}_j, o)$ for some $o \in \mathbb{Z}$ 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 [15].

Given an LLVMCRYPTO program *prog* with initial state (σ_0, m_0), we use (σ_i, m_i) to denote the state after executing the first i instructions. That is, for the i-th instruction s_i of *prog*, we have $(\sigma_{i-1}, m_{i-1}) \xrightarrow{s_i} (\sigma_i, m_i)$. We define a *simulation* relation $\preccurlyeq$ between LLVMCRYPTO states (σ, m) and CRYPTOLINE states ρ. $(\sigma, m) \preccurlyeq \rho$ reads as (σ, m) is simulated by ρ. $(\sigma, m) \preccurlyeq \rho$ holds if the values of variables and the content of memory in (σ, m) are correctly projected into ρ. For example, given a 128-bit LLVMCRYPTO variable x, its corresponding 64-bit representations x^L and x^H in CRYPTOLINE 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* LLVMCRYPTO *program prog with initial state* (σ_0, m_0). *The generated* CRYPTOLINE *program clprog* $=$ PROGTOCLPROG$(prog) = (cp_1 \cdots cp_n)$ *satisfies:*

(i) *for all* $i \in [0, n]$, *there exists a* CRYPTOLINE *state* ρ_i *of clprog such that* $(\sigma_i, m_i) \preccurlyeq \rho_i$ *;*

(ii) *for all* $i \in [1, n]$ *and* ρ *with* $(\sigma_{i-1}, m_{i-1}) \xrightarrow{s_i} (\sigma_i, m_i)$ *and* $(\sigma_{i-1}, m_{i-1}) \preccurlyeq \rho$, *there exists* ρ' *such that* $\rho \xrightarrow{cp_i} \rho'$ *and* $(\sigma_i, m_i) \preccurlyeq \rho'$.

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

With Theorem 1, assume that the given LLVMCRYPTO program *prog* has n instructions. If a property P_{llvm} does not hold in the final state (σ_n, m_n), then there must exist a CRYPTOLINE state ρ_n (with $(\sigma_n, m_n) \preccurlyeq \rho_n$) of the generated program and a trace to ρ_n, such that the corresponding property P_{cl} does not hold in ρ_n either. In other words, if the verification tool verifies that P_{cl} holds in all possible final states ρ_n's (even if $(\sigma_n, m_n) \not\preccurlyeq \rho_n$) along all possible execution traces of the generated CRYPTOLINE program, then P_{llvm} is guaranteed to hold in (σ_n, m_n) of the input *prog*. Therefore, if our verification result shows that the generated CRYPTOLINE program is correct with respect to the specification, it implies that the input LLVMCRYPTO 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 CRYPTOLINE programs verified due to limitations of the CRYPTOLINE verification tool. In our first implementation of the translator, we found that it took large amounts of human work to verify the generated CRYPTOLINE 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 specific bitwise shifting, for special `and`, for overflow/underflow, and for the `and`-after-`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 `add`, `sub` and `mul`, we introduce carry/borrow flags that indicate the presence of overflow/underflow in the input LLVMCRYPTO instructions. For example, when translating y `=` `add` $64\ a_1\ a_2$, the statement Add $d\ y\ a_1\ a_2$ is used. A new flag d is introduced that indicates the presence of overflow in `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 CRYPTOLINE tool to deduce $d = 0$ automatically and use this information to verify specified properties. Our heuristic hence automatically inserts the following two CRYPTOLINE statements for each flag d of such a kind during translation:

Assert $d = 0$;   Assume $d = 0$;

The Assert statement tells CRYPTOLINE to check whether d is 0. If it is, then Assume 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 `shl` for the same reason, to check whether the value of z^d equals 0.

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

y_1 `=` `lshr` $64\ a\ 51$;
y_2 `=` `and` $64\ a$ 0x7FFFFFFFFFFFF;

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:

Split $y_1\ z^d\ a\ 51$;
And $y_2\ a$ 0x7FFFFFFFFFFFF;

But CRYPTOLINE 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:

Assert $z^d = y_2$;   Assume $z^d = y_2$;

