Verifying Arithmetic Assembly Programs in Cryptographic Primitives

Andy Polyakov
The OpenSSL project
appro@openssl.org

Ming-Hsien Tsai[1]
Academia Sinica, Taiwan
mhtsai208@gmail.com

Bow-Yaw Wang[2]
Academia Sinica, Taiwan
bywang@iis.sinica.edu.tw

Bo-Yin Yang[3]
Academia Sinica, Taiwan
by@crypto.tw

Abstract

Arithmetic over large finite fields is indispensable in modern cryptography. For efficienty, these operations are often implemented in manually optimized assembly programs. Since these arithmetic assembly programs necessarily perform lots of non-linear computation, checking their correctness is a challenging verification problem. We develop techniques to verify such programs automatically in this paper. Using our techniques, we have successfully verified a number of assembly programs in OpenSSL. Moreover, our tool verifies the boringSSL Montgomery Ladderstep (about 1400 assembly instructions) in 1 hour. This is by far the fastest verification technique for such programs.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Formal verification, Cryptography, Assembly Programs

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.4

Category Invited Talk

1 Introduction

Cryptographic primitives are the building blocks of computer security. They are indispensable in various encryption, authentication, and key exchange protocols. Underneath these critical primitives, arithmetic over large finite fields is necessitated by modern cryptography. Efficiency of arithmetic operations such as addition and multiplication is crucial to practical applicability of cryptographic primitives due to their wide usage. Consequently, these operations are implemented by low-level languages (such as C or assembly). Indeed, more than forty arithmetic assembly subroutines for different architectures are found in the OpenSSL NIST

[1] partially supported by the Academia Sinica Project AS-106-TP-A06
[2] partially supported by the Academia Sinica Project AS-106-TP-A06
[3] partially supported by the MOST Project 105-2221-E-001-014-MY3

P-256 elliptic curve cryptographic library. Each is manually optimized to attain the best performance.

Since cryptographic primitives rely on arithmetic operations over large finite fields, correct implementations of such operations are essential to computer security. For implementations written in C, source codes are compiled into executable binary codes. Certified compilation (for instance, the CompCert project) is necessary to ensure correctness after C codes are verified. Even so, hand-optimized assembly implementations are still significantly more efficient than C implementations (up to two times faster for the OpenSSL NIST P-256 Intel Broadwell microarchitecture). Manually optimized assembly implementations for arithmetic operations are typical in practical cryptography. We verify such low-level codes for cryptographic primitives in this paper.

Several obstacles must be overcome. Different architectures have different instruction sets. Even for the same architecture (x86_64), different microarchitectures may have different instruction sets (Broadwell versus its predecessors). Since one would like to develop verification techniques across different instruction sets, a unified framework is preferred. Code sizes also vary from operations significantly. From a dozen of instructions (multiplication by two) to more than a thousand (group operation in elliptic curves), these assembly implementations must be verified with reasonable resources. Last but not least, several assembly programs realize non-linear multiplication over large finite fields. Such algebraic properties are hard to verify by bit blasting. Existing SMT solvers for the bit-vector theory cannot verify the multiplication of two 256-bit numbers. New techniques have to be developed for assembly codes in this special domain.

We propose a domain specific language CryptoLine for modeling assembly programs across different architectures. The language contains instructions used in implementations of arithmetic operations. Different from assembly languages, operands and flags are explicit in CryptoLine for clarity. CryptoLine moreover allows users to specify program properties with assertions, pre- and post-conditions. Using CryptoLine, assembly programs of different arithmetic operations from the OpenSSL cryptographic library are modeled and specified. We feel CryptoLine is a suitable abstraction for assembly programs in cryptographic primitives.

Rewriting assembly codes in CryptoLine can be a daunting task. Moreover, assembly programs can be written in various ways. For example, assembly code fragments can be put in the `asm` statement in GNU C compilers; the OpenSSL library allows programmers to write portable assembly codes with its Perl scripts. Instead of writing parsers for various assembly development environments, we extract assembly codes from execution traces. Since CryptoLine is designed to model assembly programs, each assembly instruction can be translated to CryptoLine statements straightforwardly. Using a simple Python script, extracted assembly codes are converted to CryptoLine programs automatically. Users can transform their assembly programs to CryptoLine programs for verification via simple scripts. Usability of our work is greatly improved.

For verification, we consider conjunctions of range and algebraic predicates. Range predicates specify program variable ranges with the unsigned bit-vector theory; they are resolved by SMT solvers rather straightforwardly. Algebraic predicates specify algebraic properties among program variables; they are reduced to instances of the ideal membership problem through a series of transformations. Instances of the ideal membership problem are sent to and solved by computer algebra systems. Our hybrid technique decomposes verification problems and takes advantages of recent developments in SMT solving and computer algebra. It opens new opportunities in verifying non-linear arithmetic computation in cryptographic programs.

We report case studies in OpenSSL, boringSSL, and mbedTLS. For OpenSSL, we verify eight assembly subroutines converted from execution traces in the OpenSSL NIST P-256 cryptographic library. These subroutines perform arithmetic computation including but not limited to square and multiplication over the Galois field $GF(2^{256}-2^{224}+2^{192}+2^{96}-1)$. The Montegomery multiplication subroutine over arbitrary 256-bit Montgomery primes is also verified. For boringSSL, the multiplication and square subroutines over $GF(2^{255}-19)$ are verified, as well as the Montgomery Ladderstep subroutine for X25519 ($\approx$ 1400 instructions). For mbedTLS, the multiplication subroutine accepts any size of inputs and involves both assembly and C code. The execution traces are extracted and verified.

Related Work. A domain-specific language BVCRYPTOLINE has been proposed to model low-level mathematical constructs in cryptographic programs [16]. Programs in BVCRYPTOLINE can be verified automatically by a certified approach. Both the certified approach and ours reduce the verification problem to SMT problems and ideal membership problems. However, we introduce new instructions in our domain-specific language so that more low-level arithmetic programs can be verified. Unlike the certified approach which verifies programs in BVCRYPTOLINE, we target on real industrial programs and provide scripts for the extraction of assembly codes from execution traces and for the translation from assembly codes to our domain-specific language. Although our approach is not certified, it verifies programs much faster than the certified approach does. The tool `gfverif` [6] has been used to automatically verify a C implementation of the Montgomery Ladderstep. In `gfverif`, range properties and algebraic properties are verified separatedly by a specialized range analysis and by the Sage computer-algebra system. A drawback of `gfverif` is that programs not written in the constructs provided by `gfverif` cannot be verified.

A hand-optimized assembly implementation of the Montgomery Ladderstep has been verified by a semi-automatic approach [8] with SMT solvers. As non-linear arithmetic operations are hard for SMT solvers to verify, the semi-automatic approach requires manual program annotation to reduce verification problems to smaller ones and COQ proofs for some theorems about modulo operation. Similarly, several mathematical constructs have been re-implemented in F* [18] and Vale [7] and to be verified using a combination of SMT solving and manual proofs.

Fiat-Crypto can synthesize correct-by-construction assembly codes for mathematical constructs but the synthesized codes are not as efficient as hand-optimized assembly implementations [9]. Various implementations of mathematical constructs, hash functions, and random number generators have been formalized and manually verified in proof assistants [1, 3, 2, 14, 13, 4, 5, 17]. Cryptol/SAW can automatically verify several cryptographic implementations in C and Java against their reference implementations but the correctness of the reference implementations is not proven [15].

After preliminaries (Section 2), the domain specific language CRYPTOLINE for cryptographic assembly programs is presented in Section 3. Our verification algorithm is given in Section 4. Case studies are reported in Section 5. Section 6 concludes the presentation.

