Signed Cryptographic Program Verification with Typed CryptoLine

Yu-Fu Fu
Institute of Information Science
Academia Sinica
a0919610611@gmail.com

Jiaxiang Liu
College of Computer Science and Software Engineering
Shenzhen University
jiaxiang0924@gmail.com

Xiaomu Shi
College of Computer Science and Software Engineering
Shenzhen University
xshi0811@gmail.com

Ming-Hsien Tsai
Institute of Information Science
Academia Sinica
mhtsai208@gmail.com

Bow-Yaw Wang
Institute of Information Science
Academia Sinica
bywang@iis.sinica.edu.tw

Bo-Yin Yang
Institute of Information Science
Academia Sinica
byyang@iis.sinica.edu.tw

ABSTRACT

We develop an automated formal technique to specify and verify signed computation in cryptographic programs. In addition to new instructions, we introduce a type system to detect type errors in programs. A type inference algorithm is also provided to deduce types and instruction variants in cryptographic programs. In order to verify signed cryptographic C programs, we develop a translator from the GCC intermediate representation to our language. Using our technique, we have verified 82 C functions in cryptography libraries including NaCl, wolfSSL, bitcoin, OpenSSL, and BoringSSL.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; • **Theory of computation → Verification by model checking**; • **Software and its engineering → Formal software verification**.

KEYWORDS

cryptographic programs; formal verification; model checking

ACM Reference Format:
Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Signed Cryptographic Program Verification with Typed CryptoLine. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354199

1 INTRODUCTION

In 32-bit architectures, two 32-bit unsigned integers u_H and u_L represent a 64-bit unsigned integer $U = u_H \times 2^{32} + u_L$. Let $V = v_H \times 2^{32} + v_L$ be a 64-bit unsigned integer represented by two 32-bit unsigned integers v_H and v_L. It is straightforward to compute the 64-bit unsigned product of U and V. Specifically, $[U \times V] \,\%\, 2^{64} = [u_L \times v_L + 2^{32} \times (u_L \times v_H + u_H \times v_L)] \,\%\, 2^{64}$. In C99, such computation naturally arises when the unsigned product of two `unsigned long long` variables is computed on 32-bit architectures. In this case, only unsigned computation is required.

To obtain the *signed* product of two `long long` integers on 32-bit architectures is more involved. In 32-bit architectures, a 32-bit signed integer s_H together with a 32-bit unsigned integer s_L represent a 64-bit signed integer $s_H \times 2^{32} + s_L$. Let $t_H \times 2^{32} + t_L$ be another 64-bit signed integer represented by a 32-bit signed integer t_H and a 32-bit unsigned integer t_L. The 64-bit signed product of two 64-bit signed integers $s_H \times 2^{32} + s_L$ and $t_H \times 2^{32} + t_L$ is hence $[s_L \times t_L + 2^{32} \times (s_L \times t_H + s_H \times t_L)] \,\%\, 2^{64}$. Note that $s_L \times t_L$ is unsigned but $s_L \times t_H$ and $s_H \times t_L$ are signed. Both signed and unsigned computation is needed to obtain the signed product.

Mixing signed and unsigned computation is in fact more complicated than appeared. Since 32-bit signed and unsigned integers have different ranges, one must ensure the absence of over- and underflow during computation. Moreover, signed integers have been used to represent elements in large finite fields. In such representations, mixing signed and unsigned computation is unavoidable. In order to ensure functional correctness of cryptographic programs, a practical algorithm has to be developed to verify mixed computation in various field and group operations used in cryptographic primitives.

In this paper, we propose an automated technique for verifying mixed signed and unsigned computation in implementations of various operations found in cryptographic primitives. We extend the CryptoLine language with signed instructions. In order to differentiate signed and unsigned expressions, a simple type system is introduced. Type inference moreover is provided to annotate types of variables and variants of instructions automatically. We also extend verification algorithms for CryptoLine with signed computation. Particularly, we employ Satisfiability Modulo Theories (SMT) solvers to verify the absence of overflow, underflow, and range properties. Computer algebra systems also are used to check algebraic properties in signed computation.

Our verification targets are signed C implementations of various field and group operations in cryptographic primitives. Specifically, we verify C implementations in NaCl, wolfSSL, bitcoin, OpenSSL, and BoringSSL. To this end, we additionally build a translator from the intermediate representation used in GNU compilers to CryptoLine. We identify a useful subset of the intermediate representation

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3354199

termed gimpleCryptoLine, give formal semantics, and describe how gimpleCryptoLine programs are translated to CryptoLine.

During verification, we expose a potential programming error in NaCl 20110221 and have reported it. We would also like to point out that the bitcoin cryptographic library is used by various cryptocurrency such as Ethereum [20], Zcash [27], Ripple [25], and Litecoin [22]. We verify 24 C functions for various field and group operations of the secp256k1 curve used by bitcoin. We are not aware of any prior work on verifying cryptographic C functions from cryptocurrency. Moreover, the 32-bit implementation of Curve25519 in wolfSSL represents field elements as 10 signed integers. The same implementation is also found in OpenSSL [24] and LibreSSL [21]. To the best of our knowledge, this is the first formal verification of signed C implementations in cryptographic libraries used in industry.

We summarize our contributions as follows.

- We extend the domain specific language CryptoLine with signed implementations for field and group operations in cryptographic primitives;
- We develop practical techniques for verifying functional correctness of signed CryptoLine programs;
- We build a translator from the intermediate representation of GNU compilers to CryptoLine to enable the verification of C implementations for cryptographic primitives;
- We report verification results of cryptographic C programs from NaCl, wolfSSL, bitcoin, OpenSSL, and BoringSSL.

Related Work. The first semi-automatic verification work on real-world cryptographic assembly implementations was proposed in [8]. The authors applied an SMT solver and a proof assistant to verify an extensively annotated assembly implementation of the Montgomery Ladderstep in 5 hours. The unsigned CryptoLine language and its verification algorithm were proposed in [18] by extending bvCryptoLine [28]. Our gimple translation is motivated by the translator from LLVM intermediate representation to CryptoLine developed in [14]. Vale [7, 10] is a tool and a high-level language for the specification and verification of assembly codes. Jasmin [4] is another framework for developing high-speed and high-assurance cryptographic programs. Both tools use SMT solvers for verification. When SMT solvers fail to verify a property, lemmas can be added manually to help verification. In addition to SMT solvers, our technique utilizes computer algebra systems to check algebraic properties. We also verify widely used cryptographic C programs in this work. HACL* [30] is a verified cryptography library implementing the NaCl cryptographic API. Its implementation is written in the high-level language F*. Its main objective is correctness rather than efficiency. Subsequently, HACL* is not highly optimized (yet). We focus on verifying optimized implementations in existing cryptography libraries. The Fiat-Crypto project synthesizes correct C cryptographic programs [9]. The performance of synthesized 64-bit C programs for Curve25519 is comparable to an x86_64 assembly implementation in BoringSSL. The project exploits a number of features in the proof assistant Coq and requires significant human intervention. Various implementations of algebraic operations, hash functions, and random number generators have been formalized and manually verified in proof assistants (see [1–3, 5, 6, 16, 17, 29] for examples). Our automated technique requires much less human interaction and is friendlier to average cryptography programmers.

The paper is organized as follows. We review preliminaries in Section 2. Section 3 presents CryptoLine with signed computation. The translation from gimpleCryptoLine to CryptoLine is given in Section 4. Section 5 reports experiments on cryptography libraries.

2 PRELIMINARY

Let $\mathbb{Z}$ and $\mathbb{N}$ denote the set of integers and positive integers respectively. Using the binary representation of length w for integers, an integer is represented by a bit string $(b_{w-1}b_{w-2}\cdots b_1b_0)_2$ of w bits $b_i \in \{0,1\}$ for $0 \le i < w$. In the *unsigned* interpretation, the bit string $(b_{w-1}b_{w-2}\cdots b_1b_0)_2$ represents the integer $\sum_{i=0}^{w-1} b_i \times 2^i$. In the two's complement *signed* interpretation, the same bit string represents the integer $-b_{w-1} \times 2^{w-1} + \sum_{i=0}^{w-2} b_i \times 2^i$. For instance, the bit string $(111)_2$ denotes $7(= 2^2 + 2^1 + 2^0)$ under the unsigned interpretation and $-1(= -2^2 + 2^1 + 2^0)$ under the two's complement signed interpretation. To interpret bit strings correctly, it is essential to know their intended interpretations. We only consider the unsigned and two's complement signed interpretations.

3 CRYPTOLINE WITH SIGNED ARITHMETIC

CryptoLine is a domain specific language for modeling cryptographic assembly programs and their specifications [18]. Modern cryptography relies heavily on complex algebraic structures such as large finite fields and groups. In elliptic curve cryptography, for instance, a pair of field elements satisfying a curve equation is a point on the elliptic curve. Points on the curve in turn form a group. These field and group operations are frequently used and thus critical to the performance of cryptosystems. In order to speed up computation, assembly programs are written to implement various field and group operations in practice. In OpenSSL, dedicated ARMv8 assembly programs for NIST P256 can outperform corresponding C implementations by 4 times! Indeed, a wide selection of assembly programs can be found in OpenSSL for various cryptosystems and architectures. If any of them computes incorrectly, security of cryptosystems can be compromised. In order to improve security of cryptosystems in use, CryptoLine is developed to help programmers write correct cryptographic assembly programs.

The first challenge in modeling assembly programs is diversity. Different architectures have different instruction sets, register banks, condition flags, and even word sizes. In CryptoLine, a useful set of instructions had been identified for modeling cryptographic assembly programs [18]. Its semantics however assumed unsigned representations. Programmers are hence forced to represent field elements as limbs of unsigned integers. In order to allow signed representations in cryptographic programs, we extend CryptoLine with signed instructions. It turns out that we have to introduce a type system with a type inference algorithm for the signed extension. The verification algorithm also needs to be expanded extensively. In this section, we explain our signed extension to the CryptoLine language and its verification algorithm.

3.1 Syntax

We introduce a type system to track interpretations of variables and constants in CryptoLine (Figure 1). Let w be a positive integer. The

$$
\begin{array}{rcl}
\mathit{Width} & ::= & 1 \mid 2 \mid 3 \mid \cdots \\
\mathit{Type} & ::= & \mathsf{uint}\ \mathit{Width} \mid \mathsf{sint}\ \mathit{Width}
\end{array}
$$

Figure 1: CryptoLine Type

type uint w corresponds to integers denoted by bit strings of length w in the unsigned interpretation. Similarly, sint w corresponds to integers denoted by bit strings of length w in the two's complement signed interpretation. A type only denotes a finite subset of integers. Precisely, uint w denotes integers greater than -1 and less than 2^w; sint w denotes integers greater than $-2^{w-1}-1$ and less than 2^{w-1}. We use $\underline{\tau}$ and $\overline{\tau}$ for the lower and upper bounds of integers denoted by the type τ. The type τ thus denotes the set $\{i \in \mathbb{Z} : \underline{\tau} < i < \overline{\tau}\}$. For instance, $\underline{\mathsf{uint}\ 4} = -1$ and $\overline{\mathsf{uint}\ 4} = 16$; $\underline{\mathsf{sint}\ 4} = -9$ and $\overline{\mathsf{sint}\ 4} = 8$. If an integer belongs to the set denoted by the type τ, the integer is *representable* in τ. For conveniences, bit is short for uint 1.

Constants in CryptoLine are associated with a type. 15@uint 4 and (−1)@sint 4 denote the unsigned integer 15 and signed integer −1 respectively. Both are represented by the bit string $(1111)_2$, though. Types of variables must be specified by *declarations*. For instance, uint 32 x and sint 64 y declare a 32-bit unsigned variable x and a 64-bit signed variable y respectively. An atom is either a variable or a constant. A CryptoLine program consists of variable declarations followed by instructions (Figure 2).

A typical instruction retrieves values from *sources* and stores results in *destinations*. In mov v a, the value of the source a is stored in the destination v. Depending on the value of c, cmov v c a_0 a_1 stores either value of sources a_0 or a_1 in the destination v. Converting values between different types is explicit in CryptoLine. The cast (v@uint 16) x instruction casts the value of the source x to a 16-bit unsigned integer and stores the value in the destination v.

CryptoLine instructions have both unsigned and signed *variants*. For instance, uadd r x (1@uint 32) stores the sum of the 32-bit unsigned variable x and the 32-bit unsigned constant 1 in the 32-bit unsigned destination r whereas sadd s y (1@sint 32) stores the sum of the signed variable y and the signed constant 1 in the 32-bit signed destination s. Typical arithmetic instructions are supported in CryptoLine. uadd and sadd are addition instructions; uadds and sadds are carrying addition instructions; uadc and sadc are addition with carry instructions; uadcs and sadcs are carrying addition with carry instructions. Various subtraction instructions (usub, usubs, usbb, usbbs) are allowed as well as their signed variants (ssub, ssubs, ssbb, ssbbs). umul and umull are the unsigned half- and full-multiplication instructions respectively. The corresponding signed variants are smul and smull. Left bit-shift instructions (ushl and sshl) are also provided for conveniences.

For bit masking in cryptographic programs, CryptoLine offers the uspl and sspl instructions. For instance, the instruction uspl u_H u_L ($0x1000$@uint 16) 12 assigns $0x1$ and $0x0$ to the 16-bit unsigned destinations u_H and u_L respectively. Observe that the $4(= 16 - 12)$ most and the 12 least significant bits of $0x1000$ are $0x1$ and $0x0$ respectively. The corresponding signed instruction stores most significant bits in a signed destination. Consider the signed instruction sspl s_H s_L (−$0x1000$@sint 16) 12. The 16-bit unsigned destination s_L still gets the 12 least significant bits $0x0$. The signed 16-bit destination s_H however obtains −1. The join instructions ujoin and sjoin have the reverse effect. For instance, ujoin v ($0x10$@uint 8) ($0x00$@uint 8) stores the 16-bit value $0x1000$ in the unsigned destination v.