Note that in practice, the instructions `lshr` and `and` may not be adjacent. They may not have exactly the same a as operands. And several pairs of `lshr` and `and` may even interleave. It makes this `and`-after-`lshr` pattern more complicated. The implemented simple heuristic only relates `and` to the previous `lshr`. 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 CryptoLine automatically. The very few instructions not supported by llvmCrypto required manual translation. The generated CryptoLine 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 membership solver, respectively, for the CryptoLine verification 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 $\mathbb{Z}_p$: $p = 2^{255} - 19$ 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: $X1 \times X5 \times (X2 \times Z3 - Z2 \times X3)^2 \equiv Z5 \times (X2 \times X3 - Z2 \times Z3)^2 \bmod (2^{255} - 19)$. Each of $X1$, $X2$, $X3$, $X5$, $Z2$, $Z3$ and $Z5$ 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 CryptoLine code. The "diff-*" columns show the percentage of manual modifications in each CryptoLine 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 CryptoLine. 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 CryptoLine. 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
EXPERIMENT RESULTS

function	loc-ir	loc-cl	diff-0 (%)	diff-h (%)	T_1 (s)	T_2 (s)
ecp_nistp224.c						
felem_diff	30	40	40.0	0.0	0.40	0.18
felem_diff_128_64	30	60	26.7	0.0	0.85	0.73
felem_mul	60	298	49.0	0.0	9.64	8.80
felem_scalar	15	20	40.0	0.0	0.16	0.08
felem_square	43	193	47.7	0.0	1.96	1.20
felem_sum	22	24	33.3	0.0	0.23	0.10
widefelem_diff	54	112	25.0	0.0	2.75	2.67
felem_mul_reduce	99	493	58.4	9.3	73.47	71.02
felem_neg	47	145	49.0	9.0	1.07	0.57
felem_reduce	75	246	50.0	7.7	4.56	3.94
felem_square_reduce	82	388	60.3	11.9	64.83	61.45
widefelem_scalar	31	136	45.6	2.9	5.28	4.72
ecp_nistp256.c						
felem_diff	30	64	25.0	0.0	2.09	2.11
felem_scalar	16	74	43.2	0.0	0.55	0.31
felem_small_sum	26	44	18.2	0.0	0.39	0.31
felem_sum	22	40	20.0	0.0	0.34	0.27
smallfelem_mul	109	488	49.2	0.0	8.58	6.59
smallfelem_neg	22	36	22.2	0.0	0.25	0.12
felem_shrink	65	160	63.8	26.3	4.55	4.55
felem_small_mul	175	672	51.9	6.7	15.06	12.88
smallfelem_square	74	330	51.5	1.2	4.55	3.25
ecp_nistp521.c						
felem_diff64	61	81	44.4	0.0	0.84	0.49
felem_diff128	61	126	28.6	0.0	17.59	18.30
felem_neg	43	45	40.0	0.0	0.50	0.24
felem_scalar	43	45	40.0	0.0	0.97	0.95
felem_scalar64	35	45	40.0	0.0	1.12	1.15
felem_scalar128	36	162	44.4	0.0	3.61	3.61
felem_sum64	52	54	33.3	0.0	0.28	0.13
felem_reduce	145	317	53.0	17.0	2.29	1.36
felem_diff_128_64	70	126	28.6	0.0	-	-
felem_mul	289	1618	49.9	0.0	-	-
felem_square	158	892	51.1	0.0	-	-
curve25519.c						
fe51_add	32	30	33.3	0.0	0.18	0.07
fe51_sub	37	45	44.4	0.0	0.35	0.15
fe51_mul	124	617	51.5	2.7	17.67	14.52
fe51_mul121666	57	166	44.6	4.8	1.42	0.88
fe51_sq	94	432	51.4	3.2	9.67	7.57
x25519_scalar_mult[a]	1153	5280	50.6	2.5	OM	2815.16

[a]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 CRYPTOLINE 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 13fbce1. Besides OpenSSL 1.1.1, this bug is hidden in various releases including 1.1.0, 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 `addv` 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)[4] 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

[4]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, CRYPTOLINE [13] is equipped with automated tools for translation from assembly code to CRYPTOLINE. Although the verification is not certified, it is much faster. Our approach is based on CRYPTOLINE. 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 CRYPTOLINE 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

[1] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical realisation and elimination of an ECC-related software bug attack," in *CT-RSA 2012*, ser. LNCS, O. Dunkelman, Ed., vol. 7178. Springer, 2012, pp. 171–186.

[2] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[3] V. S. Miller, "Use of elliptic curves in cryptography," in *CRYPTO '85*, ser. LNCS, H. C. Williams, Ed., vol. 218. Springer, 1985, pp. 417–426.

[4] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Public Key Cryptography - PKC 2006*, ser. LNCS, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. Springer, 2006, pp. 207–228.