2 Preliminaries

Let $\mathbb{N}$ be the set of non-negative integers and $\mathbb{Z}$ the set of integers. $[n] = \{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$. Fix a set $\vec{v} = \{x, y, z, \ldots\}$ of variables and a set $\vec{c} = \{a, b, c, \ldots\}$ of carry flags such that $\vec{v} \cap \vec{c} = \emptyset$. Let $\vec{x} = \vec{v} \cup \vec{c}$. $\mathbb{Z}[\vec{x}]$ denotes the set of polynomials over $\vec{x}$ with coefficients in $\mathbb{Z}$. A set $I \subseteq \mathbb{Z}[\vec{x}]$ is an *ideal* if $f + g \in I$ for every $f, g \in I$; and $h \times f \in I$ for every $h \in \mathbb{Z}[\vec{x}]$ and $f \in I$. Given $G \subseteq \mathbb{Z}[\vec{x}]$, $\langle G \rangle$ is the minimal ideal containing G; G are the *generators* of $\langle G \rangle$. The *ideal membership* problem is to decide if $f \in I$ for a given ideal I and $f \in \mathbb{Z}[\vec{x}]$.

$$
\begin{array}{rcl}
\mathit{Num} & ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Var} & ::= & \vec{v} \\
\mathit{Flag} & ::= & \vec{c} \\
\mathit{Expr} & ::= & \mathit{Num} \mid \mathit{Var} \mid \mathit{Flag} \mid \\
 & & \mathit{Expr}+\mathit{Expr} \mid \mathit{Expr}-\mathit{Expr} \mid \\
 & & \mathit{Expr}\times\mathit{Expr}
\end{array}
$$

(a) Expressions.

$$
\begin{array}{l}
\mathit{APred} ::= \top \mid \mathit{APred} \wedge \mathit{APred} \mid \\
\qquad \mathit{Expr} = \mathit{Expr} \mid \mathit{Expr} \equiv \mathit{Expr} \; \mathsf{mod} \; \mathit{Expr} \mid \\
\mathit{RPred} ::= \top \mid \mathit{RPred} \wedge \mathit{RPred} \mid \\
\qquad \mathit{Expr} <_w \mathit{Expr} \mid \mathit{Expr} \leq_w \mathit{Expr}
\end{array}
$$

(b) Predicates.

Figure 1 Syntax of Expressions and Predicates.

3 Domain Specific Language – CryptoLine

CryptoLine is designed to model and specify arithmetic assembly programs in cryptograhpic primitives. Arithmetic over large finite fields is essential to modern cryptography. In practice, it is necessary to perform arithmetic computation with numbers in hundreds of bits lest security may be compromised due to cryptoanalysis. We analyze real arithmetic assembly programs, identify a small subset of assembly instructions, and formalize the subset in our domain specific language CryptoLine. In order to specify properties about programs, the language is enriched with statements like Assert and Assume. We detail the design of CryptoLine in this section.

3.1 Syntax

As in low-level cryptographic programs, numbers are non-negative in CryptoLine. The language also allows variables and binary flags. Expressions are only used in property specifications. They admit arithmetic operators $+$, $-$, and $\times$ (Figure 1a). Property specifications are divided into two classes: algebraic and range predicates. An algebraic predicate is a conjunction of equalities or modulo equalities. A range predicate is a conjunction of finite-width comparisons[4]. For instance, $\mathit{Expr} <_w \mathit{Expr}$ means the w-bit less-than relation between two expressions in Figure 1b.

CryptoLine statements contain assembly instructions used in arithmetic computation[5]. They even have a similar syntax: mnemonic, destination variables, and source arguments in order (Figure 2a). Set is the assignment statement. The Cmov statement is the conditional assignment. Mul is the half multiplication whereas Mulf is the full multiplication. Add! is the addition statement without setting the carry flag. Add is the addition statement with setting the carry flag. Adc is the addition with carry statement. Similarly, Sub! is the subtraction statement without setting the borrow flag, Sub is the subtraction with setting the borrow flag, and Sbb is the subtraction with borrow statement. A predicate consists of an algebraic and a range predicate separated by $\|$ (Figure 2b). The Assert statement asserts a predicate. The Assume statement assumes a predicate. A program is a sequence of statements. Finally, a specification contains a program with two predicates as the pre- and post-conditions.

[4] Range predicates such as negation, equality, and disjunction are also allowed in our implementation. Expressions allowed in a range predicate also include signed/unsigned remainder, bit-wise and, bit-wise or, and bit-wise xor.

[5] Instructions such as setting a binary flag, clearing a binary flag, and instructions in [16] are also allowed in our implementation.

$$\begin{array}{llll} \mathit{Stmt} ::= & \mathsf{Set}\ \mathit{Var}\ \mathit{Arg} & \mid & \mathsf{Cmov}\ \mathit{Var}\ \mathit{Flag}\ \mathit{Arg}\ \mathit{Arg} \quad \mid \\ & \mathsf{Add!}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} & \mid & \mathsf{Add}\ \mathit{Flag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} \quad \mid \\ & \mathsf{Sub!}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} & \mid & \mathsf{Sub}\ \mathit{Flag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} \quad \mid \\ & \mathsf{Adc}\ \mathit{Flag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg}\ \mathit{Flag} & \mid & \mathsf{Mul}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} \quad \mid \\ & \mathsf{Sbb}\ \mathit{Flag}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg}\ \mathit{Flag} & \mid & \mathsf{Mulf}\ \mathit{Var}\ \mathit{Var}\ \mathit{Arg}\ \mathit{Arg} \quad \mid \\ & \mathsf{Assert}\ \mathit{Pred} & \mid & \mathsf{Assume}\ \mathit{Pred} \end{array}$$

(a) Statements.

$$\begin{array}{rcl} \mathit{Arg} & ::= & \mathit{Num} \mid \mathit{Var} \\ \mathit{Prog} & ::= & \mathit{Stmt};\ \mid \mathit{Stmt}; \mathit{Prog} \\ \mathit{Pred} & ::= & \mathit{RPred} \| \mathit{APred} \\ \mathit{Spec} & ::= & (\!|\mathit{Pred}|\!)\ \mathit{Prog}\ (\!|\mathit{Pred}|\!) \end{array}$$

(b) Programs and Specifications.

Figure 2 Syntax of Programs and Specifications.

1:	Set r_0 x_0;	6:	Add! r_0 r_0 4503599627370458;	11:	Sub! r_0 r_0 y_0;
2:	Set r_1 x_1;	7:	Add! r_1 r_1 4503599627370494;	12:	Sub! r_1 r_1 y_1;
3:	Set r_2 x_2;	8:	Add! r_2 r_2 4503599627370494;	13:	Sub! r_2 r_2 y_2;
4:	Set r_3 x_3;	9:	Add! r_3 r_3 4503599627370494;	14:	Sub! r_3 r_3 y_3;
5:	Set r_5 x_4;	10:	Add! r_4 r_4 4503599627370494;	15:	Sub! r_4 r_4 y_4;

Figure 3 Subtraction *sub*.

Example. Consider the CryptoLine program for the subtraction over $GF(2^{255}-19)$ in X25519. In Figure 3, each element in the finite field is represented by five 51-bit numbers. Each 51-bit number is a *limb* of the representation. The program first assigns the minuend (x_i's) to the result (r_i's). It then adds $4503599627370458 = 2^{52} - 38$ to the lowest limb and $4503599627370494 = 2^{52} - 2$ to other limbs of the result. Finally, the program subtracts the subtrahend (y_i's) from the result.