Another common pattern in cryptography programming is bit concatenation followed by shifting. Such a pattern is often found in carry propagation in unsaturated representations of field elements. The ucshl u_H u_L ($0x0011$@uint 16) ($0x2222$@uint 16) 8 instruction concatenates $0x0011$ with $0x2222$, shifts the concatenated result ($0x00112222$) 8 bits to the left, splits the shifted result ($0x11222200$) into two 16-bit unsigned values ($0x1122$ and $0x2200$), right-shifts the less significant value by 8 bits, then stores the results in the two 16-bit unsigned destinations. Thus $u_H = 0x1122$ and $u_L = 0x0022$. The corresponding signed instruction is almost identical except it splits the shifted result into a signed and an unsigned 16-bit values.

Finally, an expression is an atom, or the sum, difference, product of two expressions. An algebraic predicate is a conjunction of equations or modulo equations. A range predicate is a conjunction of comparisons on expressions. Given an algebraic predicate P and a range predicate Q, assert $P \curlywedge Q$ aborts with an error if their conjunction does not hold. assume $P \curlywedge Q$ aborts without error if the conjunction does not hold.

3.2 Type System and Inference

Recall the computation of the product of two `long long` variables in Section 1. Its mixed signed and unsigned computation must be interpreted carefully to obtain correct results. If an unsigned bit string is misinterpreted as a signed one or vice versa, incorrect results will be computed and contaminate cryptographic primitives. Keeping track of unsigned and signed interpretations can be tedious. Assembly programmers have to remember interpretations of memory cells and registers so as to implement field and group operations in cryptographic primitives correctly.

In order to identify misinterpretation, we introduce a simple type system in CryptoLine with signed computation. Let w be a positive integer. An *unsigned* type ρ is of the form uint w; a *signed* type σ is of the form sint w. We also use τ to denote a (signed or unsigned) type. Two types τ, τ' are *compatible* (written $\tau \| \tau'$) if they are of the same bit length. We also write $2 \bullet \tau$ for the type with double length of τ. For instance, sint 16‖uint 16 and $2 \bullet$ (sint 16) denotes sint 32. A *variable typing relation* $v : \tau$ specifies the type τ for the variable v. A *type context* Γ is a set of variable typing relations. Figure 3 gives the type system for CryptoLine.

Each declaration in CryptoLine specifies a type for a variable and hence defines a variable typing relation. Let Γ be the type context composed of all variable typing relations in a CryptoLine program. An instruction *inst* is *typable* in Γ if $\Gamma \vdash$ *inst*. Typable expressions and predicates are defined similarly.

The mov v a instruction is typable if v and a are of the same type. The conditional move instruction cmov v c a_0 a_1 is typable if v, a_0, a_1 are of the same type and c is of the type bit. The cast instruction cast v@τ a expects v to be of the designated type τ.

Most arithmetic instructions require sources and destinations are of the same type. uadd v a_0 a_1 expects v, a_0, a_1 to have the same unsigned type; uadds c v a_0 a_1 additionally requires c is of the type bit. Addition with carry is similar. uadc v a_0 a_1 d is typable if v, a_0, a_1 have the same unsigned type and an additional carry d has the type

$$
\begin{array}{rcl}
\mathit{Num} & ::= & \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Const} & ::= & \mathit{Num}@\mathit{Type} \\
\mathit{Var} & ::= & \cdots \mid x \mid y \mid z \mid \cdots \\
\mathit{Atom} & ::= & \mathit{Var} \mid \mathit{Const} \\
\mathit{Exp} & ::= & \mathit{Atom} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid \mathit{Exp} \times \mathit{Exp} \\
\mathit{APred} & ::= & \mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} \equiv \mathit{Exp} \bmod \mathit{Exp} \mid \mathit{APred} \wedge \mathit{APred} \\
\mathit{RPred} & ::= & \mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} < \mathit{Exp} \mid \mathit{RPred} \wedge \mathit{RPred} \\
\mathit{Inst} & ::= & \mathsf{mov}\ \mathit{Var}\ \mathit{Atom} \mid \mathsf{cmov}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{cast}\ \mathit{Var}@\mathit{Type}\ \mathit{Atom} \\
 & \mid & \mathsf{uadd}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{uadds}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{uadc}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{uadcs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{sadd}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{sadds}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{sadc}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{sadcs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{usub}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{usubs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{usbb}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{usbbs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{ssub}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{ssubs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{ssbb}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{ssbbs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{umul}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{smul}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{umull}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{smull}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{ushl}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \mid \mathsf{sshl}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \mid \mathsf{uspl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \mid \mathsf{ucshl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Num} \\
 & \mid & \mathsf{ujoin}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{assert}\ \mathit{APred} \curlywedge \mathit{RPred} \mid \mathsf{sspl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \mid \mathsf{scshl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Num} \\
 & \mid & \mathsf{sjoin}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \mid \mathsf{assume}\ \mathit{APred} \curlywedge \mathit{RPred} \\
\mathit{Decl} & ::= & \mathit{Type}\ \mathit{Var} \qquad \mathit{Prog} ::= \mathit{Decl}^*\ \mathit{Inst}^*
\end{array}
$$

Figure 2: CryptoLine Syntax

$$
\frac{}{\Gamma, v : \tau \vdash v : \tau} \qquad \frac{}{\Gamma \vdash c@\tau : \tau}
$$

$$
\frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash E_0 + E_1 : \tau} \qquad \frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash E_0 - E_1 : \tau}
$$

$$
\frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash E_0 \times E_1 : \tau} \qquad \frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash E_0 = E_1} \qquad \frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_0 \equiv E_1 \bmod E_2} \qquad \frac{\Gamma \vdash E_0 : \tau \quad \Gamma \vdash E_1 : \tau}{\Gamma \vdash E_0 < E_1}
$$

$$
\frac{\Gamma \vdash P_0 \quad \Gamma \vdash P_1}{\Gamma \vdash P_0 \wedge P_1} \qquad \frac{\Gamma \vdash a : \tau}{\Gamma, v : \tau \vdash \mathsf{mov}\ v\ a} \qquad \frac{\Gamma \vdash c : \mathsf{bit} \quad \Gamma \vdash a_0 : \tau \quad \Gamma \vdash a_1 : \tau}{\Gamma, v : \tau \vdash \mathsf{cmov}\ v\ c\ a_0\ a_1} \qquad \frac{\Gamma \vdash a : \tau'}{\Gamma, v : \tau \vdash \mathsf{cast}\ v@\tau\ a}
$$

$$
\frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, v : \rho \vdash \mathsf{uadd}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, c : \mathsf{bit}, v : \rho \vdash \mathsf{uadds}\ c\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, v : \rho \vdash \mathsf{uadc}\ v\ a_0\ a_1\ d} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, c : \mathsf{bit}, v : \rho \vdash \mathsf{uadcs}\ c\ v\ a_0\ a_1\ d}
$$

$$
\frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma}{\Gamma, v : \sigma \vdash \mathsf{sadd}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma}{\Gamma, c : \mathsf{bit}, v : \sigma \vdash \mathsf{sadds}\ c\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, v : \sigma \vdash \mathsf{sadc}\ v\ a_0\ a_1\ d} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, c : \mathsf{bit}, v : \sigma \vdash \mathsf{sadcs}\ c\ v\ a_0\ a_1\ d}
$$

$$
\frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, v : \rho \vdash \mathsf{usub}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, c : \mathsf{bit}, v : \rho \vdash \mathsf{usubs}\ c\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, v : \rho \vdash \mathsf{usbb}\ v\ a_0\ a_1\ d} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, c : \mathsf{bit}, v : \rho \vdash \mathsf{usbbs}\ c\ v\ a_0\ a_1\ d}
$$

$$
\frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma}{\Gamma, v : \sigma \vdash \mathsf{ssub}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma}{\Gamma, c : \mathsf{bit}, v : \sigma \vdash \mathsf{ssubs}\ c\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, v : \sigma \vdash \mathsf{ssbb}\ v\ a_0\ a_1\ d} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma \quad \Gamma \vdash d : \mathsf{bit}}{\Gamma, c : \mathsf{bit}, v : \sigma \vdash \mathsf{ssbbs}\ c\ v\ a_0\ a_1\ d}
$$

$$
\frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, v : \rho \vdash \mathsf{umul}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma}{\Gamma, v : \sigma \vdash \mathsf{smul}\ v\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \rho \quad \Gamma \vdash a_1 : \rho}{\Gamma, v_H : \rho, v_L : \rho \vdash \mathsf{umull}\ v_H\ v_L\ a_0\ a_1} \qquad \frac{\Gamma \vdash a_0 : \sigma \quad \Gamma \vdash a_1 : \sigma \quad \sigma \| \rho}{\Gamma, v_H : \sigma, v_L : \rho \vdash \mathsf{smull}\ v_H\ v_L\ a_0\ a_1}
$$

$$
\frac{\Gamma \vdash a : \rho}{\Gamma, v : \rho \vdash \mathsf{ushl}\ v\ a\ n} \qquad \frac{\Gamma \vdash a : \sigma}{\Gamma, v : \sigma \vdash \mathsf{sshl}\ v\ a\ n} \qquad \frac{\Gamma \vdash a_H : \rho \quad \Gamma \vdash a_L : \rho}{\Gamma, v_H : \rho, v_L : \rho \vdash \mathsf{ucshl}\ v_H\ v_L\ a_H\ a_L\ n} \qquad \frac{\Gamma \vdash a_H : \sigma \quad \Gamma \vdash a_L : \rho \quad \sigma \| \rho}{\Gamma, v_H : \sigma, v_L : \rho \vdash \mathsf{scshl}\ v_H\ v_L\ a_H\ a_L\ n}
$$

$$
\frac{\Gamma \vdash a_H : \rho \quad \Gamma \vdash a_L : \rho}{\Gamma, v : 2 \bullet \rho \vdash \mathsf{ujoin}\ v\ a_H\ a_L} \qquad \frac{\Gamma \vdash a_H : \sigma \quad \Gamma \vdash a_L : \rho \quad \sigma \| \rho}{\Gamma, v : 2 \bullet \sigma \vdash \mathsf{sjoin}\ v\ a_H\ a_L} \qquad \frac{\Gamma \vdash a : \rho}{\Gamma, v_H : \rho, v_L : \rho \vdash \mathsf{uspl}\ v_H\ v_L\ a\ n} \qquad \frac{\Gamma \vdash a : \sigma \quad \sigma \| \rho}{\Gamma, v_H : \sigma, v_L : \rho \vdash \mathsf{sspl}\ v_H\ v_L\ a\ n}
$$

$$
\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{assert}\ P \curlywedge Q} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \mathsf{assume}\ P \curlywedge Q} \qquad \frac{}{\Gamma \vdash} \qquad \frac{\Gamma \vdash \mathit{inst} \quad \Gamma \vdash \mathit{insts}}{\Gamma \vdash \mathit{inst}\ \mathit{insts}}
$$

Figure 3: CryptoLine Type System

bit. uadcs $c\ v\ a_0\ a_1\ d$ moreover requires the carry c to be of the type bit. Signed addition instructions sadd, sadds, sadc, sadcs similarly require sources and destinations are of the same signed type and carries of the type bit. There is no surprise for subtraction instructions. Unsigned subtraction instructions usub, usubs, usbb, usbbs are typable when sources and destinations have the same unsigned type with additional borrow or borrowing bits if needed. Signed subtraction instructions must have signed sources and destinations instead. Unsigned half-multiplication umul $v\ a_0\ a_1$ requires v, a_0, and a_1 to have the same unsigned type. Signed half-multiplication smul is typable if sources and destinations are of the same signed type. The left-shift instructions ushl $v\ a\ n$ and sshl $v\ a\ n$ are as expected. The destination v is required to have the same type as a.

Full multiplication however is slightly surprising. The unsigned full-multiplication umull $v_H\ v_L\ a_0\ a_1$ is typable if v_H, v_L, a_0, a_1 have the same unsigned type. Suppose a_0 and a_1 are of the type uint w. Their product then needs bit length $2w$. The w least significant bits are put in v_L and the remaining bits are stored in v_H. Thus both v_H and v_L are of the same unsigned type uint w. Now consider the signed full-multiplication smull $v_H\ v_L\ a_0\ a_1$. Suppose a_0 and a_1 are of the signed type sint w. Their signed product therefore has bit length $2w$ ($2w - 1$ bits for magnitude and 1 bit for sign). The w least significant bits are unsigned and assigned to v_L. The

w most significant bits are signed and stored in v_H of the type sint w. Using different interpretations for v_H and v_L maintains the equation $v_H \times 2^w + v_L = a_0 \times a_1$ (see Section 3.3 for details).

For the source a of the type uint w, the uspl v_H v_L a n instruction splits the bit string for a. The $w - n$ most significant bits are stored in v_H and the remaining n bits are in v_L. Both v_H and v_L are required to have the type uint w. To split a source a of the type sint w, sspl v_H v_L a n is used. Similar to signed full-multiplication, the $w - n$ most significant bits are signed and put in v_H; the n least significant bits are unsigned and stored in v_L. Subsequently, v_H and v_L must be of the types sint w and uint w respectively. For sources a_H and a_L of the type uint w, ujoin v a_H a_L concatenates the bit strings of a_H and a_L and requires v to have the type uint $2w$. Similarly, v must have the type sint $2w$ in sjoin v a_H a_L where a_H and a_L are of the types sint w and uint w respectively.