[5] The OpenSSH website. [Online]. Available: https://www.openssh.com/

[6] The OpenSSL website. [Online]. Available: https://www.openssl.org/

[7] OpenSSL committers. Accessed: 2019-05-10. [Online]. Available: https://www.openssl.org/community/committers.html

[8] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[9] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *CAV 2011*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 184–190.

[10] C. Richter and H. Wehrheim, "PeSCo: Predicting sequential combinations of verifiers - (competition contribution)," in *25 Years of TACAS: TOOLympics*, ser. LNCS, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., vol. 11429. Springer, 2019, pp. 229–233.

[11] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski, "Ultimate Automizer with an on-demand construction of Floyd-Hoare automata - (competition contribution)," in *TACAS 2017*, ser. LNCS, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 394–398.

[12] Z. Rakamaric and M. Emmi, "SMACK: decoupling source language details from verifier implementations," in *CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 106–113.

[13] A. Polyakov, M. Tsai, B. Wang, and B. Yang, "Verifying arithmetic assembly programs in cryptographic primitives (invited talk)," in *CONCUR 2018*, ser. LIPIcs, S. Schewe and L. Zhang, Eds., vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 4:1–4:16.

[14] The LLVM compiler infrastructure project. [Online]. Available: https://llvm.org/

[15] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS 2002*. IEEE Computer Society, 2002, pp. 55–74.

[16] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[17] P. Schrammel and D. Kroening, "2LS for program analysis - (competition contribution)," in *TACAS 2016*, ser. LNCS, M. Chechik and J. Raskin, Eds., vol. 9636. Springer, 2016, pp. 905–907.

[18] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *TACAS 2014*, ser. LNCS, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391.

[19] M. Y. R. Gadelha, F. R. Monteiro, L. C. Cordeiro, and D. A. Nicole, "ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution)," in *25 Years of TACAS: TOOLympics*, ser. LNCS, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., vol. 11429. Springer, 2019, pp. 209–213.

[20] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[21] B. Chimdyalwar, P. Darke, A. Chauhan, P. Shah, S. Kumar, and R. Venkatesh, "VeriAbs: Verification by abstraction (competition contribution)," in *TACAS 2017*, ser. LNCS, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 404–408.

[22] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *25 Years of TACAS: TOOLympics*, ser. LNCS, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., vol. 11429. Springer, 2019, pp. 133–155.

[23] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO 2005*, ser. LNCS, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 364–387.

[24] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[25] D. J. Bernstein and P. Schwabe, "gfverif: Fast and easy verification of finite-field arithmetic," 2016, http://gfverif.cryptojedi.org.

[26] A. Tomb, "Automated verification of real-world cryptographic implementations," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 26–33, 2016.

[27] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *CCS*. ACM, 2017, pp. 1789–1806.

[28] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *USENIX Security Symposium 2017*. USENIX Association, 2017, pp. 917–934.

[29] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2019, pp. 73–90.

[30] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.9*, Jan. 2019. [Online]. Available: http://coq.inria.fr

[31] R. Affeldt, "On construction of a library of formally verified low-level arithmetic functions," *Innovations in Systems and Software Engineering*, vol. 9, no. 2, pp. 59–77, 2013.

[32] R. Affeldt, D. Nowak, and K. Yamada, "Certifying assembly with formal security proofs: The case of BBS," *Science of Computer Programming*, vol. 77, no. 10–11, pp. 1058–1074, 2012.

[33] R. Affeldt and N. Marti, "An approach to formal verification of arithmetic functions in assembly," in *Advances in Computer Science*, ser. LNCS, M. Okada and I. Satoh, Eds., vol. 4435. Springer, 2007, pp. 346–360.

[34] M. O. Myreen and M. J. C. Gordon, "Hoare logic for realistically modelled machine code," in *TACAS*, ser. LNCS, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 568–582.

[35] M. O. Myreen and G. Curello, "Proof pearl: A verified bignum implementation in x86-64 machine code," in *Certified Programs and Proofs*, ser. LNCS, vol. 8307. Springer, 2013, pp. 66–81.

[36] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 2, pp. 7:1–7:31, 2015.

[37] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *USENIX Security Symposium 2015*. USENIX Association, 2015, pp. 207–221.

[38] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedTLS HMAC-DRBG," in *CCS*. ACM, 2017, pp. 2007–2020.

[39] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang, "Verifying Curve25519 software," in *CCS*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 299–309.

[40] M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs," in *CCS*, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017.