In order to specify the subtraction program, let $\mathit{radix51}(\ell_4, \ell_3, \ell_2, \ell_1, \ell_0)$ denote

$$\ell_4 \times 2^{51\times 4} + \ell_3 \times 2^{51\times 3} + \ell_2 \times 2^{51\times 2} + \ell_1 \times 2^{51\times 1} + \ell_0.$$

That is, $\mathit{radix51}(\ell_4, \ell_3, \ell_2, \ell_1, \ell_0)$ is the element represented by ℓ_i's. We have the following specification for the program in Figure 3:

$$\begin{array}{c} (\!|\ \bigwedge_{i=0}^{4} x_i \leq_{64} 2^{51} + 2^{15} \wedge \bigwedge_{i=0}^{4} y_i \leq_{64} 2^{51} + 2^{15}\ \| \top |\!) \\ \mathit{sub} \\ (\!|\ \bigwedge_{i=0}^{4} r_i <_{64} 2^{54}\ \|\ \begin{array}{c} \mathit{radix51}(x_4, x_3, x_2, x_1, x_0) - \mathit{radix51}(y_4, y_3, y_2, y_1, y_0) \\ \equiv \mathit{radix51}(r_4, r_3, r_2, r_1, r_0) \bmod 2^{255} - 19 \end{array} |\!). \end{array}$$

Given each limb of the minuend and subtrahend slightly greater than 2^{51}, the specification says the subtraction program computes the difference over $GF(2^{255}-19)$ with the result less than 2^{54}. To see why the subtraction program satisfies the algebraic specification, observe that $\mathit{radix51}(2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-19) = 2^{255}-19$. Hence $\mathit{radix51}(2^{52}-2, 2^{52}-2, 2^{52}-2, 2^{52}-2, 2^{52}-38) = 2\times(2^{255}-19)$. That is, the subtraction program adds $2\times(2^{255}-19)$ to the minuend and then subtracts the subtrahend. Since the algebraic specification only requires modulo equality, the program is indeed correct. Adding $2 \times (2^{255} - 19)$ does not induce the propagation of carry flags during addition. It moreover prevents borrow flag propagation during subtraction.

3.2 Semantics

Let W be a architecture-dependent parameter: $W = 2^{64}$ for 64-bit architectures; $W = 2^{32}$ for 32-bit architectures. A *state* is a mapping from *Var* to $[W-1]$ and from *Flag* to $\{0,1\}$, and $\perp$ is the designated *error* state. The error state does not satisfy any predicate. Figure 4 gives the operational semantics of each statement.

The semantics for CRYPTOLINE is standard for unsigned bounded arithmetic. The Set statement updates the value of a variable in a state. Cmov updates a variable by the given flag. Add! updates a variable by the sum of arguments if the sum is smaller than W, and otherwise results in the error state (overflows in this case). Add and Adc update a flag and a variable by the sum of arguments. Sub! updates a variable by the difference of arguments if the difference is non-negative, and otherwise results in the error state (underflows in this case). Sub and Sbb update a flag and a variable by the difference of arguments. For the half multiplication Mul, it is an error if the higher bits of the product is non-zero (overflows in this case). The full multiplication Mulf always terminates successfully if the two updated variables are different. Assert results in the error state if the current state does not satisfy the given predicate. Assume ensures the new state satisfying the given predicate. Finally, the error state always propagates.

Let σ and τ be states. The semantics of a program is inductively defined as follows. $\sigma \xrightarrow{stmt} \tau$ is defined in Figure 4. $\sigma \xrightarrow{stmt;prog} \tau$ if there is a state λ such that $\sigma \xrightarrow{stmt} \lambda$ and $\lambda \xrightarrow{prog} \tau$. We call a program *prog safe* with respect to a predicate P if for every σ and τ, $\sigma \models P$ and $\sigma \xrightarrow{prog} \tau$ imply $\tau \neq \perp$. We write $\models (\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$ if for every σ, τ with $\sigma \models P \wedge Q$ and $\sigma \xrightarrow{prog} \tau$, we have $\tau \models P' \wedge Q'$. Note that $\not\models (\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$ if there is an assertion failure during execution.

Example (continued). The subtraction program will never reach the error state. For example, consider the variable r_0. The statement at line 1 assigns r_0 the value of x_0, which is smaller than $2^{51} + 2^{15}$ by the precondition. The Add! statement at line 6 assigns r_0 a value between $4503599627370458 = 2^{52} - 38$ and $2^{51} + 2^{15} + 4503599627370458 = 2^{51} + 2^{15} + 2^{52} - 38 < 2^{64}$. Thus the Add! statement never goes to the error state. Finally, observe $y_0 \leq 2^{51} + 2^{15} \leq 2^{52} - 38 \leq r_0$ after line 6. The Sub! statement at line 11 therefore never goes to the error state.

4 Verifying CryptoLine Programs

We want to check if $\models (\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$ for a given specification $(\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$ with range predicates P, P' and algebraic predicates Q, Q'. Firstly, we transform the specification to static single assignments where variables are indexed such that no input variables are assigned and every variable is assigned at most once. As CRYPTOLINE programs are straight-line, the transformation can be done easily so that the validity of specification is preserved. In this section, we will assume the specification $(\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$ is in static single assignments and write $x^{(i)}$ to explicitly indicate a variable x with index i when needed.

Secondly, we need to ensure that all assertions in *prog* are valid. Consider the first assertion Assert$P'' \| Q''$ in *prog* and let $prog = prog_1;$ Assert$P'' \| Q''; prog_2$. We verify the validity of this assertion by checking if $\models (\!|P \| Q|\!) prog_1 (\!|P'' \| Q''|\!)$ is valid. Once the assertion is valid, it is safe to be removed when verifying *prog*. Therefore all the assertion checking can be reduced to specification checking of programs without assertions. In this section, we will assume there is no assertion in *prog*.

Finally, observe $\models (\!|P \| \top|\!) prog (\!|P' \| \top|\!)$ and $\models (\!|P \| Q|\!) prog (\!|\top \| Q'|\!)$ imply $\models (\!|P \| Q|\!) prog (\!|P' \| Q'|\!)$. The specification $(\!|P \| \top|\!) prog (\!|P' \| \top|\!)$ involves only range predicates; and the specification $(\!|P \| Q|\!) prog (\!|\top \| Q'|\!)$ concerns only algebraic properties. We therefore divide the verfication task into two parts: range and algebraic properties.

$$\llbracket n \rrbracket_\sigma = n \qquad \llbracket x \rrbracket_\sigma = \sigma(x), \text{ for } x \in \vec{v} \qquad \llbracket c \rrbracket_\sigma = \sigma(c), \text{ for } c \in \vec{c}$$

$$\begin{array}{llll}
\sigma & \xrightarrow{\mathsf{Set}\ x\ u} & \sigma[x \mapsto \llbracket u \rrbracket_\sigma] & \\
\sigma & \xrightarrow{\mathsf{Cmov}\ x\ b\ u\ v} & \sigma[x \mapsto R] & \text{where } R = \llbracket u \rrbracket_\sigma \text{ if } \llbracket b \rrbracket_\sigma = 1; \llbracket v \rrbracket_\sigma \text{ if } \llbracket b \rrbracket_\sigma = 0 \\
\sigma & \xrightarrow{\mathsf{Add!}\ x\ u\ v} & \sigma' & \text{where } R = \llbracket u \rrbracket_\sigma + \llbracket v \rrbracket_\sigma \text{ and } \sigma' = \sigma[x \mapsto R] \\
 & & & \text{if } R/W = 0; \bot \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Add}\ b\ x\ u\ v} & \sigma[b, x \mapsto R/W, R \bmod W] & \text{where } R = \llbracket u \rrbracket_\sigma + \llbracket v \rrbracket_\sigma \\
\sigma & \xrightarrow{\mathsf{Adc}\ b\ x\ u\ v\ c} & \sigma[b, x \mapsto R/W, R \bmod W] & \text{where } R = \llbracket u \rrbracket_\sigma + \llbracket v \rrbracket_\sigma + \llbracket c \rrbracket_\sigma \\
\sigma & \xrightarrow{\mathsf{Sub!}\ x\ u\ v} & \sigma' & \text{where } R = \llbracket u \rrbracket_\sigma - \llbracket v \rrbracket_\sigma \text{ and } \sigma' = \sigma[x \mapsto R] \\
 & & & \text{if } \llbracket u \rrbracket_\sigma \geq \llbracket v \rrbracket_\sigma; \bot \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Sub}\ b\ x\ u\ v} & \sigma[b, x \mapsto B, R \bmod W] & \text{where } R = \llbracket u \rrbracket_\sigma - \llbracket v \rrbracket_\sigma + W \text{ and } B = 0 \text{ if} \\
 & & & \llbracket u \rrbracket_\sigma \geq \llbracket v \rrbracket_\sigma; 1 \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Sbb}\ b\ x\ u\ v\ c} & \sigma[b, x \mapsto B, R \bmod W] & \text{where } R = \llbracket u \rrbracket_\sigma - \llbracket v \rrbracket_\sigma - \llbracket c \rrbracket_\sigma + W \text{ and } B \\
 & & & = 0 \text{ if } \llbracket u \rrbracket_\sigma \geq \llbracket v \rrbracket_\sigma + \llbracket c \rrbracket_\sigma; 1 \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Mul}\ x\ u\ v} & \sigma' & \text{where } R = \llbracket u \rrbracket_\sigma \times \llbracket v \rrbracket_\sigma \text{ and } \sigma' = \sigma[x \mapsto R] \\
 & & & \text{if } R/W = 0; \bot \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Mulf}\ x\ y\ u\ v} & \sigma[x, y \mapsto R/W, R \bmod W] & \text{if } x \neq y \text{ and } R = \llbracket u \rrbracket_\sigma \times \llbracket v \rrbracket_\sigma \\
\sigma & \xrightarrow{\mathsf{Assert} P \| Q} & \sigma' & \text{where } \sigma' = \sigma \text{ if } \sigma \models P \wedge Q; \bot \text{ otherwise} \\
\sigma & \xrightarrow{\mathsf{Assume} P \| Q} & \sigma & \text{if } \sigma \models P \wedge Q \\
\bot & \xrightarrow{stmt} & \bot & \text{where } stmt \in Stmt
\end{array}$$