Concatenation followed by left-shift instructions combine left-shift and split instructions. The unsigned ucshl v_H v_L a_H a_L n takes a_H and a_L of the type uint w and requires the destinations v_H and v_L to have the same type. On the other hand, the signed scshl v_H v_L a_H a_L n requires a_H and a_L to have the compatible types sint w and uint w respectively. The destinations v_H and v_L should also be of the types sint w and uint w respectively.

Finally, assert $P \curlywedge Q$ and assume $P \curlywedge Q$ require the algebraic predicate P and range predicate Q to be typable in the given context. A sequence of instructions is typable if each instruction is typable.

From Figure 3, it is not hard to see that types of destinations are determined by instructions. It is subsequently not necessary to declare types for all variables in a CryptoLine program. Types of variables containing intermediate computation results can be inferred automatically. Figure 4 gives a type inference algorithm for CryptoLine type system.

Given a type context, each type inference rule updates the type context with variable typing relations for destinations. For instance, the [Cast] rule requires a to have the type τ' in the type context Γ ($\Gamma \vdash a : \tau'$). If so, the instruction cast $v@\tau$ a updates the type context and obtains a new type context $\Gamma \uplus \{v : \tau\}$. Compared to the corresponding rule in CryptoLine type system (Figure 3), the inference rule does not require the destination v to be typable in Γ. Rather, it updates Γ with the variable typing relation for v and hence implicitly declares the type of v to be τ ($\Gamma \vdash \text{cast } v@\tau\ a \Rightarrow \Gamma \uplus \{v : \tau\}$). All type rules can be reformulated as inference rules straightforwardly. We illustrate the basic ideas in selected examples.

In uadcs c v a_0 a_1 d, the sources a_0, a_1 need to have an unsigned type ρ and d the type bit in the current type context. If so, the type context is updated with the variable typing relations c : bit and $v : \rho$. Thus c and v effectively have the respective types bit and ρ afterwards. Similarly, smull v_H v_L a_0 a_1 requires a_0, a_1 to have a signed type σ. After the instruction, the type context is updated with the variable typing relations $v_H : \sigma$ and $v_L : \rho$ where types σ and ρ are compatible. If $a_H : \sigma$ and $a_L : \rho$ with compatible types σ and ρ are typable in the current type context, scshl v_H v_L a_H a_L n adds $v_H : \sigma$ and $v_L : \rho$ to the type context. Finally, ujoin v a_H a_L declares v to have the type $2 \bullet \rho$ if a_H and a_L are of the type ρ.

Starting from the first instruction, the [Inst] rule updates the given type context and uses the updated type context for the remaining instructions. With the type inference rules in Figure 4, it suffices to declare types of uninitialized variables in the initial type context. The type inference rules will annotate all variables used in a CryptoLine program automatically without user intervention. The type inference rules hence greatly improve the usability.

Our implementation goes even further. In Figure 3 and 4, it is easily seen that types of sources determine the variant of instructions. Intended variants of CryptoLine instructions can hence be decided automatically. Consider, for instance, a generic addition instruction with two unsigned sources. The addition instruction is easily seen to be unsigned because the signed addition requires two signed sources. Subsequently, it suffices to write add v a_0 a_1. CryptoLine type inference will determine whether the unsigned uadd v a_0 a_1 or signed sadd v a_0 a_1 is needed. Users simply write generic mnemonics for each instruction (say, add). CryptoLine will choose the intended variant (uadd or sadd) automatically.

Allowing generic mnemonics in CryptoLine is more than for users' conveniences. When verifying cryptographic assembly programs, it is crucial to distinguish unsigned from signed interpretations. Yet not all assembly instructions indicate variants of operations clearly. The x86 add instruction, for example, is used for both variants of addition. For such instructions, it is unclear which variants of instructions are intended by programmers. CryptoLine users of course could guess programmers' intention and annotate instructions accordingly, but they might also misinterpret programmers' intention and verify incorrectly annotated programs. Generic mnemonics in CryptoLine relieve tedious and possibly harmful annotations during verification. Users can greatly benefit from these simple yet useful features in the CryptoLine type system.

3.3 Semantics

CryptoLine is designed to model cryptographic assembly programs. In order to model overflow, underflow, and even CPU flags in such programs, we give a bit-accurate semantics for CryptoLine. Following the standard operational semantics of imperative languages [12, 15], a program state is formalized by an environment. Formally, an *environment* ϵ is a mapping from variables to bit strings. Note that variables are mapped to bit strings, not values.

Using bit strings can be tedious sometimes. A bit string may denote different values in different interpretations. Conversely, a value can be represented by different bit strings under different interpretations. For instance, $(1111)_2$ denotes 15 in uint 4 but -1 in sint 4; -1 can be represented by $(1111)_2$ in sint 4 or $(11111111)_2$ in sint 8. It is essential to specify interpretations before representing values in the semantics. Particularly, it is ambiguous to update the variable v in an environment ϵ with the value -1 since both $(1111)_2$ and $(11111111)_2$ represent -1 in sint 4 and sint 8 respectively. One has to specify the type of v so as to update its bit string correctly. The CryptoLine type system luckily provides the needed typing information. Let ϵ be an environment, v a variable, and V a value. We write $\epsilon[v \mapsto V]$ for the environment obtained by updating the bit string for v in ϵ with the bit string representing V *(interpreted) in the type of* v. Thus $\epsilon[v \mapsto -1](v) = (1111)_2$ when v : sint 4 but $\epsilon[v \mapsto -1](v) = (11111111)_2$ when v : sint 8.

Figure 5 gives the semantics for CryptoLine arithmetic instructions. Recall an atom a is either a variable v or a constant $c@\tau$ with a type τ. When $a : \tau$, $[\![a]\!]^\tau_\epsilon$ denotes the value of a in ϵ interpreted in the type τ. The mov v a instruction simply updates the bit string

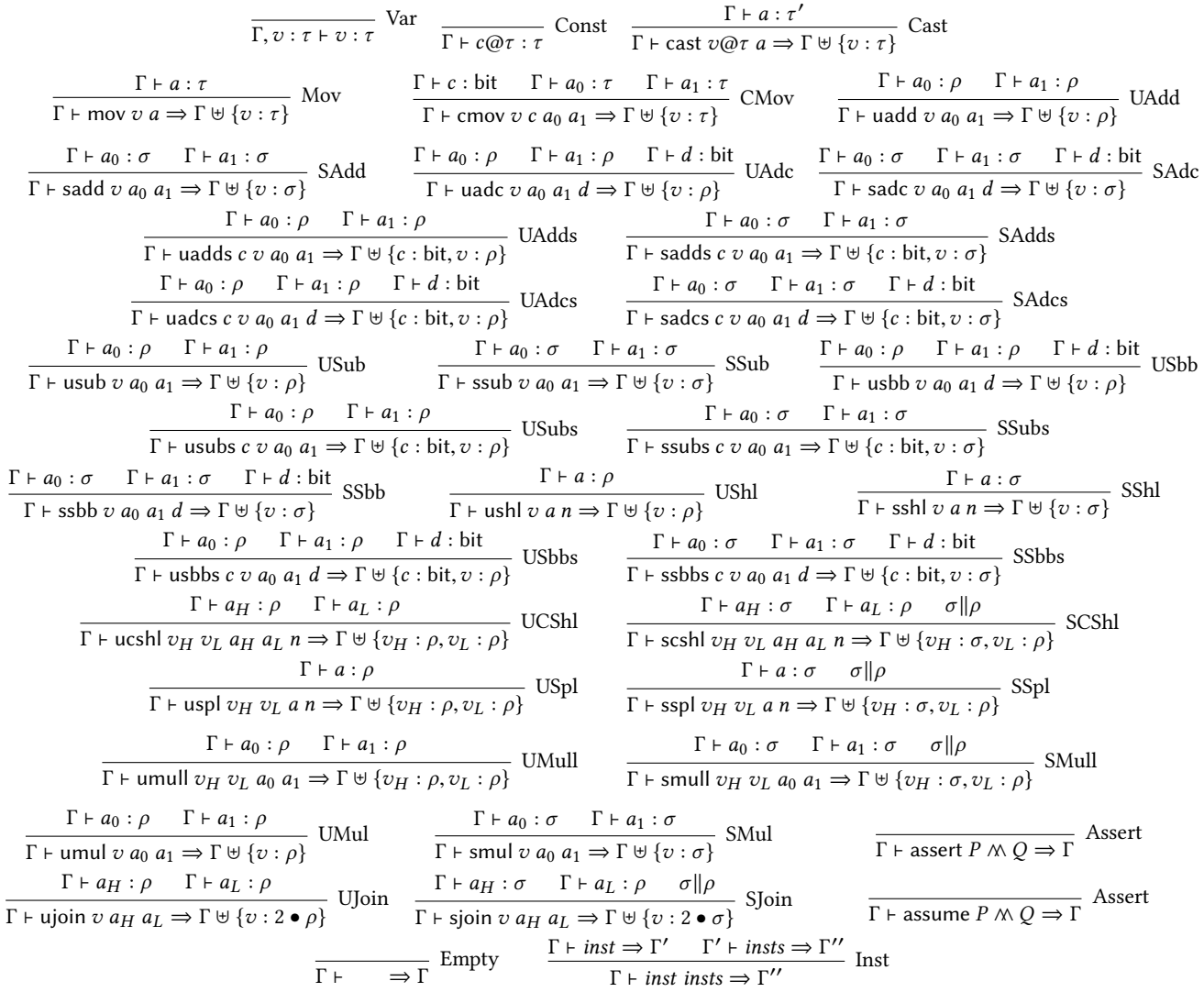

Figure 4: CryptoLine Type Inference

for v with the bit string representing the value of a in the type of v. Similarly, cmov $v\ c\ a_0\ a_1$ updates v with the bit string representing the value of a_0 or a_1 depending on the value of the bit c.

Unsigned addition instructions are essentially those in [18] enriched with typing information. uadd $v\ a_0\ a_1$ updates the bit string for v with the bit string representing the sum of a_0 and a_1 if the sum is representable in the type of v; otherwise, the instruction yields the error environment $\perp$. The addition with carry instruction uadc $v\ a_0\ a_1\ d$ updates the bit string for v with the bit string representing the sum of a_0, a_1, d if the sum is representable in the type of v; otherwise, it yields the error environment. Carrying addition instructions uadds $c\ v\ a_0\ a_1$ and uadcs $c\ v\ a_0\ a_1\ d$ never err. The bit c is 1 iff the sum is not representable in the unsigned type of v.

The signed uncarrying addition instructions sadd $v\ a_0\ a_1$ and sadc $v\ a_0\ a_1\ d$ are similar. If the sum is representable in the signed type of v, the bit string for v is updated accordingly. Otherwise, these instructions yield the error environment. The signed carrying addition instructions are slightly curious. The sadds $c\ v\ a_0\ a_1$ instruction updates the bit string for v if the sum of a_0 and a_1 is representable in the type σ of v. In contrast to its unsigned counterpart, sadds will yield the error environment when the sum is not representable in σ. Moreover, the carrying bit c is computed by interpreting the bit strings of a_0 and a_1 in the unsigned type ρ compatible with σ. If the sum in the unsigned interpretation is representable in ρ, the carrying bit c is set to 0; otherwise, it is set to 1. In assembly, the carry flag is computed as if sources are unsigned since there is no typing information. Our bit-accurate semantics is designed to mimic the semantics of assembly instructions. The semantics of the signed carrying addition with carry instruction sadcs $c\ v\ a_0\ a_1\ d$ is defined similarly.

Unsigned subtraction instructions usub $v\ a_0\ a_1$, usubs $c\ v\ a_0\ a_1$, usbb $v\ a_0\ a_1\ d$, and usbbs $c\ v\ a_0\ a_1\ d$ again are similar to those

$[\![v]\!]^{\tau}_{\epsilon}$ = the value represented by $\epsilon(v)$ in τ for $v \in \mathit{Var}$ $\quad$ $[\![c@\tau]\!]^{\tau}_{\epsilon} = c$ for $c@\tau \in \mathit{Const}$

$\epsilon \xrightarrow{\text{mov } v\ a} \epsilon[v \mapsto [\![a]\!]^{\tau}_{\epsilon}] \quad (v, a : \tau)$

$\epsilon \xrightarrow{\text{cmov } v\ c\ a_0\ a_1} \epsilon[v \mapsto V] \quad (c : \text{bit}; v, a_0, a_1 : \tau) \quad V = \begin{cases} [\![a_0]\!]^{\tau}_{\epsilon} & \text{if } [\![c]\!]^{\text{bit}}_{\epsilon} = 1 \\ [\![a_1]\!]^{\tau}_{\epsilon} & \text{if } [\![c]\!]^{\text{bit}}_{\epsilon} = 0 \end{cases}$

$\epsilon \xrightarrow{\text{uadd } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \rho) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } V < \overline{\rho} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{uadds } c\ v\ a_0\ a_1} \epsilon[c, v \mapsto C, V] \quad (c : \text{bit}; v, a_0, a_1 : \rho) \quad \tilde{V} = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon}, (C, V) = \begin{cases} (0, \tilde{V}) & \text{if } \tilde{V} < \overline{\rho} \\ (1, \tilde{V} - \overline{\rho}) & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{uadc } v\ a_0\ a_1\ d} \epsilon' \quad (v, a_0, a_1 : \rho; d : \text{bit}) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon} + [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } V < \overline{\rho} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{uadcs } c\ v\ a_0\ a_1\ d} \epsilon[c, v \mapsto C, V] \quad (c, d : \text{bit}; v, a_0, a_1 : \rho) \quad \tilde{V} = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon} + [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } (C, V) = \begin{cases} (0, \tilde{V}) & \text{if } \tilde{V} < \overline{\rho} \\ (1, \tilde{V} - \overline{\rho}) & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{sadd } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} + [\![a_1]\!]^{\sigma}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{sadds } c\ v\ a_0\ a_1} \epsilon' \quad (c : \text{bit}; v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} + [\![a_1]\!]^{\sigma}_{\epsilon}, U = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon}, \sigma \| \rho,$
$C = \begin{cases} 0 & \text{if } U < \overline{\rho} \\ 1 & \text{otherwise} \end{cases} \text{ and } \epsilon' = \begin{cases} \epsilon[c, v \mapsto C, V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{sadc } v\ a_0\ a_1\ d} \epsilon' \quad (v, a_0, a_1 : \sigma; d : \text{bit}) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} + [\![a_1]\!]^{\sigma}_{\epsilon} + [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{sadcs } c\ v\ a_0\ a_1\ d} \epsilon' \quad (c, d : \text{bit}; v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} + [\![a_1]\!]^{\sigma}_{\epsilon} + [\![d]\!]^{\text{bit}}_{\epsilon}, U = [\![a_0]\!]^{\rho}_{\epsilon} + [\![a_1]\!]^{\rho}_{\epsilon} + [\![d]\!]^{\text{bit}}_{\epsilon}, \sigma \| \rho,$
$C = \begin{cases} 0 & \text{if } U < \overline{\rho} \\ 1 & \text{otherwise} \end{cases} \text{ and } \epsilon' = \begin{cases} \epsilon[c, v \mapsto C, V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{usub } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \rho) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\rho} < V \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{usubs } c\ v\ a_0\ a_1} \epsilon[c, v \mapsto C, V] \quad (c : \text{bit}; v, a_0, a_1 : \rho) \quad \tilde{V} = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon} \text{ and } (C, V) = \begin{cases} (0, \tilde{V}) & \text{if } \underline{\rho} < \tilde{V} \\ (1, \tilde{V} + \overline{\rho}) & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{usbb } v\ a_0\ a_1\ d} \epsilon' \quad (v, a_0, a_1 : \rho; d : \text{bit}) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon} - [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\rho} < V \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{usbbs } c\ v\ a_0\ a_1\ d} \epsilon[c, v \mapsto C, V] \quad (c, d : \text{bit}; v, a_0, a_1 : \rho) \quad \tilde{V} = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon} - [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } (C, V) = \begin{cases} (0, \tilde{V}) & \text{if } \underline{\rho} < \tilde{V} \\ (1, \tilde{V} + \overline{\rho}) & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{ssub } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} - [\![a_1]\!]^{\sigma}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{ssubs } c\ v\ a_0\ a_1} \epsilon' \quad (c : \text{bit}; v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} - [\![a_1]\!]^{\sigma}_{\epsilon}, U = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon}, \sigma \| \rho,$
$C = \begin{cases} 0 & \text{if } \underline{\rho} < U \\ 1 & \text{otherwise} \end{cases} \text{ and } \epsilon' = \begin{cases} \epsilon[c, v \mapsto C, V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{ssbb } v\ a_0\ a_1\ d} \epsilon' \quad (v, a_0, a_1 : \sigma; d : \text{bit}) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} - [\![a_1]\!]^{\sigma}_{\epsilon} - [\![d]\!]^{\text{bit}}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{ssbbs } c\ v\ a_0\ a_1\ d} \epsilon' \quad (c, d : \text{bit}; v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} - [\![a_1]\!]^{\sigma}_{\epsilon} - [\![d]\!]^{\text{bit}}_{\epsilon}, U = [\![a_0]\!]^{\rho}_{\epsilon} - [\![a_1]\!]^{\rho}_{\epsilon} - [\![d]\!]^{\text{bit}}_{\epsilon}, \sigma \| \rho$
$C = \begin{cases} 0 & \text{if } \underline{\rho} < U \\ 1 & \text{otherwise} \end{cases} \text{ and } \epsilon' = \begin{cases} \epsilon[c, v \mapsto C, V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{ushl } v\ a\ n} \epsilon' \quad (v, a : \rho) \quad V = [\![a]\!]^{\rho}_{\epsilon} \times 2^n \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } V < \overline{\rho} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{sshl } v\ a\ n} \epsilon' \quad (v, a : \sigma) \quad V = [\![a]\!]^{\sigma}_{\epsilon} \times 2^n \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{umul } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \rho) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} \times [\![a_1]\!]^{\rho}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } V < \overline{\rho} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{smul } v\ a_0\ a_1} \epsilon' \quad (v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} \times [\![a_1]\!]^{\sigma}_{\epsilon} \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$

$\epsilon \xrightarrow{\text{umull } v_H\ v_L\ a_0\ a_1} \epsilon[v_H, v_L \mapsto V_H, V_L] \quad (v_H, v_L, a_0, a_1 : \rho) \quad V = [\![a_0]\!]^{\rho}_{\epsilon} \times [\![a_1]\!]^{\rho}_{\epsilon}, V_H \times \overline{\rho} + V_L = V \text{ and } \underline{\rho} < V_L < \overline{\rho}$

$\epsilon \xrightarrow{\text{smull } v_H\ v_L\ a_0\ a_1} \epsilon[v_H, v_L \mapsto V_H, V_L](v_H, a_0, a_1 : \sigma; v_L : \rho, \sigma \| \rho) \quad V = [\![a_0]\!]^{\sigma}_{\epsilon} \times [\![a_1]\!]^{\sigma}_{\epsilon}, V_H \times \overline{\rho} + V_L = V \text{ and } \underline{\rho} < V_L < \overline{\rho}$

Figure 5: CryptoLine Semantics

in [18]. For usub and usbb, the error environment is yielded if the difference is negative. On the other hand, the usubs and usbbs instructions never yield the error environment. The signed subtraction instructions follow a similar line as signed addition instructions. ssub $v\ a_0\ a_1$ and ssbb $v\ a_0\ a_1\ d$ update the bit string for v if the differences $a_0 - a_1$ and $a_0 - a_1 - d$ are representable in the signed type of v respectively. Otherwise, both instructions yield the error environment. The borrowing subtraction instructions ssubs $c\ v\ a_0\ a_1$ and ssbbs $c\ v\ a_0\ a_1\ d$ still yield the error environment if the differences are not representable in the type σ of v. Otherwise, the bit string for v is updated with the bit string representing differences in σ. The borrowing bit c moreover is updated as if the sources are unsigned. This is the semantics consistent with assembly as well.

The left-shift (ushl $v\ a\ n$, sshl $v\ a\ n$) and half-multiplication (umul $v\ a_0\ a_1$, smul $v\ a_0\ a_1$) instructions are as usual. If the result is representable in the type τ of v, the bit string for v is updated with the bit string representing the result interpreted in the type τ. Otherwise, the error environment is yielded.

Recall that the product of two integers representable in the type τ is representable in the type $2\bullet\tau$. In full-multiplication, the product is splitted into two parts and stored in two destinations of types compatible with τ. The following lemma will be useful in defining the semantics of full-multiplication instructions.

Lemma 3.1. *Consider bit strings $\overline{b} = (b_{w-1}b_{w-2}\cdots b_0)_2$ of length w, $\overline{b}_H = (b_{w-1}b_{w-2}\cdots b_{k+1}b_k)_2$, and $\overline{b}_L = (b_{k-1}b_{k-2}\cdots b_0)_2$ with $0 \leq k < w$. Let $[\![\overline{b}]\!]^\tau$ be the value represented by $\overline{b}$ in the type τ.*

(1) $[\![\overline{b}]\!]^{\mathsf{uint}\ w} = 2^k \times [\![\overline{b}_H]\!]^{\mathsf{uint}\ (w-k)} + [\![\overline{b}_L]\!]^{\mathsf{uint}\ k}$; *and*
(2) $[\![\overline{b}]\!]^{\mathsf{sint}\ w} = 2^k \times [\![\overline{b}_H]\!]^{\mathsf{sint}\ (w-k)} + [\![\overline{b}_L]\!]^{\mathsf{uint}\ k}$.

Lemma 3.1 shows how to split a bit string of length w in different interpretations into bit strings of lengths $w-k$ and k. If the interpretation is unsigned, both the $w-k$ most significant bits and k least significant bits use the unsigned interpretation. On the other hand, if the interpretation is signed, then the $w-k$ most significant bits use the signed representation but the k least significant bits use the unsigned interpretation.

With Lemma 3.1, it is now easy to explain the semantics of full-multiplication instructions. For the unsigned full-multiplication instruction umull $v_H\ v_L\ a_0\ a_1$, the product of a_0 and a_1 is written as $V_H \times \overline{\rho} + V_L$ where v_H, v_L, a_0, a_1 are of the type ρ and V_L is representable in ρ. The values V_H and V_L are thus representable in ρ and stored in v_H and v_L respectively. For the signed instruction smull $v_H\ v_L\ a_0\ a_1$ with $v_H, a_0, a_1 : \sigma$, $v_L : \rho$, and $\sigma \| \rho$, the product of a_0 and a_1 is written as $V_H \times \overline{\rho} + V_L$ where V_L is representable in ρ. Thus V_H and V_L are representable in σ and ρ and stored in v_H and v_L respectively. Lemma 3.1 allows us to generalize the semantics of unsigned full-multiplication instructions in [18] naturally.

Figure 6 gives the semantics of the remaining CryptoLine instructions. Split instructions are another application of Lemma 3.1. The unsigned uspl $v_H\ v_L\ a\ n$ instruction splits the bit string for a of an unsigned type into destinations v_H and v_L of the same type. The n least significant bits are stored in v_L and the remaining bits are stored in v_H. When a is of a signed type, sspl $v_H\ v_L\ a\ n$ stores the n least significant bits of a in the unsigned destination v_L and the remaining bits in the signed destination v_H. The join instructions are almost trivial with Lemma 3.1. The unsigned ujoin $v\ a_H\ a_L$ stores the concatenated bit strings of unsigned a_H and a_L in v of the type $2 \bullet \rho$. The signed sjoin $v\ a_H\ a_L$ stores the concatenation of the signed a_H and unsigned a_L in v of the type $2 \bullet \sigma$.

The unsigned concatenation followed by left-shift instruction ucshl $v_H\ v_L\ a_H\ a_L\ n$ concatenates the bit strings of unsigned a_H with a_L, shifts the concatenated bit string to the left by n bits, and keeps the value in V. It then decomposes V into two parts V_H and V_L with V_L representable in ρ. If V_H is also representable in ρ, the bit strings representing V_H and $V_L \div 2^n$ are stored in the destinations v_H and v_L of the type ρ respectively. Note that the n least significant bits of V_L are 0 and $V_L \div 2^n$ shifts the bit string of V_L by n bits to the right. The signed variant scshl $v_H\ v_L\ a_H\ a_L\ n$ concatenates the signed bit string of a_H with the unsigned bit string of a_L, shifts the concatenated bit string to the left by n bits, and keeps the value in V. V is again decomposed into V_H and V_L with V_L representable in the type ρ of v_L. If V_H is representable in the compatible signed type σ, the bit strings of V_H and $V_L \div 2^n$ are stored in the destinations v_H and v_L of the types σ and ρ respectively. Both variants err if V_H is not representable in the type of v_H.

The cast $v@\tau\ a$ updates v with the bit string representing the value of a if the value is representable in the type τ of v. Otherwise, it yields the error environment.

A predicate ϕ *holds* in an environment ϵ (written $\epsilon \models \phi$) if ϕ evaluates to true by substituting its variables for the respective values represented by their bit strings in their types. The assert $P \twoheadrightarrow Q$ instruction checks if $P \wedge Q$ holds in the current environment. If not, the error environment is yielded. The assume $P \twoheadrightarrow Q$ instruction can only be executed when $P \wedge Q$ holds in the environment.

3.4 Specification and Verification

Let P, P' be algebraic predicates and Q, Q' range predicates. We write $\models \{P \wedge Q\} Prog \{P' \wedge Q'\}$ for the *partial correctness* of *Prog* with the *precondition* $P \wedge Q$ and *postcondition* $P' \wedge Q'$. That is, if $P \wedge Q$ holds in the environment ϵ and $\epsilon \xrightarrow{Prog} \epsilon'$ with $\epsilon' \neq \bot$, then $P' \wedge Q'$ must hold in the environment ϵ' [13]. $\models \{P \wedge Q\} Prog \{P' \wedge Q'\}$ only considers environments on termination and hence guarantees the partial correctness of *Prog*. Given algebraic predicates P, P', range predicates Q, Q', and a CryptoLine program *Prog*, the CryptoLine *verification* problem is to determine whether $\models \{P \wedge Q\} Prog \{P' \wedge Q'\}$ or not. In the remaining section, we briefly describe our algorithm for the CryptoLine verification problem and focus on our signed extension to the algorithm developed in [18].

3.4.1 Checking Safety and Range Properties. Recall that the error environment is yielded if computation results are not representable in destinations (Figure 5 and 6). Intuitively, the error environment is yielded when over- or underflow arises in computation. In order to detect over- and underflow, the CryptoLine verification algorithm performs the *safety* check to ensure the absence of error environment. All over- and underflow in computation will be identified and reported to programmers during verification. Although the idea is simple, the safety check is proved to be a rather useful tool for cryptography programming in practice.