Figure 4 Semantics.

4.1 Range Properties

Range properties are amenable to analysis by bit blasting. We therefore reduce the problem of deciding $\models (\!| P \| \top |\!) prog (\!| P' \| \top |\!)$ to SMT solving. More specifically, we construct a formula Ψ in the bit vector theory such that Ψ is satisfiable if and only if $\not\models (\!| P \| \top |\!) prog (\!| P' \| \top |\!)$. An SMT solver is then employed to check range properties. Since the reduction is standard, details are omitted here (see, for instance, [12]). Note that $\not\models (\!| P \| \top |\!) prog (\!| P' \| \top |\!)$ may be caused by the violation of the post-condition or by the violation of *program safety* (that is, overflow/underflow of some program statement). Therefore, safety check of *prog* is also encoded in Ψ by the same way as in [16]) and $\models (\!| P \| \top |\!) prog (\!| P' \| \top |\!)$ actually indicates that *prog* is safe with respect to P.

4.2 Algebraic Properties

Algebraic properties, especially those involving non-linear multiplication, are not suitable for SMT solving. In order to effectively verify algebraic properties, we propose modular polynomial abstraction. In modular polynomial abstraction, program behaviors are modeled by solutions to systems of modular polynomial equations. Checking algebraic properties is reduced to modular polynomial equation entailments. The modular polynomial equation entailment problem in turn is reduced to the ideal membership problem. The ideal membership problem is widely studied in commutative algebra. Decision procedures for the ideal membership problem are available in computer algebra systems. We hence employ computer algebra systems to verify algebraic properties.

Set x u	$\hookrightarrow$	$x - u = 0$
Cmov x b u v	$\hookrightarrow$	$x - (b \times u + (1 - b) \times v) = 0$
Add! x u v	$\hookrightarrow$	$x - (u + v) = 0$
Add b x u v	$\hookrightarrow$	$(x + b \times W) - (u + v) = 0 \wedge b \times (1 - b) = 0$
Adc b x u v c	$\hookrightarrow$	$(x + b \times W) - (u + v + c) = 0 \wedge b \times (1 - b) = 0$
Sub! x u v	$\hookrightarrow$	$x - (u - v) = 0$
Sub b x u v	$\hookrightarrow$	$(x - b \times W) - (u - v) = 0 \wedge b \times (1 - b) = 0$
Sbb b x u v c	$\hookrightarrow$	$(x - b \times W) - (u - v - c) = 0 \wedge b \times (1 - b) = 0$
Mul x u v	$\hookrightarrow$	$x - (u \times v) = 0$
Mulf x y u v	$\hookrightarrow$	$(x \times W + y) - (u \times v) = 0$
Assert $P \| Q$	$\hookrightarrow$	$\top$
Assume $P \| Q$	$\hookrightarrow$	$\mathit{poly}(Q)$

Figure 5 Modular Polynomial Equations.

In the following, the three transformations from the verification problem of algebraic properties on CryptoLine programs to the ideal membership problem are explained. We first show how to transform program behaviors to systems of modular polynomial equations. The algebraic property verification problem is then reduced to the modular polynomial equation entailment problem. Finally, we explicate how to solve the entailment problem by the ideal membership problem in commutative algebra.

4.2.1 Modular Polynomial Equations

Let $f(\vec{x}), g(\vec{x}) \in \mathbb{Z}[\vec{x}]$. A *modular polynomial equation* is of the form $f(\vec{x}) = 0$ or $f(\vec{x}) \equiv 0 \bmod g(\vec{x})$. A *system* of modular polynomial equations is denoted by $\bigwedge_{i=1}^{k} f_i(\vec{x}) = 0 \wedge \bigwedge_{i=1}^{l} g_i(\vec{x}) \equiv 0 \bmod h_i(\vec{x})$ where $f_i(\vec{x}), g_i(\vec{x}), h_i(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for all i. A state σ is a *solution* to a modular polynomial equation (written $\sigma \models f(\vec{x}) = 0$ or $\sigma \models f(\vec{x}) \equiv 0 \bmod g(\vec{x})$) if the equation holds under the valuation σ. A state σ is a solution to a system of modular polynomial equations if it is a solution to every equations in the system.

Our first task is to describe program behaviors. Consider the transformation from CryptoLine statements to systems of modular polynomial equations (Figure 5). Set is transformed to the equation stating that the updated variable is equal to the argument. For Cmov, the argument b can be either 0 or 1. Hence its corresponding equation identifying the updated variable to u or v by the value of b. Add! and Sub! are transformed to equations respectively mimicing the addition and the subtraction. In addition to the equations for the updated variable and flag, Add, Adc, Sub, Sbb also have equations restricting the values of flags to be 0 or 1. Mul and Mulf are transformed to equations mimicing the multiplication. Assert is verified as another specification and is ignored. Finally, Assume$P \| Q$ is transformed to $\mathit{poly}(Q)$. Given $Q \in \mathbb{Z}[\vec{x}]$, $\mathit{poly}(Q)$ is Q with $e_1 = e_2$ replaced by $e_1 - e_2 = 0$ and $e_1 \equiv e_2 \bmod e_3$ replaced by $e_1 - e_2 \equiv 0 \bmod e_3$.

We have the following theorem for the transformation from CryptoLine to a system of modular polynomial equations.

▶ **Theorem 1.** *For each stmt $\hookrightarrow \Phi$ in Figure 5 and non-error states σ and τ, we have $\sigma \xrightarrow{stmt} \tau$ implies $\tau \models \Phi$.*

1:	Set $r_0^{(0)}$ $x_0^{(0)}$;	6:	Add! $r_0^{(1)}$ $r_0^{(0)}$ 4503599627370458;	11:	Sub! $r_0^{(2)}$ $r_0^{(1)}$ $y_0^{(0)}$;
2:	Set $r_1^{(0)}$ $x_1^{(0)}$;	7:	Add! $r_1^{(1)}$ $r_1^{(0)}$ 4503599627370494;	12:	Sub! $r_1^{(2)}$ $r_1^{(1)}$ $y_1^{(0)}$;
3:	Set $r_2^{(0)}$ $x_2^{(0)}$;	8:	Add! $r_2^{(1)}$ $r_2^{(0)}$ 4503599627370494;	13:	Sub! $r_2^{(2)}$ $r_2^{(1)}$ $y_2^{(0)}$;
4:	Set $r_3^{(0)}$ $x_3^{(0)}$;	9:	Add! $r_3^{(1)}$ $r_3^{(0)}$ 4503599627370494;	14:	Sub! $r_3^{(2)}$ $r_3^{(1)}$ $y_3^{(0)}$;
5:	Set $r_5^{(0)}$ $x_4^{(0)}$;	10:	Add! $r_4^{(1)}$ $r_4^{(0)}$ 4503599627370494;	15:	Sub! $r_4^{(2)}$ $r_4^{(1)}$ $y_4^{(0)}$;

Figure 6 *sub* in static single assignments.