For range properties in assertions or postconditions, the CryptoLine verification algorithm simply formulates the negation of the given range property in the bit vector theory of Satisfiability

$$\epsilon \xrightarrow{\text{uspl } v_H\ v_L\ a\ n} \epsilon[v_H, v_L \mapsto V_H, V_L] \quad (v_H, v_L, a : \rho) \quad V_H \times 2^n + V_L = [\![a]\!]^\rho_\epsilon \text{ and } \underline{\rho} < V_L < \overline{\rho}$$

$$\epsilon \xrightarrow{\text{sspl } v_H\ v_L\ a\ n} \epsilon[v_H, v_L \mapsto V_H, V_L] \quad (v_H, a : \sigma; v_L : \rho; \sigma \| \rho) \quad V_H \times 2^n + V_L = [\![a]\!]^\sigma_\epsilon \text{ and } \underline{\rho} < V_L < \overline{\rho}$$

$$\epsilon \xrightarrow{\text{ucshl } v_H\ v_L\ a_H\ a_L\ n} \epsilon' \quad (v_H, v_L, a_H, a_L : \rho) \quad V = ([\![a_H]\!]^\rho_\epsilon \times \overline{\rho} + [\![a_L]\!]^\rho_\epsilon) \times 2^n, V_H \times \overline{\rho} + V_L = V, \underline{\rho} < V_L < \overline{\rho}, \text{ and}$$
$$\epsilon' = \begin{cases} \epsilon[v_H, v_L \mapsto V_H, V_L \div 2^n] & \text{if } V_H < \overline{\rho} \\ \bot & \text{otherwise} \end{cases}$$

$$\epsilon \xrightarrow{\text{scshl } v_H\ v_L\ a_H\ a_L\ n} \epsilon' \quad (v_H, a_H : \sigma; v_L, a_L : \rho; \sigma \| \rho) \quad V = ([\![a_H]\!]^\sigma_\epsilon \times \overline{\rho} + [\![a_L]\!]^\rho_\epsilon) \times 2^n, V_H \times \overline{\rho} + V_L = V, \underline{\rho} < V_L < \overline{\rho}, \text{ and}$$
$$\epsilon' = \begin{cases} \epsilon[v_H, v_L \mapsto V_H, V_L \div 2^n] & \text{if } \underline{\sigma} < V_H < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$$

$$\epsilon \xrightarrow{\text{ujoin } v\ a_H\ a_L} \epsilon[v \mapsto V] \quad (v : 2 \bullet \rho; a_H, a_L : \rho) \quad V = [\![a_H]\!]^\rho_\epsilon \times \overline{\rho} + [\![a_L]\!]^\rho_\epsilon$$

$$\epsilon \xrightarrow{\text{sjoin } v\ a_H\ a_L} \epsilon[v \mapsto V] \quad (v : 2 \bullet \sigma; a_H : \sigma; a_L : \rho) \quad V = [\![a_H]\!]^\sigma_\epsilon \times \overline{\rho} + [\![a_L]\!]^\rho_\epsilon$$

$$\epsilon \xrightarrow{\text{cast } v@\tau\ a} \epsilon' \quad (v : \tau, a : \tau') \quad V = [\![a]\!]^{\tau'}_\epsilon \text{ and } \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\tau} < V < \overline{\tau} \\ \bot & \text{otherwise} \end{cases}$$

$$\epsilon \xrightarrow{\text{assert } P \curlywedge Q} \epsilon' \quad \epsilon' = \begin{cases} \epsilon & \text{if } \epsilon \models P \wedge Q \\ \bot & \text{otherwise} \end{cases}$$

$$\epsilon \xrightarrow{\text{assume } P \curlywedge Q} \epsilon \quad \text{if } \epsilon \models P \wedge Q$$

$$\bot \xrightarrow{inst} \bot \quad inst \in Inst$$

Figure 6: CryptoLine Semantics (continued)

Modulo Theories Library (SMT-LIB2) and employs an SMT solver to find errors. If the SMT solver finds the negated range property is satisfiable, an error is found. Otherwise, the range property cannot be falsified and thus must hold in all computation.

3.4.2 Checking Algebraic Properties. Verifying non-linear algebraic properties is notoriously hard for SMT solvers [8]. In [18], the authors formulated the verification problem as the ideal membership problem and solved it with computer algebra systems. The idea is to formulate the computation of each CryptoLine instruction by roots of multivariate polynomial equations. The computation of CryptoLine programs thus corresponds to common roots of systems of polynomial equations. Algebraic properties about roots of these equations can then be verified by computer algebra systems.

To see how to formulate computation as roots of polynomial equations, consider a CryptoLine program in static single assignment (SSA) form. Destination and source variables in an instruction are then distinct. Figure 7 characterizes each instruction by roots of multivariate polynomial equations.

For mov $v\ a$, v will be equal to a after executing the instruction. The equation $v - a = 0$ suffices to characterize the computation. The cmov $v\ c\ a_0\ a_1$ instruction will assign a_0 or a_1 to v if $c = 1$ or 0 respectively. Observe that $(c, v) = (1, a_0)$ and $(c, v) = (0, a_1)$ are roots of $v - (c \times a_0 + (1 - c) \times a_1) = 0$. The computation is characterized by the equation soundly. Both variants of addition instructions uadd $v\ a_0\ a_1$ and sadd $v\ a_0\ a_1$ are characterized by $v - (a_0 + a_1) = 0$ because their normal computation satisfies the same equation. The unsigned carrying addition uadds $c\ v\ a_0\ a_1$ sets c to 1 if $a_0 + a_1 \geq \overline{\rho}$ when v is of the unsigned type ρ; otherwise, it sets c to 0. Subsequently, $(v + c \times \overline{\rho}) - (a_0 + a_1) = 0$ and $c \times (1 - c) = 0$ suffice to characterize the computation. The signed carrying addition sadds $c\ v\ a_0\ a_1$ differs slightly. The equations $v - (a_0 + a_1) = 0$ and $c \times (1 - c) = 0$ do not characterize the computation precisely. For instance, c can be 0 or 1 regardless of the sum $a_0 + a_1$. Nonetheless, all normal computation does satisfy the equations. Other addition and subtraction instructions are characterized similarly. The computation of both variants of half-multiplication instructions satisfies the equation $v - (a_0 \times a_1) = 0$. Both ushl $v\ a\ n$ and sshl $v\ a\ n$ use the same equation $v - (a \times 2^n) = 0$. For full-multiplication, the computation of both variants is characterized by $(v_H \times \overline{\rho} + v_L) - (a_0 \times a_1) = 0$ when v_L is of the type ρ.

The same equation $(v_H \times 2^n + v_L) - a = 0$ is used for both uspl $v_H\ v_L\ a\ n$ and sspl $v_H\ v_L\ a\ n$. Similarly, ujoin $v\ a_H\ a_L$ and sjoin $v\ a_H\ a_L$ are characterized by $v - (a_H \times \overline{\rho} + a_L) = 0$ when $a_L : \rho$. Concatenation followed by left-shift instructions are slightly more complicated. Consider ucshl $v_H\ v_L\ a_H\ a_L\ n$ with $v_L, a_L : \rho$. The concatenated result $(a_H \times \overline{\rho} + a_L)$ is shifted to the left by multiplying 2^n. The intermediate result $((a_H \times \overline{\rho} + a_L) \times 2^n)$ is splitted into two parts: v_H and $v_L \times 2^n$. Hence the equation $(v_H \times \overline{\rho} + v_L \times 2^n) - (a_H \times \overline{\rho} + a_L) \times 2^n = 0$ is used. The signed scshl $v_H\ v_L\ a_H\ a_L\ n$ instruction has the same equation. The cast $v@\tau\ a$ simply uses the equation $v - a = 0$ since v is assigned to a in successful computation.

Lemma 3.2. *Consider $inst \hookrightarrow eqns$ in Figure 7 where inst is a CryptoLine instruction and eqns are multivariate polynomial equations. If $\epsilon \xrightarrow{inst} \epsilon'$ and $\epsilon' \neq \bot$, then the values of source and destination variables of inst in the environment ϵ' are a root of eqns.*

Theorem 3.3. *Let Prog be a CryptoLine program in static single assignment form and Eqns a system of multivariate polynomial equations obtained by converting each instruction in Prog using Figure 7. If $\epsilon \xrightarrow{Prog} \epsilon'$ and $\epsilon' \neq \bot$, then the values of all variables of Prog in the environment ϵ' are a common root of Eqns.*

Let $\overline{\mathbf{v}}$ be a set of variables. We write $\mathbb{Z}[\overline{\mathbf{v}}]$ for the set of multivariate polynomials in $\overline{\mathbf{v}}$ with integer coefficients. An *ideal* I in $\mathbb{Z}[\overline{\mathbf{v}}]$ is a set of polynomials in $\mathbb{Z}[\overline{\mathbf{v}}]$ such that (1) $f + g \in I$ if $f, g \in I$; and (2) $hf \in I$ if $h \in \mathbb{Z}[\overline{\mathbf{v}}]$ and $f \in I$. Let $f_1, f_2, \ldots, f_n \in \mathbb{Z}[\overline{\mathbf{v}}]$. We write $\langle f_1, f_2, \ldots, f_n \rangle$ for the ideal generated by $f_1, f_2, \ldots, f_n$.

$$
\begin{array}{lcl}
\mathsf{mov}\ v\ a & \hookrightarrow & v - a = 0 \\
\mathsf{uadd}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 + a_1) = 0 \\
\mathsf{uadds}\ c\ v\ a_0\ a_1 & \hookrightarrow & \left[\begin{array}{l} (v + c \times \overline{\rho}) - (a_0 + a_1) = 0 \\ c \times (1 - c) = 0 \qquad (v : \rho) \end{array}\right. \\
\mathsf{uadc}\ v\ a_0\ a_1 d & \hookrightarrow & v - (a_0 + a_1 + d) = 0 \\
\mathsf{uadcs}\ c\ v\ a_0\ a_1 d & \hookrightarrow & \left[\begin{array}{l} (v + c \times \overline{\rho}) - (a_0 + a_1 + d) = 0 \\ c \times (1 - c) = 0 \qquad (v : \rho) \end{array}\right. \\
\mathsf{usub}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 - a_1) = 0 \\
\mathsf{usubs}\ c\ v\ a_0\ a_1 & \hookrightarrow & \left[\begin{array}{l} (v - c \times \overline{\rho}) - (a_0 - a_1) = 0 \\ c \times (1 - c) = 0 \qquad (v : \rho) \end{array}\right. \\
\mathsf{usbb}\ v\ a_0\ a_1\ d & \hookrightarrow & v - (a_0 - a_1 - d) = 0 \\
\mathsf{usbbs}\ c\ v\ a_0\ a_1\ d & \hookrightarrow & \left[\begin{array}{l} (v - c \times \overline{\rho}) - (a_0 - a_1 - d) = 0 \\ c \times (1 - c) = 0 \qquad (v : \rho) \end{array}\right. \\
\mathsf{umul}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 \times a_1) = 0 \\
\mathsf{umull}\ v_H\ v_L\ a_0\ a_1 & \hookrightarrow & (v_H \times \overline{\rho} + v_L) - (a_0 \times a_1) = 0\ (v_L : \rho) \\
\mathsf{ushl}\ v\ a\ n & \hookrightarrow & v - a \times 2^n = 0 \\
\mathsf{sshl}\ v\ a\ n & \hookrightarrow & v - a \times 2^n = 0 \\
\mathsf{uspl}\ v_H\ v_L\ a\ n & \hookrightarrow & (v_H \times 2^n + v_L) - a = 0 \\
\mathsf{ujoin}\ v\ a_H\ a_L & \hookrightarrow & v - (a_H \times \overline{\rho} + a_L) = 0 \qquad (a_L : \rho) \\
\mathsf{cast}\ v@\tau\ a & \hookrightarrow & v - a = 0
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{cmov}\ v\ c\ a_0\ a_1 & \hookrightarrow & v - (c \times a_0 + (1 - c) \times a_1) = 0 \\
\mathsf{sadd}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 + a_1) = 0 \\
\mathsf{sadds}\ c\ v\ a_0\ a_1 & \hookrightarrow & \left[\begin{array}{l} v - (a_0 + a_1) = 0 \\ c \times (1 - c) = 0 \end{array}\right. \\
\mathsf{sadc}\ v\ a_0\ a_1 d & \hookrightarrow & v - (a_0 + a_1 + d) = 0 \\
\mathsf{sadcs}\ c\ v\ a_0\ a_1 d & \hookrightarrow & \left[\begin{array}{l} v - (a_0 + a_1 + d) = 0 \\ c \times (1 - c) = 0 \end{array}\right. \\
\mathsf{ssub}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 - a_1) = 0 \\
\mathsf{ssubs}\ c\ v\ a_0\ a_1 & \hookrightarrow & \left[\begin{array}{l} v - (a_0 - a_1) = 0 \\ c \times (1 - c) = 0 \end{array}\right. \\
\mathsf{ssbb}\ v\ a_0\ a_1\ d & \hookrightarrow & v - (a_0 - a_1 - d) = 0 \\
\mathsf{ssbbs}\ c\ v\ a_0\ a_1\ d & \hookrightarrow & \left[\begin{array}{l} v - (a_0 - a_1 - d) = 0 \\ c \times (1 - c) = 0 \end{array}\right. \\
\mathsf{smul}\ v\ a_0\ a_1 & \hookrightarrow & v - (a_0 \times a_1) = 0 \\
\mathsf{smull}\ v_H\ v_L\ a_0\ a_1 & \hookrightarrow & (v_H \times \overline{\rho} + v_L) - (a_0 \times a_1) = 0\ (v_L : \rho) \\
\mathsf{ucshl}\ v_H\ v_L\ a_H\ a_L\ n & \hookrightarrow & (v_H \times \overline{\rho} + v_L \times 2^n) - (a_H \times \overline{\rho} + a_L) \times 2^n = 0\ \ (v_L, a_L : \rho) \\
\mathsf{scshl}\ v_H\ v_L\ a_H\ a_L\ n & \hookrightarrow & (v_H \times \overline{\rho} + v_L \times 2^n) - (a_H \times \overline{\rho} + a_L) \times 2^n = 0\ \ (v_L, a_L : \rho) \\
\mathsf{sspl}\ v_H\ v_L\ a\ n & \hookrightarrow & (v_H \times 2^n + v_L) - a = 0 \\
\mathsf{sjoin}\ v\ a_H\ a_L & \hookrightarrow & v - (a_H \times \overline{\rho} + a_L) = 0 \qquad (a_L : \rho)
\end{array}
$$

Figure 7: Polynomial Equations

Given $h, f_1, f_2, \ldots, f_n \in \mathbb{Z}[\overline{\mathbf{v}}]$, the *ideal membership problem* is to determine whether $h \in \langle f_1, f_2, \ldots, f_n \rangle$.

Theorem 3.4. *Let $h, f_1, f_2, \ldots, f_n \in \mathbb{Z}[\overline{\mathbf{v}}]$. If $h \in \langle f_1, f_2, \ldots, f_n \rangle$, then $\forall \overline{\mathbf{v}}. f_1 = 0 \wedge f_2 = 0 \wedge \cdots \wedge f_n = 0 \implies h = 0$.*

Intuitively, Theorem 3.4 says that if h is in the ideal generated by $f_1, f_2, \ldots, f_n$, then all common roots of $f_1 = 0, f_2 = 0, \ldots, f_n = 0$ are also roots of $h = 0$. Let us consider the system of polynomial equations *Eqns* corresponding to the CryptoLine program *Prog* (Theorem 3.3). An algebraic equality on CryptoLine program variables is but a polynomial equation over these variables. To verify a given algebraic equality is hence to check if all common roots of *Eqns* are also roots of the algebraic equality. By Theorem 3.4, it suffices to solve the corresponding ideal membership problem. Algebraic modulo equalities can be reduced to the ideal membership problem as well [18]. Through Theorem 3.3, algebraic properties in our signed extension to CryptoLine can be verified algebraically.

4 GIMPLECRYPTOLINE

An important application of CryptoLine with signed computation is to verify cryptographic C programs. The 32-bit C implementation of Curve25519 field operations in wolfSSL uses a signed representation. A 255-bit field element is represented by 5 26-bit and 5 25-bit signed limbs. In order to verify such programs, we develop a translator from the intermediate representation GIMPLE in GNU Compiler Collection (GCC) to CryptoLine. Not all GIMPLE statements are needed for cryptographic C programs however. We identify a subset (called gimpleCryptoLine) after inspecting GIMPLE code for such programs. We describe gimpleCryptoLine and its translation here.

4.1 Syntax and Semantics

Figure 8 shows the syntax of gimpleCryptoLine. An operand is either a variable or a constant. A $Vec(\ell)$ (denoted by $v_1(\ell), v_2(\ell), \ldots$) is a vector variable or a vector of ℓ constants. In gimpleCryptoLine, each operand has a type with its bit width information.

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \texttt{int32\_t} \mid \texttt{uint32\_t} \mid \texttt{int64\_t} \mid \texttt{uint64\_t} \mid \cdots \\
\mathit{Const} & ::= & 1 \mid 2 \mid 3 \cdots \qquad \mathit{Var} ::= x \mid y \mid z \cdots \\
\mathit{Vec}(\ell) & ::= & \mathit{Var} \mid \mathit{Const}^{\ell} \qquad \mathit{Op} ::= \mathit{Var} \mid \mathit{Const} \\
\mathit{Stmt} & ::= & \mathit{Var} = \mathit{Op} + \mathit{Op} \mid \mathit{Var} = \mathit{Op} - \mathit{Op} \\
 & \mid & \mathit{Var} = \mathit{Op} * \mathit{Op} \mid \mathit{Var} = \mathit{Op}\ \texttt{w*}\ \mathit{Op} \\
 & \mid & \mathit{Var} = \mathit{Op} >> \mathit{Const} \mid \mathit{Var} = \mathit{Op} << \mathit{Const} \\
 & \mid & \mathit{Var} = (\mathit{Type})\mathit{Var} \mid \mathit{Var} = \mathit{Vec}(\ell) +_v \mathit{Vec}(\ell) \\
 & \mid & \mathit{Var} = \mathit{Vec}(\ell) -_v \mathit{Vec}(\ell) \\
\mathit{Decls} & ::= & \mathit{Type}\ \mathit{Var}; \mid \mathit{Type}\ \mathit{Var}; \mathit{Decls} \\
\mathit{Stmts} & ::= & \mathit{Stmt}; \mid \mathit{Stmt}; \mathit{Stmts} \\
\mathit{Prog} & ::= & \mathit{Decls}\ \mathit{Stmts}
\end{array}
$$

Figure 8: gimpleCryptoLine Syntax

Let *op* be an operand. We write t_{op} and w_{op} for the type and bit width of *op*. $w_{v(\ell)}$ on the other hand denotes the bit width of an *element* in the vector $v(\ell)$. The addition statement x = op_1 + op_2 computes the sum of op_1 and op_2 and assigns it to x. The subtraction statement x = op_1 - op_2 assigns the difference of op_1 and op_2 to x. In addition and subtraction, t_x, t_{op_1}, and t_{op_2} must be the same.

The multiplication statement x = op_1 * op_2 stores the product of op_1 and op_2 in x. It requires x, op_1, and op_2 to have the same type. The wide multiplication statement x = op_1 w* op_2 is similar. The types of x, op_1, and op_2 must be all signed or unsigned with $w_x = 2w_{op_1} = 2w_{op_2}$. The arithmetic shift statements x = op_1 << n and x = op_1 >> n shift op_1 to the left or right by n bits respectively and assign to x the result. x and op_1 must be of the same type. Since GIMPLE statements are typed, type casting is essential. The statement x = $(T)y$ casts y to the type T and assigns the result to x.

In cryptography library binary codes, we find Single Instruction Multiple Data (SIMD) assembly instructions are generated from

Table 1: Translation

gimpleCryptoLine	CryptoLine
$x = op_1 + op_2$	adds $dc\ x\ op_1\ op_2$
$x = op_1 - op_2$	subs $dc\ x\ op_1\ op_2$
$x = op_1 * op_2$	mull $dc\ x\ op_1\ op_2$
$x = op_1$ w* op_2	mull $x_H\ x_L\ op_1\ op_2$ join $x\ x_H\ x_L$
$x = op_1 >> n$	spl $x\ dc\ op_1\ n$
$x = op_1 << n$	spl $dc\ t\ op_1\ (w_x - n)$ shl $x\ t\ n$
$v_1(\ell) = v_2(\ell) +_v v_3(\ell)$	sequence of adds
$v_1(\ell) = v_2(\ell) -_v v_3(\ell)$	sequence of subs

sequential C source codes through compiler optimization. GCC supports SIMD instructions via vector statements. The vector addition statement $v_1(\ell) = v_2(\ell) +_v v_3(\ell)$ assigns to $v_1(\ell)$ the elementwise sum of $v_2(\ell)$ and $v_3(\ell)$. Similarly, the vector subtraction statement $v_1(\ell) = v_2(\ell) -_v v_3(\ell)$ stores the elementwise difference of $v_2(\ell)$ and $v_3(\ell)$ in $v_1(\ell)$. In vector statements, $v_1(\ell)$, $v_2(\ell)$, and $v_3(\ell)$ must have the same element type and length.

Figure 9 gives the operational semantics of gimpleCryptoLine. In the figure, a *state* $\theta : Var \to \mathbb{Z}$ is a mapping from variables to values. Each rule specifies the effects of a statement on a state. Generally, each statement performs its computation by interpreting all operands in unsigned types. Unsigned intermediate results are truncated to the bit width of the destination variable. Finally, truncated results are converted to correct types and stored in states.

The addition statement $x = op_1 + op_2$ computes the unsigned sum of op_1 and op_2, truncates to the bit width of x, converts to the type of x, and assigns the result to x. Subsequently, x will not be the sum of op_1 and op_2 if over- or underflow occurs. Other statements are similar except the wide multiplication and arithmetic right shift statements. In wide multiplication, the product of operands is always computed accurately. The semantics of the arithmetic right shift statement uses the floor function to discard fractional parts. Let $v(\ell)$ be a vector. $v(\ell)[i]$ denotes the i-th element of $v(\ell)$ for $0 \le i < \ell$. For any integer function f, $\theta[v(\ell)[i] \leftarrow f(i)]_{i=0}^{\ell-1}$ is short for $\theta[v_1(\ell)[0] \leftarrow f(0), \ldots, v_1(\ell)[\ell-1] \leftarrow f(\ell-1)]$. The vector addition statement $v_1(\ell) = v_2(\ell) +_v v_3(\ell)$ computes the unsigned sum of $v_2(\ell)[i]$ and $v_3(\ell)[i]$, truncates to the bit width of elements, converts the truncated sum to the type of elements, and stores the result in $v_1(\ell)[i]$. The vector subtraction statement is similar.

The type conversion $x = (T)y$ compares the bit widths of x and y. If $w_x < w_y$, we convert the w_x least significant bits of y to the type of x and store the converted value in x. Otherwise, we simply convert the value of y to the type of x and update the value of x.

4.2 From gimpleCryptoLine to CryptoLine

The translation from gimpleCryptoLine to CryptoLine is summarized in Table 1. Addition and subtraction statements are translated to corresponding CryptoLine instructions. The gimpleCryptoLine multiplication statement is translated to the CryptoLine full-multiplication by discarding the more significant half. On the other hand, the wide multiplication statement is translated to a full-multiplication followed by a join instruction. The gimpleCryptoLine right shift statement is translated to the spl instruction by discarding the least significant bits. For the left shift statement, only the $w_x - n$ least significant bits are shifted to the left by n bits to prevent errors in CryptoLine. Observe that the translation rules do not specify variants of CryptoLine instructions. The CryptoLine type system will infer the intended variant automatically.

4.3 Example

Consider the following C function `fe_sub` for Curve25519 field operations in wolfSSL (some comments and whitespace are removed):

```
/* h = f - g
Preconditions:
|f| bounded by 1.1*2^25,1.1*2^24,1.1*2^25,1.1*2^24,etc.
|g| bounded by 1.1*2^25,1.1*2^24,1.1*2^25,1.1*2^24,etc.
Postconditions:
|h| bounded by 1.1*2^26,1.1*2^25,1.1*2^26,1.1*2^25,etc.
 */
typedef int32_t fe[10];
void fe_sub(fe h, const fe f, const fe g) {
  int32_t f0=f[0]; int32_t f1=f[1]; int32_t f2=f[2];
  int32_t f3=f[3]; int32_t f4=f[4]; int32_t f5=f[5];
  int32_t f6=f[6]; int32_t f7=f[7]; int32_t f8=f[8];
  int32_t f9=f[9];
  int32_t g0=g[0]; int32_t g1=g[1]; int32_t g2=g[2];
  int32_t g3=g[3]; int32_t g4=g[4]; int32_t g5=g[5];
  int32_t g6=g[6]; int32_t g7=g[7]; int32_t g8=g[8];
  int32_t g9=g[9];
  int32_t h0=f0-g0; int32_t h1=f1-g1; int32_t h2=f2-g2;
  int32_t h3=f3-g3; int32_t h4=f4-g4; int32_t h5=f5-g5;
  int32_t h6=f6-g6; int32_t h7=f7-g7; int32_t h8=f8-g8;
  int32_t h9=f9-g9;
  h[0]=h0; h[1]=h1; h[2]=h2; h[3]=h3; h[4]=h4; h[5]=h5;
  h[6]=h6; h[7]=h7; h[8]=h8; h[9]=h9;
}
```

In wolfSSL, a field element is stored as an array of 10 32-bit signed integers. The `fe_sub` function computes the difference of two field elements `f` and `g` and stores it in the field element `h`. GCC transforms the C function to the following gimple program:

```
f0_3 = *f_2(D);
f1_4 = MEM[(const int32_t *)f_2(D) + 4B];
f2_5 = MEM[(const int32_t *)f_2(D) + 8B];
    ...
f9_12 = MEM[(const int32_t *)f_2(D) + 36B];
g0_14 = *g_13(D);
g1_15 = MEM[(const int32_t *)g_13(D) + 4B];
g2_16 = MEM[(const int32_t *)g_13(D) + 8B];
    ...
g9_23 = MEM[(const int32_t *)g_13(D) + 36B];
h0_24 = f0_3 - g0_14;   h1_25 = f1_4 - g1_15;
h2_26 = f2_5 - g2_16;   h3_27 = f3_6 - g3_17;
h4_28 = f4_7 - g4_18;   h5_29 = f5_8 - g5_19;
h6_30 = f6_9 - g6_20;   h7_31 = f7_10 - g7_21;
h8_32 = f8_11 - g8_22;  h9_33 = f9_12 - g9_23;
*h_34(D) = h0_24;
MEM[(int32_t *)h_34(D) + 4B] = h1_25;
MEM[(int32_t *)h_34(D) + 8B] = h2_26;
    ...
MEM[(int32_t *)h_34(D) + 36B] = h9_33;
```