Sketch. We only show the proof for two statements here. Suppose $\sigma \xrightarrow{\mathsf{Add}\ b\ x\ u\ v} \tau$. If $[\![u]\!]_\sigma + [\![v]\!]_\sigma < W$, $[\![b]\!]_\tau = 0$ and $[\![x]\!]_\tau = [\![u]\!]_\sigma + [\![v]\!]_\sigma$. Otherwise, we have $[\![b]\!]_\tau = 1$ and $[\![x]\!]_\tau = [\![u]\!]_\sigma + [\![v]\!]_\sigma - W$. As statements are in static single assignments, $[\![u]\!]_\tau = [\![u]\!]_\sigma$ and $[\![v]\!]_\tau = [\![v]\!]_\sigma$. Hence $\tau \models (x + b \times W) - (u + v) = 0$ and $\tau \models b \times (1 - b) = 0$ in both cases.

Now suppose $\sigma \xrightarrow{\mathsf{Cmov}\ x\ b\ u\ v} \tau$. If $[\![b]\!]_\sigma = 1$, $[\![x]\!]_\tau = [\![u]\!]_\sigma = [\![b]\!]_\sigma \times [\![u]\!]_\sigma$. Otherwise $[\![b]\!]_\sigma = 0$ and $[\![x]\!]_\tau = [\![v]\!]_\sigma = (1 - [\![b]\!]_\sigma) \times [\![v]\!]_\sigma$. As statements are in static single assignments, $[\![b]\!]_\tau = [\![b]\!]_\sigma$, $[\![u]\!]_\tau = [\![u]\!]_\sigma$ and $[\![v]\!]_\tau = [\![v]\!]_\sigma$. In both cases, we have $\tau \models x - (b \times u + (1 - b) \times v) = 0$. ◀

By the assumption that programs are in static single assignments and Theorem 1, a system of modular polynomial equations whose solutions are program execution traces is constructed. More formally, we have the following corollary:

▶ **Corollary 2.** *Suppose $stmt_1; stmt_2; \cdots ; stmt_n$ is in static single assignments. Let σ and τ be non-error states with $\sigma \xrightarrow{stmt_1; stmt_2; \cdots ; stmt_n} \tau$. Suppose $stmt_i \hookrightarrow \Phi_i$ (Figure 5) for every $1 \leq i \leq n$. Then $\tau \models \bigwedge_{i=1}^{n} \Phi_i$.*

Example (continued). Let us compute the system of modular polynomial equations for the subtraction program in Figure 3. We rewrite the program in static single assignments (Figure 6). The specification concerning algebraic properties, denoted by $aspec_{sub}$, is rewritten as well:

$$
\begin{array}{c}
(\!|\ \bigwedge_{i=0}^{4} x_i^{(0)} \leq_{64} 2^{51} + 2^{15} \wedge \bigwedge_{i=0}^{4} y_i^{(0)} \leq_{64} 2^{51} + 2^{15}\ \|\top|\!) \\
sub \\
(\!|\top\| \begin{array}{c} radix51(x_4^{(0)}, x_3^{(0)}, x_2^{(0)}, x_1^{(0)}, x_0^{(0)}) - radix51(y_4^{(0)}, y_3^{(0)}, y_2^{(0)}, y_1^{(0)}, y_0^{(0)}) \\ \equiv radix51(r_4^{(2)}, r_3^{(2)}, r_2^{(2)}, r_1^{(2)}, r_0^{(2)}) \bmod 2^{255} - 19 \end{array} |\!)
\end{array}
$$

For each statement in the static single assignments, we apply the transformation in Figure 5. Figure 7 shows the corresponding modular polynomial equations.

4.2.2 Modular Polynomial Equation Entailment

Let Φ and Φ' be systems of modular polynomial equations with variables over $\vec{x}$. A *modular polynomial entailment* is a formula of the form $\forall \vec{x}(\Phi \implies \Phi')$. Given a modular polynomial entailment, the *modular polynomial entailment problem* is to decide whether the entailment holds in the theory of integers. Given systems of modular polynomial equations describing program behaviors and intended algebraic properties, it is standard to transform the verification problem to an instance of the modular polynomial entailment problem.

$$\begin{array}{rlrl}
1: & r_0^{(0)} - x_0^{(0)} = 0 \wedge & 9: & r_3^{(1)} - (r_3^{(0)} + 4503599627370494) = 0 \wedge \\
2: & r_1^{(0)} - x_1^{(0)} = 0 \wedge & 10: & r_4^{(1)} - (r_4^{(0)} + 4503599627370494) = 0 \wedge \\
3: & r_2^{(0)} - x_2^{(0)} = 0 \wedge & 11: & r_0^{(2)} - (r_0^{(1)} - y_0^{(0)}) = 0 \wedge \\
4: & r_3^{(0)} - x_3^{(0)} = 0 \wedge & 12: & r_1^{(2)} - (r_1^{(1)} - y_1^{(0)}) = 0 \wedge \\
5: & r_4^{(0)} - x_4^{(0)} = 0 \wedge & 13: & r_2^{(2)} - (r_2^{(1)} - y_2^{(0)}) = 0 \wedge \\
6: & r_0^{(1)} - (r_0^{(0)} + 4503599627370458) = 0 \wedge & 14: & r_3^{(2)} - (r_3^{(1)} - y_3^{(0)}) = 0 \wedge \\
7: & r_1^{(1)} - (r_1^{(0)} + 4503599627370494) = 0 \wedge & 15: & r_4^{(2)} - (r_4^{(1)} - y_4^{(0)}) = 0 \\
8: & r_2^{(1)} - (r_2^{(0)} + 4503599627370494) = 0 \wedge & &
\end{array}$$

Figure 7 System of Modular Polynomial Equations Φ_{sub} for *sub*.

▶ **Theorem 3.** *Given a specification* $(\!|P \| Q|\!) prog (\!|\top \| Q'|\!)$ *in static single assignments and the corresponding system of modular polynomial equations* Φ_{prog} *for prog where there is no* Assert *statement, if prog is safe with respect to* P *and* $\forall \vec{x}(poly(Q) \wedge \Phi_{prog} \implies poly(Q'))$ *holds, then* $\models (\!|P \| Q|\!) prog (\!|\top \| Q'|\!)$*.*

Example (continued). We apply Theorem 3 to verify algebraic properties on the subtraction program in Figure 3. Recall the system of modular polynomial equations Φ_{sub} for the program in Figure 7. If we can show

$$\forall \vec{x} \left[\top \wedge \Phi_{sub} \Longrightarrow \begin{array}{c} radix51(x_4^{(0)}, x_3^{(0)}, x_2^{(0)}, x_1^{(0)}, x_0^{(0)}) - radix51(y_4^{(0)}, y_3^{(0)}, y_2^{(0)}, y_1^{(0)}, y_0^{(0)}) \\ -radix51(r_4^{(2)}, r_3^{(2)}, r_2^{(2)}, r_1^{(2)}, r_0^{(2)}) \equiv 0 \bmod 2^{255} - 19 \end{array} \right],$$

denoted by ent_{sub}, then $\models aspec_{sub}$.

Our next task is to solve the modular polynomial entailment problem. It is known how to replace modular polynomial equations with polynomial equations and hence simplify the modular polynomial entailment problem [11]. In the following, we review the simplication for the sake of completeness.

Let $e_i(\vec{x}), f_j(\vec{x}), n_j(\vec{x}) g_k(\vec{x}), h_l(\vec{x}), m_l(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for $i \in [I]$, $j \in [J]$, $k \in [K]$, and $l \in [L]$. Consider the instance of the modular polynomial entailment problem:

$$\forall \vec{x} [\bigwedge_{i \in [I]} e_i(\vec{x}) = 0 \wedge \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv 0 \bmod n_j(\vec{x}) \implies \bigwedge_{k \in [K]} g_k(\vec{x}) = 0 \wedge \bigwedge_{l \in [L]} h_l(\vec{x}) \equiv 0 \bmod m_l(\vec{x})]$$

Its consequent has $K + L + 2$ modular polynomial equations. We decompose the problem into $K + L + 2$ instances of the modular polynomial entailment problem. Each instance is of the form

$$\forall \vec{x} [\bigwedge_{i \in [I]} e_i(\vec{x}) = 0 \wedge \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv 0 \bmod n_j(\vec{x}) \implies g(\vec{x}) = 0]; \text{ or}$$

$$\forall \vec{x} [\bigwedge_{i \in [I]} e_i(\vec{x}) = 0 \wedge \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv 0 \bmod n_j(\vec{x}) \implies h(\vec{x}) \equiv 0 \bmod m(\vec{x})].$$

For the first form, we expand the modular polynomial equations and obtain

$$\forall \vec{x} [\bigwedge_{i \in [I]} e_i(\vec{x}) = 0 \wedge \bigwedge_{j \in [J]} [\exists d_j . f_j(\vec{x}) - d_j \cdot n_j(\vec{x}) = 0] \implies g(\vec{x}) = 0],$$

$$\left\langle \begin{array}{c} 2^{255}-19, r_0^{(0)}-x_0^{(0)}, r_1^{(0)}-x_1^{(0)}, r_2^{(0)}-x_2^{(0)}, r_3^{(0)}-x_3^{(0)}, r_4^{(0)}-x_4^{(0)}, \\ r_0^{(1)}-r_0^{(0)}-4503599627370458, r_1^{(1)}-r_1^{(0)}-4503599627370494, \\ r_2^{(1)}-r_2^{(0)}-4503599627370494, r_3^{(1)}-r_3^{(0)}-4503599627370494, \\ r_4^{(1)}-r_4^{(0)}-4503599627370494, \\ r_0^{(2)}-r_0^{(1)}+y_0^{(0)}, r_1^{(2)}-r_1^{(1)}+y_1^{(0)}, r_2^{(2)}-r_2^{(1)}+y_2^{(0)}, r_3^{(2)}-r_3^{(1)}+y_3^{(0)}, \\ r_4^{(2)}-r_4^{(1)}+y_4^{(0)}, \end{array} \right\rangle$$

Figure 8 Ideal for Checking Algebraic Property on Subtraction.

which in turn is equivalent to

$$\forall\vec{x}\forall\vec{d}[\bigwedge_{i\in[I]} e_i(\vec{x})=0 \wedge \bigwedge_{j\in[J]} f_j(\vec{x})-d_j\cdot n_j(\vec{x})=0 \implies g(\vec{x})=0].$$

Similarly, we obtain the following entailment for the second form:

$$\forall\vec{x}\forall\vec{d}[\bigwedge_{i\in[I]} e_i(\vec{x})=0 \wedge \bigwedge_{j\in[J]} f_j(\vec{x})-d_j\cdot n_j(\vec{x})=0 \implies h(\vec{x})\equiv 0 \bmod m(\vec{x})].$$

Note that antecedents in both forms are but polynomial equations. In order to solve the modular polynomial entailment problem, it suffices to solve the following forms:

$$\forall\vec{x}[\bigwedge_{i\in[I]} e_i(\vec{x})=0 \implies g(\vec{x})=0] \tag{1}$$

$$\forall\vec{x}[\bigwedge_{i\in[I]} e_i(\vec{x})=0 \implies h(\vec{x})\equiv 0 \bmod m(\vec{x})] \tag{2}$$

4.2.3 Solving Modular Polynomial Equation Entailment Problem

There is an interesting connection between solving the formula (2) and the ideal membership problem. Suppose $h(\vec{x}) \in \langle e_i(\vec{x}), m(\vec{x})\rangle_{i\in[I]}$. By the definition of ideal, $h(\vec{x})$ is a linear combination of $e_i(\vec{x})$ and $m(\vec{x})$. Hence, there are $c(\vec{x}), c_i(\vec{x}) \in \mathbb{Z}[\vec{x}]$ such that

$$h(\vec{x}) = c(\vec{x})\cdot m(\vec{x}) + \sum_{i\in[I]} c_i(\vec{x})\cdot e_i(\vec{x}).$$

When $e_i(\vec{x})=0$ for every $i\in[I]$, we have $h(\vec{x}) = c(\vec{x})\cdot m(\vec{x})$, that is, $h(\vec{x})\equiv 0 \bmod m(\vec{x})$. The following theorem summarizes the connection:

▶ **Theorem 4** ([11]). *Let $e_i(\vec{x}), g(\vec{x}), h(\vec{x}), m(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for $i\in[I]$.*
1. *The formula (1) holds if $g(\vec{x})$ is in the ideal $\langle e_i(\vec{x})\rangle_{i\in[I]}$;*
2. *The formula (2) holds if $h(\vec{x})$ is in the ideal $\langle e_i(\vec{x}), m(\vec{x})\rangle_{i\in[I]}$.*

Example (continued). Recall that we would like to establish ent_{sub}. By Theorem 4, it suffices to show $radix51(x_4^{(0)}, x_3^{(0)}, x_2^{(0)}, x_1^{(0)}, x_0^{(0)}) - radix51(y_4^{(0)}, y_3^{(0)}, y_2^{(0)}, y_1^{(0)}, y_0^{(0)}) - radix51(r_4^{(2)}, r_3^{(2)}, r_2^{(2)}, r_1^{(2)}, r_0^{(2)})$ is in the ideal from Figure 8.

4.2.4 Completeness

The reduction from the algebraic verification problem to the ideal membership problem is sound but incomplete. There are instances of (1) or (2) whose corresponding ideal membership problems do not hold.

4.2.5 Optimization

The ideal membership problem typically becomes harder when the number of polynomials grows [10]. To reduce the number of polynomials, we apply variable substitution same as in [16]. For the instruction Set x u and its corresponding modular polynomial equation $x - u = 0$, we remove the equation and replace x with u in all other modular polynomial equations. Other instructions that update only one variable are processed similarly. Consider Mulf x y u v and its corresponding modular polynomial equation $(x \times W + y) - (u \times v) = 0$ for an example of instructions that update two variables. Since $(x \times W + y) - (u \times v) = 0$ implies $y = u \times v - x \times W$, we remove the equation and replace y with $u \times v - x \times W$ in all other modular polynomial equations. For an Assume$P \| Q$ statement with an equation $e_1 - e_2 = 0$ in $\mathit{poly}(Q)$, we identify a variable x and an expression e such that $e_1 - e_2 = 0$ if and only if $x = e$. Then $e_1 - e_2 = 0$ is removed after replacing x with e.

Example (continued). By the optimization, it suffices to show

$$\begin{aligned}
&\mathit{radix51}(x_4^{(0)}, x_3^{(0)}, x_2^{(0)}, x_1^{(0)}, x_0^{(0)}) - \mathit{radix51}(y_4^{(0)}, y_3^{(0)}, y_2^{(0)}, y_1^{(0)}, y_0^{(0)}) - \\
&\mathit{radix51}(x_4^{(0)} + 4503599627370494 - y_4^{(0)}, x_3^{(0)} + 4503599627370494 - y_3^{(0)}, \\
&\qquad x_2^{(0)} + 4503599627370494 - y_2^{(0)}, x_1^{(0)} + 4503599627370494 - y_1^{(0)}, \\
&\qquad x_0^{(0)} + 4503599627370458 - y_0^{(0)})
\end{aligned}$$

is in the ideal of $\langle 2^{255} - 19 \rangle$.