In the program, the 32-bit signed variables `f0_3`, `f1_4`, ..., `f9_12` represent the field element `f` in the C code; `g0_14`, `g1_15`, ..., `g9_23` represent the field element `g`. The difference of `f` and `g` is stored in `h0_24`, `h1_25`, ..., `h9_33`. Using our tool, the gimple program is translated to the following CryptoLine program automatically:

$$[\![c]\!]_\theta = c \quad \text{for } c \in \mathit{Const} \qquad [\![v]\!]_\theta = \theta(v) \quad \text{for } v \in \mathit{Var}$$

$$u(z,w) = \begin{cases} z + 2^w & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} \quad \text{for } z \in \mathbb{Z}, w \in \mathbb{N} \qquad s(z,w) = \begin{cases} z - 2^w & \text{if } z > 2^{w-1} - 1 \\ z & \text{otherwise} \end{cases} \quad \text{for } z \in \mathbb{Z}, w \in \mathbb{N}$$

$$[\![v]\!]^u_\theta = u([\![v]\!]_\theta, w_v) \quad \text{for } v \in \mathit{Var} \qquad [\![v]\!]^s_\theta = s([\![v]\!]_\theta, w_v) \quad \text{for } v \in \mathit{Var}$$

$$\mathit{cvt}(v,z) = \begin{cases} u(z,w_v) & \text{if } t_v \text{ is unsigned} \\ s(z,w_v) & \text{if } t_v \text{ is signed} \end{cases} \quad \text{for } v \in \mathit{Var}$$

$$\theta \xrightarrow{x = op_1 + op_2} \theta[x \leftarrow \mathit{cvt}(x, ([\![op_1]\!]^u_\theta + [\![op_2]\!]^u_\theta) \bmod 2^{w_x})] \qquad \theta \xrightarrow{x = op_1 - op_2} \theta[x \leftarrow \mathit{cvt}(x, ([\![op_1]\!]^u_\theta - [\![op_2]\!]^u_\theta) \bmod 2^{w_x})]$$

$$\theta \xrightarrow{x = op_1 * op_2} \theta[x \leftarrow \mathit{cvt}(x, ([\![op_1]\!]^u_\theta \times [\![op_2]\!]^u_\theta) \bmod 2^{w_x})] \qquad \theta \xrightarrow{x = op_1 \,\mathtt{w*}\, op_2} \theta[x \leftarrow [\![op_1]\!]_\theta \times [\![op_2]\!]_\theta]$$

$$\theta \xrightarrow{x = op_1 >> n} \theta[x \leftarrow \lfloor [\![op_1]\!]_\theta \div 2^{[\![n]\!]_\theta} \rfloor] \qquad \theta \xrightarrow{x = op_1 << n} \theta[x \leftarrow \mathit{cvt}(x, ([\![op_1]\!]^u_\theta \times 2^{[\![n]\!]_\theta}) \bmod 2^{w_x})]$$

$$\theta \xrightarrow{v_1(\ell) = v_2(\ell) +_v v_3(\ell)} \theta[v_1(\ell)[i] \leftarrow \mathit{cvt}(v_1(\ell)[i], ([\![v_2(\ell)[i]]\!]^u_\theta + [\![v_3(\ell)[i]]\!]^u_\theta) \bmod 2^{w_{v_1(\ell)}})]_{i=0}^{\ell-1}$$

$$\theta \xrightarrow{v_1(\ell) = v_2(\ell) -_v v_3(\ell)} \theta[v_1(\ell)[i] \leftarrow \mathit{cvt}(v_1(\ell)[i], ([\![v_2(\ell)[i]]\!]^u_\theta - [\![v_3(\ell)[i]]\!]^u_\theta) \bmod 2^{w_{v_1(\ell)}})]_{i=0}^{\ell-1}$$

$$\theta \xrightarrow{x = (T)y} \theta[x \leftarrow Y] \text{ where } Y = \begin{cases} \mathit{cvt}(x, [\![y]\!]^u_\theta \bmod 2^{w_x}) & \text{if } w_x < w_y \\ \mathit{cvt}(x, [\![y]\!]_\theta) & \text{otherwise} \end{cases}$$

Figure 9: The semantics of GIMPLECRYPTOLINE

```
1  mov f03 f2_0          21 sub h024 f03 g014
2  mov f14 f2_4          22 sub h125 f14 g115
3  mov f25 f2_8          23 sub h226 f25 g216
4  mov f36 f2_12         24 sub h327 f36 g317
5  mov f47 f2_16         25 sub h428 f47 g418
6  mov f58 f2_20         26 sub h529 f58 g519
7  mov f69 f2_24         27 sub h630 f69 g620
8  mov f710 f2_28        28 sub h731 f710 g721
9  mov f811 f2_32        29 sub h832 f811 g822
10 mov f912 f2_36        30 sub h933 f912 g923
11 mov g014 g13_0        31 mov h34_0 h024
12 mov g115 g13_4        32 mov h34_4 h125
13 mov g216 g13_8        33 mov h34_8 h226
14 mov g317 g13_12       34 mov h34_12 h327
15 mov g418 g13_16       35 mov h34_16 h428
16 mov g519 g13_20       36 mov h34_20 h529
17 mov g620 g13_24       37 mov h34_24 h630
18 mov g721 g13_28       38 mov h34_28 h731
19 mov g822 g13_32       39 mov h34_32 h832
20 mov g923 g13_36       40 mov h34_36 h933
```

Let fe be an array representing a field element. From programmers' comments in the fe_sub function (not shown), it is seen that |fe[0]|<1.1*2^25, |fe[1]|<1.1*2^24, |fe[2]|<1.1*2^25, |fe[3]|<1.1*2^24, and so on are required for each field element. Now $1.1 \times 2^{25} = 36909875.2$ and $1.1 \times 2^{24} = 18454937.6$. Define

$$\begin{aligned} &R(\mathtt{fe0}, \mathtt{fe1}, \ldots, \mathtt{fe9}) = \\ &\quad (-36909876@\text{sint } 32 < \mathtt{fe0} \wedge \mathtt{fe0} < 36909876@\text{sint } 32 \\ &\wedge \quad -18454938@\text{sint } 32 < \mathtt{fe1} \wedge \mathtt{fe1} < 18454938@\text{sint } 32 \\ &\quad \ldots \\ &\wedge \quad -18454938@\text{sint } 32 < \mathtt{fe9} < 18454938@\text{sint } 32). \end{aligned}$$

The precondition for the C function is therefore

$$\text{true} \curlywedge R(\mathtt{f2\_0}, \mathtt{f2\_4}, \ldots, \mathtt{f2\_36}) \wedge R(\mathtt{g13\_0}, \mathtt{g13\_4}, \ldots, \mathtt{g13\_36}).$$

To specify the postcondition, it is necessary to understand how a 255-bit field element is represented by an array of signed integers. Let fe be an array representing a field element. $\mathit{FE}(\mathtt{fe[0]}, \mathtt{fe[1]}, \ldots, \mathtt{fe[9]})$ gives the field element represented by fe where

$$\begin{aligned} \mathit{FE}(\mathtt{fe0}, \mathtt{fe1}, \ldots, \mathtt{fe9}) = \mathtt{fe0} &\times 2^0 + \mathtt{fe1} \times 2^{26} + \\ \mathtt{fe2} &\times 2^{51} + \mathtt{fe3} \times 2^{77} + \mathtt{fe4} \times 2^{102} + \mathtt{fe5} \times 2^{128} + \\ \mathtt{fe6} &\times 2^{153} + \mathtt{fe7} \times 2^{179} + \mathtt{fe8} \times 2^{204} + \mathtt{fe9} \times 2^{230}. \end{aligned}$$

That is, a field element is represented by 5 26-bit limbs (fe[0], fe[2], ..., fe[8]) and 5 25-bit limbs (fe[1], fe[3], ..., fe[9]). From programmers' comments, define the output limb ranges by

$$\begin{aligned} &R'(\mathtt{fe0}, \mathtt{fe1}, \ldots, \mathtt{fe9}) = \\ &\quad (-73819751@\text{sint } 32 < \mathtt{fe0} \wedge \mathtt{fe0} < 73819751@\text{sint } 32 \\ &\wedge \quad -36909876@\text{sint } 32 < \mathtt{fe1} \wedge \mathtt{fe1} < 36909876@\text{sint } 32 \\ &\quad \ldots \\ &\wedge \quad -36909876@\text{sint } 32 < \mathtt{fe9} < 36909876@\text{sint } 32). \end{aligned}$$

The postcondition for fe_sub is therefore

$$\begin{aligned} &\mathit{FE}(\mathtt{h34\_0}, \ldots, \mathtt{h34\_36}) \equiv \\ &\quad \mathit{FE}(\mathtt{f2\_0}, \ldots, \mathtt{f2\_36}) - \mathit{FE}(\mathtt{g13\_0}, \ldots, \mathtt{g13\_36}) \bmod 2^{255} - 19 \\ &\curlywedge R'(\mathtt{h34\_0}, \ldots, \mathtt{h34\_36}). \end{aligned}$$

Using a laptop, our verification tool verifies the fe_sub function in 2 seconds. The C function computes the difference of two field elements correctly. There cannot be any over- or underflow if limbs of input field elements are in ranges specified in the comments. Each limb of the output field elements is always in ranges too.

Our translator handles memory and pointer arithmetic automatically. Several heuristics are implemented for better usability (such as translating $x = op_1 - op_2$ to sub x op_1 op_2). Our verification technique supports bitwise logical operations, comparators, and branches as well. See [14] for a corresponding translation from LLVM intermediate representation to CRYPTOLINE.

5 EVALUATION

We implement our signed extension to CRYPTOLINE and the GIMPLECRYPTOLINE translator. The GIMPLECRYPTOLINE is implemented as a GCC plugin. Our plugin translates the GIMPLE representation of any indicated C function after the machine-independent optimization pass. If an unrecognized GIMPLE statement is encountered, the plugin simply copies the GIMPLE statement for manual translation.

Table 2: Experimental Result

Function	L_{IR}	L_{CL}	D	P	TR_{M1}	MR_{M1}	TA_{M1}	MA_{M1}	TR_{M2}	MR_{M2}	TA_{M2}	MA_{M2}
nacl/curve25519/donna_c64/curve25519.c (MathSAT, SMT-LIB2 format)												
fdifference_backwards	69	69	66	0	-	-	0.23	6.3	-	-	0.14	9.1
fmul	91	127	10	14	12.51	452.2	0.20	6.3	4.05	486.6	0.14	9.3
fscalar_product	38	38	7	10	2.75	104.4	0.20	5.6	0.95	108.4	0.12	8.6
fsquare	68	116	10	12	7.44	288.1	0.22	6.3	2.61	301.0	0.13	9.3
fsum	20	20	0	0	0.48	5.6	0.15	4.8	0.22	10.0	0.10	8.2
fmonty	1147	1493	361	127	-	-	OOM	OOM	-	-	353.66	32764
wolfssl/fe_operations.c (Boolector with Lingeling, BTOR format)												
fe_add	40	40	0	0	1.48	6.5	0.19	5.6	0.61	9.5	0.11	8.6
fe_mul	305	305	20	24	OOT	OOT	0.32	7.0	13178	883.3	0.15	9.9
fe_mul121666	91	91	20	20	19.68	17.9	0.26	6.4	3.75	13.8	0.13	9.4
fe_neg	30	30	0	0	1.24	6.5	0.18	5.3	0.63	9.3	0.10	8.3
fe_sq	204	204	20	24	13411.84	351.9	0.33	6.7	2033	355.6	0.14	9.6
fe_sq2	214	214	20	24	18252.02	388.9	0.30	6.8	2763	385.5	0.14	9.6
fe_sub	40	40	0	0	1.31	6.5	0.16	5.7	0.64	9.4	0.11	8.6
curve25519[1]	2770	2770	200	236	OOT	OOT	12.06	385.6	68140	796.7	8.26	382.1
bitcoin/field_5x52_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_fe_add	13	20	0	0	0.33	5.3	0.14	4.8	0.22	10.0	0.09	8.3
secp256k1_fe_cmov	29	49	13	20	1.35	28.7	0.29	6.4	0.46	29.6	0.17	9.3
secp256k1_fe_from_storage	24	32	6	14	0.53	6.4	0.15	5.2	0.31	10.7	0.09	8.4
secp256k1_fe_mul_int	16	16	2	0	0.52	26.1	0.14	4.7	0.28	28.0	0.10	8.4
secp256k1_fe_negate	20	20	2	0	0.52	5.7	0.18	4.9	0.27	9.9	0.11	8.6
bitcoin/field_5x52_impl.h (Boolector with Lingeling, BTOR format)												
secp256k1_fe_normalize	52	60	21	0	117.18	45.3	0.12	5.3	91.89	31.5	0.08	8.3
secp256k1_fe_normalize_var	63	63	29	0	120.80	47.1	0.12	5.4	95.65	34.1	0.08	8.3
secp256k1_fe_normalize_weak	26	26	15	0	63.85	40.0	0.25	5.3	51.51	28.3	0.13	8.8
secp256k1_fe_normalizes_to_zero	34	39	10	0	203.12	60.3	0.16	5.2	151.03	42.9	0.08	8.2
bitcoin/field_5x52_int128_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_fe_mul_inner	111	137	17	24	16.09	461.0	0.22	6.5	4.00	489.1	0.14	9.5
secp256k1_fe_sqr_inner	90	116	21	22	9.91	284.5	0.20	6.4	2.72	303.2	0.14	9.3
bitcoin/scalar_4x64_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_scalar_add	81	102	55	22	2.03	10.1	0.21	6.5	1.11	14.1	0.13	9.4
secp256k1_scalar_eq	17	17	23	0	0.29	9.2	0.10	4.7	0.26	14.5	0.07	7.6
secp256k1_scalar_mul_512	273	384	136	90	13.75	263.3	0.26	7.1	4.96	280.0	0.16	9.9
secp256k1_scalar_mul	652	947	379	228	128.19	453.9	0.84	19.8	741.35	2219	0.43	16.3
secp256k1_scalar_negate	41	55	4	1	28.50	132.4	0.10	5.0	40.31	135.5	0.08	8.0
secp256k1_scalar_reduce_512	379	563	243	138	31.84	127.5	0.37	8.7	8.25	128.2	0.23	11.7
secp256k1_scalar_reduce	34	32	11	8	1.52	11.7	0.18	6.4	0.88	15.2	0.14	9.3
secp256k1_scalar_sqr_512	235	333	145	88	23.75	212.9	0.26	7.2	7.39	204.8	0.17	10.1
secp256k1_scalar_sqr	614	896	388	226	234.87	349.1	0.82	19.8	26.69	341.5	0.45	16.5
bitcoin/group_impl.h (MathSAT, SMT-LIB2 format)												
secp256k1_ge_from_storage	48	65	12	28	0.93	6.5	0.19	6.3	0.48	10.7	0.12	9.2
secp256k1_ge_neg	33	31	0	10	0.76	6.6	0.19	5.4	0.44	11.2	0.13	8.7
secp256k1_gej_add_ge_var	2109	2457	371	396	574.39	3166.9	OOM	OOM	75	3354	9363	70156
secp256k1_gej_double_var	899	1042	154	160	163.30	1703.0	0.77	18.4	25.27	1806	0.57	22.7
openssl/curve25519.c (MathSAT, SMT-LIB2 format)												
fe51_add	20	20	0	0	0.85	6.0	0.19	4.9	0.36	10.0	0.10	8.3
fe51_mul	96	105	11	20	17.95	381.2	0.26	6.4	3.69	409.3	0.13	9.2
fe51_mul121666	44	44	11	14	1.3	17.3	0.25	5.8	0.63	20.2	0.12	8.7
fe51_sq	73	82	11	20	8.07	227.0	0.23	6.3	2.22	247.6	0.14	9.2