5 Evaluation

We have implemented our approach in OCaml. Followed by a case study on Montgomery multiplication in this section, the verification of arithmetic assembly programs in cryptographic libraries is reported.

5.1 Montgomery Multiplication

Consider modulo arithmetic computation over $\mathbb{Z}_m$. Since results must be in $\mathbb{Z}_m$, a modulo operation is necessary. For modulo addition or subtraction, it is relatively easy since results are obtained by subtracting or adding m respectively. For modulo multiplication, the naïve algorithm requires division. This is inefficient. Consider, for instance, modulo arithmetic computation over $\mathbb{Z}_{93}$. Let $a = 79$ and $b = 39$. We have $a + b = 118$ and $118 - 93 = 25$. Hence $(a + b) \bmod 93 = 25$. But $a \times b = 79 \times 39 = 3081$. Since $3081 = 93 \times 33 + 12$. We obtain $(79 \times 39) \bmod 93 = 12$ by division.

To avoid inefficient division, cryptographic programs perform modulo arithmetic computation over $\mathbb{Z}_m$ in Montgomery forms. Two numbers R and m' with $\gcd(R, m) = 1$ and $mm' \equiv -1 \bmod R$ are chosen by programmers. For any $a \in \mathbb{Z}_m$, its *Montgomery representation* is $aR \bmod m$. For modulo addition and subtraction, it is still easy to compute in Montgomery forms since $(a \pm b)R \equiv aR \pm bR \bmod m$. For modulo multiplication, one would like to compute the Montgomery representation $abR \bmod m$ from the representations $aR \bmod m$ and $bR \bmod m$ of a and b respectively. Yet $aR \cdot bR \equiv abR^2 \bmod m$. It appears that one has to multiply the inverse of R and then modulo m to obtain the result.

Surprisingly, Montgomery proposed a reduction algorithm which multiplies the inverse of R and modulo m *simultaneously*. The following computation performs the Montgomery reduction on a number T in Montgomery representation:

$$\begin{array}{rcl} n & = & ((T \bmod R) \times m') \bmod R \\ t & = & (T + n \times m)/R \\ r & = & \text{if } t \geq m \text{ then } t - m \text{ else } t \end{array}$$

We will illustrate by an example. Choose $(m, R, m') = (93, 100, 43)$. We have $\gcd(100, 93) = 1$ and $93 \times 43 \equiv -1 \bmod 100$. The numbers a and b in Montgomery representation are $7900 \bmod 93 = 88$ and $3900 \bmod 93 = 87$ respectively (two hard divisions). To perform the Montegomery reduction on $T = 88 \times 87 = 7656$, we compute $n = ((7656 \bmod 100) \times 43) \bmod 100 = (56 \times 43) \bmod 100 = 2408 \bmod 100 = 8$. $t = (7656 + 8 \times 93)/100 = 8400/100 = 84$. Since $84 < m = 93$, we obtain the product 84 in Montgomery representation. To compute $ab \bmod m$, we perform Montgomery reduction again on $T = 84$. Thus $n = ((84 \bmod 100) \times 43) \bmod 100 = (84 \times 43) \bmod 100 = 3612 \bmod 100 = 12$ and $t = (84 + 12 \times 93)/100 = 1200/100 = 12$. Since $12 < 93$, we have $(79 \times 39) \bmod 93 = 12$ as before. Observe that only two hard divisions are necessary for computing Montgomery representations of 79 and 39. Modulo 100 and dividing multiples of 100 by 100 are trivial. If a number of arithmetic operations are required (as in the case for cryptographic primitives), computation in Montgomery representation is much more efficient than textbook algorithms. Cryptographic libraries subsequently implement arithmetic in Montgomery representation.

Assembly subroutines in OpenSSL go even further than that. Previously, we compute multiplication 88×87 followed by reduction to perform one Montgomery multiplication. In practice, multiplication and reduction are performed simultaneously in Montgomery multiplication. OpenSSL moreover uses the multi-limb Montegomery multiplication algorithm where R can be a large power of 2. Consider the four-limb Montgomery multiplication with 64-bit limbs. m is hence a 256-bit number. Choose $R = 2^{256}$. Define $\mathit{radix64}(\ell_3, \ell_2, \ell_1, \ell_0)$ to be the expression

$$\ell_3 \times 2^{64\times 3} + \ell_2 \times 2^{64\times 2} + \ell_1 \times 2^{64\times 1} + \ell_0.$$

Let $m = \mathit{radix64}(m_3, m_2, m_1, m_0)$, $x = \mathit{radix64}(x_3, x_2, x_1, x_0)$, $y = \mathit{radix64}(\ y_3, y_2, y_1, y_0)$ and $m' \in [2^{64} - 1]$ be inputs and $r = \mathit{radix64}(r_3, r_2, r_1, r_0)$ the output. The four-limb Montgomery multiplication subroutine bn_mul_mont_4 in OpenSSL has the following specification:

$$\begin{array}{c} (\!| \top \| m_0 \equiv 1 \bmod 2 \land m' \times m + 1 \equiv 0 \bmod 2^{64} |\!) \\ r = \text{bn\_mul\_mont\_4}(x, y, m, m') \\ (\!| \top \| x \times y \equiv r \times 2^{256} \bmod m |\!) \end{array}$$

The precondition $m_0 \equiv 1 \bmod 2$ is equivalent to $\gcd(m, 2^{64}) = 1$. On input numbers $x = aR \bmod m$ and $y = bR \bmod m$ in Montgomery representation, the output r satisfies $xy \equiv abR^2 \equiv rR \bmod m$. That is, $r \equiv abR \bmod m$. The output r is the product of a and b in Montgomery representation.

The Montgomery multiplication subroutine for x86_64 is invoked by the C fragment:

```
bn_mul_mont(r, x, y, m, m', n_limbs);
```

where `x`, `y`, `m`, `m'` are arrays of 64-bit unsigned integer and `n_limbs` is the number of limbs. We compile the C code and link it with the OpenSSL cryptographic library. The program execution trace is then extracted by `gdb` along with effective addresses automatically. For 4-limb bn_mul_mont_4 (`n_limbs` = 4), there are about 350 assembly instructions. As an illustration, the following three instructions load the value of `m'`, `y[0]`, and `x[0]` to the registers `r8`, `rbx`, and `rax` respectively.

```
mov         (%r8),%r8                                   #!  EA = L0x6060e0
mov         (%r12),%rbx                                 #!  EA = L0x6060a0
mov         (%rsi),%rax                                 #!  EA = L0x606080
```

For each instruction, we write a Python script to translate it to a CryptoLine statement. Here are the corresponding CryptoLine code for the three instructions:

Set $r8$ $L0x6060e0$;
Set rbx $L0x6060a0$;
Set rax $L0x606080$;

In our automatic translation, each memory cell is a variable identified by its address. Each register is also a variable with the same name. Since effective addresses are obtained from `gdb`, indirect memory operands (such as `-0x10(%rsp,%r15,8)`) are translated to variables corresponding to their effective addresses. It remains to initialize memory cells for inputs.

1:	Set $L0x606080$ x_0;	5:	Set $L0x6060a0$ y_0;	9:	Set $L0x6060c0$ m_0;
2:	Set $L0x606088$ x_1;	6:	Set $L0x6060a8$ y_1;	10:	Set $L0x6060c8$ m_1;
3:	Set $L0x606090$ x_2;	7:	Set $L0x6060b0$ y_2;	11:	Set $L0x6060d0$ m_2;
4:	Set $L0x606098$ x_3;	8:	Set $L0x6060b8$ y_3;	12:	Set $L0x6060d8$ m_3;
				13:	Set $L0x6060e0$ m';

The Assert and Assume statements are indispensable in verifying bn_mul_mont. Consider the following fragment extracted from the assembly subroutine:

1 :	Set rbx y_0;	4 :	Mulf rdx rax rbx rax;	7 :	Mulf $unused$ rbp $r10$ rbp;
2 :	Set rax x_0;	5 :	Set $r10$ rax;	8 :	Mulf rdx rax rbp rax;
3 :	Set rbp m';	6 :	Set rax m_0;	9 :	Add! $carry$ $r10$ rax $r10$;

At line 5, we have $r10 \equiv y_0 \times x_0 \bmod 2^{64}$. At line 7, $rbp \equiv y_0 \times x_0 \times m' \bmod 2^{64}$. At line 8, $rax \equiv y_0 \times x_0 \times m' \times m_0 \bmod 2^{64}$. Finally at line 9, we have $r10 \equiv (y_0 \times x_0 \times m' \times m_0) + (y_0 \times x_0) \equiv (y_0 \times x_0) \times (m' \times m_0 + 1) \bmod 2^{64}$. From the precondition $m' \times m + 1 \equiv 0 \bmod 2^{64}$, we have $m' \times \mathit{radix64}(m_3, m_2, m_1, m_0) + 1 \equiv 0 \bmod 2^{64}$ and $m' \times m_0 + 1 \equiv 0 \bmod 2^{64}$. Hence $r10 \equiv (y_0 \times x_0) \times (m' \times m_0 + 1) \equiv 0 \bmod 2^{64}$. Now $r10$ is a 64-bit register. $r10 \equiv 0 \bmod 2^{64}$ implies $r10 = 0$ on x86_64. Its value can be safely discarded. The equality is essential to the proof of correctness in the Montgomery multiplication program.

Although $r10 \equiv 0 \bmod 2^{64}$ can be verified by modular polynomial equation entailment, the algebraic technique fails to prove $r10 = 0$. In order to verify bn_mul_mont_, we add two instructions following the code fragment: Assert$\top \| r10 \equiv 0 \bmod 2^{64}$ and Assume$\top \| r10 = 0$. The Assert statement is automatically verified; the Assume statement is safe because $r10 \equiv 0 \bmod 2^{64}$ implies $r10 = 0$ when $r10$ is a 64-bit register.

5.2 Arithmetic in Cryptographic Libraries

We have successfully verified assembly codes extracted from the arithmetic programs in cryptographic libraries OpenSSL, boringSSL, and mbedTLS. The extracted traces are not affected by the inputs in all programs except mbedTLS. The big integer multiplication in mbedTLS contains a loop with undetermined iterations in C for propagating carry chains. For this multiplication, we extracted and verified assembly codes for different cases of carry chains but only report two cases with longest carry chains. All the verification tasks are performed on a Linux machine with a 3.47GHz CPU and 128GB memory. We use Boolector

Table 1 Experimental Results.

library	program	ln	assert	range	alg	total
OpenSSL	ecp_nistz256_add	89	0.44	4.17	0.03	4.63
	ecp_nistz256_sub	88	-	18.54	~0	18.55
	ecp_nistz256_from_mont	82	-	0.41	0.02	0.45
	ecp_nistz256_mul_mont	192	-	21.49	0.03	21.53
	ecp_nistz256_mul_mont$^+$	153	-	15.43	0.03	15.47
	ecp_nistz256_mul_by_2	49	-	0.05	0.02	0.08
	ecp_nistz256_sqr_mont	148	-	16.43	0.03	16.47
	ecp_nistz256_sqr_mont$^+$	131	-	22.50	0.03	22.54
	x86_64_mont_2	228	832.60	13.41	0.03	846.05
	x86_64_mont_4	490	8279.87	523.27	0.91	8804.06
boringSSL	x25519_x86_64_mul	226	-	28.73	0.03	28.78
	x25519_x86_64_sqr	171	-	6.14	0.03	6.18
	x25519_x86_64_ladderstep	1459	-	2921.82	107.93	3029.78
mbedTLS	mbedtls_mpi_mul_mpi_2	76	0.46	0.42	0.03	0.92
	mbedtls_mpi_mul_mpi_4	249	12.85	9.27	0.02	22.16

2.4.0 for SMT solving and use Singular 4.1.0 for ideal membership solving. Table 1 shows the verification results. For each arithmetic program, we report the number of lines (ln) in CryptoLine (including the specification), the time in seconds for assertion checking (assert), range checking (range), algebraic checking (alg), and overall verification (total). A "-" in the assertion column indicates that the program contains no assertion. In OpenSSL, two versions of ecp_nistz256_mul_mont and ecp_nistz256_sqr_mont are availble: one for typical x86_64 microarchitectures, the other for Broadwell microarchitecture (annotated by "+" in the table). For multiplication, we verified 2- and 4-limb versions in OpenSSL (x86_64_mont_*) and mbedTLS (mbedtls_mpi_mul_mpi_*).

To the best of our knowledge, our work is the first on automatically verifying assembly codes extracted from low-level arithmetic implementations of industrial cryptographic libraries. Most of the other works verified re-implementations of arithmetic operations written in high-level languages. In the most related work [16], an implementation of the Montgomery Ladderstep in bvCryptoLine was verified in days. With our approach, the implementation of the Montgomery Ladderstep in boringSSL can be verified in 1 hour.

6 Conclusion

We have described a domain-specific language CryptoLine for modeling arithmetic assembly programs in cryptographic primitives across different instruction sets. Scripts have been developed to extract execution traces from programs as assembly codes and to translate assembly codes to CryptoLine. A specification for a program in CryptoLine is divided into range predicates and algebraic predicates. While range predicates are verified by SMT solving, algebraic predicates are verified via transformation to ideal membership problems solved by computer algebra systems. We have implemented our verification approach to successfully verified several arithmetic programs in cryptographic libraries OpenSSL, boringSSL, and mbedTLS.

We are working on a certified translator to accurately generate CryptoLine codes from different assembly. The case studies in mbedTLS expose limitations of verifying cryptographic codes with CryptoLine. We would like to extend our techniques to such programs.

References

1 Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013.

2 Reynald Affeldt and Nicolas Marti. An approach to formal verification of arithmetic functions in assembly. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science*, volume 4435 of *LNCS*, pages 346–360. Springer, 2007.

3 Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Science of Computer Programming*, 77(10–11):1058–1074, 2012.

4 Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, 2015. `doi:10.1145/2701415`.

5 Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *USENIX Security Symposium 2015*, pages 207–221. USENIX Association, 2015.

6 Daniel J. Bernstein and Peter Schwabe. gfverif: Fast and easy verification of finite-field arithmetic, 2016. URL: `http://gfverif.cryptojedi.org`.

7 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*, pages 917–934. USENIX Association, 2017.

8 Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying curve25519 software. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *CCS*, pages 299–309. ACM, 2014.

9 Fiat-crypto. `https://github.com/mit-plv/fiat-crypto`, 2015. Accessed: 2017-05-19.

10 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and company, 1979.

11 John Harrison. Automating elementary number-theoretic proofs using Gröbner bases. In Frank Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 51–66. Springer, 2007.

12 D. Kroening and O. Strichman. *Decision Procedures - an algorithmic point of view.* EATCS. Springer, 2008.

13 Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 66–81. Springer, 2013. `doi:10.1007/978-3-319-03545-1_5`.

14 Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 568–582. Springer, 2007.

15 Aaron Tomb. Automated verification of real-world cryptographic implementations. *IEEE Security & Privacy*, 14(6):26–33, 2016. `doi:10.1109/MSP.2016.125`.

16 Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In David Evans, Tal Malkin, and Dongyan Xu, editors, *CCS*. ACM, 2017.

17 Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. In *CCS*, pages 2007–2020. ACM, 2017. `doi:10.1145/3133956.3133974`.

18 Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *CCS*, pages 1789–1806. ACM, 2017. `doi:10.1145/3133956.3134043`.