Table 2 – continued on next page

Table 2 – continued from previous page

Function	L_{IR}	L_{CL}	D	P	TR_{M1}	MR_{M1}	TA_{M1}	MA_{M1}	TR_{M2}	MR_{M2}	TA_{M2}	MA_{M2}
fe51_sub	25	25	10	10	0.37	6.8	0.24	5.4	0.26	11.4	0.13	8.9
x25519_scalar_mult[1]	923	1047	110	194	558.56	1419.8	187.40	5538	119.89	1472	145.12	5511
openssl/ecp_nistp224.c (MathSAT, SMT-LIB2 format)												
felem_diff_128_64	24	36	0	0	0.56	6.4	0.23	5.1	0.32	10.7	0.14	8.6
felem_diff	24	24	0	0	0.55	5.8	0.19	4.9	0.33	10.4	0.11	8.8
felem_mul	40	40	0	0	2.24	83.2	0.15	5.2	0.65	88	0.09	8.2
felem_mul_reduce	82	121	15	16	10.65	321.8	0.20	6.4	3.11	322.5	0.13	9.1
felem_neg	47	58	5	10	0.95	6.8	0.19	5.8	0.55	11.1	0.12	8.7
felem_reduce	56	95	6	18	1.67	13.7	0.20	6.3	0.88	17.3	0.13	9.3
felem_scalar	12	12	0	0	0.48	26.7	0.14	4.6	0.24	28.9	0.09	8.1
felem_square	27	27	0	0	1.11	45.1	0.15	4.9	0.43	47.6	0.10	8.2
felem_square_reduce	69	108	14	18	6.36	195.8	0.21	6.4	1.81	198.8	0.13	9.2
felem_sum	16	16	0	0	0.41	5.4	0.15	4.7	0.26	10.0	0.10	8.3
widefelem_diff	41	63	0	0	0.90	6.5	0.19	5.7	0.46	10.6	0.12	8.7
widfefelem_scalar	21	21	0	0	2.58	87.7	0.14	4.8	0.70	88.3	0.10	8.4
openssl/ecp_nistp256.c (MathSAT, SMT-LIB2 format)												
felem_diff	24	36	0	0	0.59	7.6	0.18	5.1	0.35	11.7	0.12	8.6
felem_scalar	13	13	0	0	0.70	47.7	0.17	4.6	0.31	48.8	0.10	8.2
felem_shrink	65	95	18	16	1.78	14.0	0.20	6.4	0.95	17.1	0.13	9.3
felem_small_mul	145	95	17	46	4.75	123.0	0.23	7.0	2.29	123.2	0.14	9.8
felem_small_sum	20	20	0	0	0.41	5.8	0.14	4.8	0.25	10.2	0.10	8.4
felem_sum	16	16	0	0	0.41	5.6	0.14	4.7	0.24	10.3	0.09	8.2
smallfelem_mul	88	136	0	30	2.80	91.9	0.17	6.4	1.22	95.4	0.11	9.4
smallfelem_neg	26	28	0	0	0.1	5.4	0.19	4.9	0.27	9.7	0.12	8.6
smallfelem_square	60	108	0	20	1.92	55.8	0.15	6.3	0.85	55.5	0.10	9.2
openssl/ecp_nistp521.c[2] (MathSAT, SMT-LIB2 format)												
felem_diff64	45	45	18	18	0.81	6.9	0.20	6.4	0.48	11.4	0.13	9.3
felem_diff128	45	72	18	18	1.13	7.9	0.21	6.4	0.47	11.9	0.12	9.2
felem_neg	27	27	0	0	0.77	6.4	0.18	5.3	0.48	10.0	0.12	8.6
felem_reduce	122	155	74	72	4.10	7.8	0.24	6.7	2.06	10.8	0.14	9.6
felem_scalar	27	27	0	0	0.80	28.4	0.14	5.0	0.36	29.0	0.09	8.3
felem_scalar64	27	27	0	0	0.82	28.2	0.15	4.9	0.35	28.9	0.09	8.3
felem_scalar128	27	27	0	0	1.26	48.4	0.14	5.0	0.41	48.8	0.09	8.4
felem_sum64	36	36	0	0	0.49	6.0	0.14	5.2	0.29	10.0	0.10	8.3
felem_diff_128_64	54	54	0	0	1.34	7.2	0.29	6.0	0.68	11.4	0.15	8.7
felem_mul	188	188	0	0	23.92	187.0	0.22	6.6	3.13	182.5	0.13	9.5
felem_square	111	111	0	0	7.38	95.5	0.21	6.4	0.99	103.9	0.13	9.3
boringssl/fiat/curve25519.c (MathSAT, SMT-LIB2 format)												
fe_add	11	20	0	0	0.33	5.3	0.14	4.8	0.20	10.0	0.10	8.2
fe_mul_impl	96	108	9	22	18.39	452.9	0.21	6.4	5.11	473.9	0.13	9.2
fe_mul121666	43	43	9	14	1.12	18.4	0.20	5.7	0.62	21.2	0.11	8.6
fe_sqr_impl	73	85	9	22	10.59	278.7	0.26	6.3	3.11	293.0	0.12	9.2
fe_sub	15	25	0	0	0.51	5.9	0.19	5.0	0.28	10.4	0.11	8.8
x25519_scalar_mult_generic[1]	927	1073	161	212	470.68	1489.0	120.33	5726	118.95	1579	91.99	5766

[1] Only an iteration of Montgomery Ladder step is verified.
[2] https://github.com/openssl/openssl commit 13fbce1

To illustrate the usability of our tools, the experiments are carried out on two machines: a Macbook Pro and a dedicated Linux server. Table 5 in Table 5 lists experimental results. We extract GCC 8.1.0 GIMPLE code of C functions in NaCl 20110221, wolfSSL 3.15.5, bitcoin 0.17.0.1, OpenSSL 1.1.1, and BoringSSL master branch with git commit hash f36c3ad. Machine M1 is a Macbook Pro running OS X 10.11.6 with a dual-core 2.7GHz CPU and 16GB RAM. M2 is a Linux server running Ubuntu 16.04.5 LTS with two octa-core 3.20GHz CPU and 1024GB RAM. The SMT solvers Boolector 3.0.0 and MathSAT 5.5.4 are used. We also use the computer algebra system SINGULAR 4.1.1. Columns L_{IR} and L_{CL} are the numbers of GIMPLE statements and CRYPTOLINE instructions respectively. Columns *D* and *P* are the numbers of manually translated CRYPTOLINE and annotated assert and assume instructions respectively. *TR* and *MR* are the time (in seconds) and the peak memory usage (in MB) when checking safety and range properties. *TA* and *MA* are for the algebraic properties. Subscripts of *TR*, *MR*, *TA*, and *MA* denote the machine used. OOM means out of memory and OOT means out of time (greater than 86400 seconds).

Our experiments show that almost all functions can be verified on a laptop in 15 minutes. Particularly, the C implementations of the Montgomery ladder step for Curve25519 in OpenSSL and BoringSSL are verified in 13 and 10 minutes respectively. If a dedicated server is used, the verification time shortens to 5 and 4 minutes respectively. 25 functions can be checked without manual translation nor annotation; 39 (=25 + 14) functions require less than 10% of manual translation (column *D*). Although our technique is not fully automatic, it does not require much human intervention.

We have verified 82 C functions implementing field and group operations for cryptographic primitives in 5 cryptography libraries: NaCl [23], wolfSSL [26], bitcoin [19], OpenSSL [24], and BoringSSL [9, 11]. The 32-bit C implementations in wolfSSL's Curve25519 is found in OpenSSL and LibreSSL [21] as well. The secp256k1 cryptography library in bitcoin is also used by other cryptocurrency including Ethereum [20], Zcash [27], Ripple [25], and Litecoin [22].

The elliptic curve Curve25519 is defined over the field $\mathbb{Z}_{2^{255}-19}$ and implemented in NaCl, wolfSSL, OpenSSL, and BoringSSL. Our verification exposes a potential missing carry in NaCl 20110221. We have reported our finding. The BoringSSL implementation is synthesized by Fiat-Crypto [9]. The synthesized 64-bit unsigned C implementation is verifiably correct at the C source level. We extract its GIMPLE representation after machine-independent optimization. Interestingly, vector statements are used in the optimized sequential implementation of Montgomery ladder step. Our result shows that the implementation is still correct after vectorization. Due to errors in the computer algebra system SINGULAR, the 32-bit signed implementation in wolfSSL is almost verified except one algebraic property. We are exploring other means to solve the corresponding ideal membership problem. The bitcoin secp256k1 cryptography library uses a Koblitz curve over the field $\mathbb{Z}_{2^{256}-2^{32}-2^9-2^8-2^7-2^6-2^4-1}$. 24 C functions for various field and group operations in the curve are verified. We also verify field operations in 3 NIST curves (P224, P256, and P521) over different fields ($\mathbb{Z}_{2^{224}-2^{96}+1}$, $\mathbb{Z}_{2^{256}-2^{224}+2^{192}+2^{96}-1}$, and $\mathbb{Z}_{2^{521}-1}$ respectively) from OpenSSL. To the best of our knowledge, this is the first verification result of cryptographic programs in NaCl, wolfSSL, and bitcoin.

ACKNOWLEDGMENTS

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

[2] Reynald Affeldt and Nicolas Marti. 2007. An Approach to Formal Verification of Arithmetic Functions in Assembly. In *Advances in Computer Science (LNCS)*, Mitsu Okada and Ichiro Satoh (Eds.), Vol. 4435. Springer, Heidelberg Dordrecht London New York, 346–360.

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

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 1807–1823.

[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems* 37, 2 (2015), 7:1–7:31.

[6] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security Symposium 2015*. USENIX Association, 207–221.

[7] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium 2017*. USENIX Association, 917–934.

[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. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, New York, NY, USA, 299–309.

[9] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA.

[10] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A Verified, Efficient Embedding of a Verifiable Assembly Language. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 63:1–63:30.

[11] Google. 2019. BoringSSL. https://boringssl.googlesource.com/boringssl/.

[12] Carl A. Gunter. 1993. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, MA, USA.

[13] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[14] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Verifying Arithmetic in Cryptographic C Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering*, Julia Lawall and Darko Marinov (Eds.). IEEE, San Diego, CA, USA.

[15] John C. Mitchell. 1996. *Foudations for Programming Languages*. The MIT Press, Cambridge, MA, USA.

[16] Magnus O. Myreen and Gregorio Curello. 2013. Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code. In *Certified Programs and Proofs (LNCS)*, Vol. 8307. Springer, Heidelberg Dordrecht London New York, 66–81.

[17] Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Orna Grumberg and Michael Huth (Eds.), Vol. 4424. Springer, Heidelberg Dordrecht London New York, 568–582.

[18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2018. Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk). In *29th International Conference on Concurrency Theory (LIPIcs)*, Sven Schewe and Lijun Zhang (Eds.), Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Beijing, China, 4:1–4:16.

[19] The Bitcoin Developers. 2019. Bitcoin Source Code. https://github.com/bitcoin/bitcoin.
[20] The Ethereum Developers. 2019. Ethereum Source Code. https://github.com/ethereum/go-ethereum.
[21] The LibreSSL Developers. 2019. LibreSSL. https://www.libressl.org/.
[22] The Litecoin Developers. 2019. Litecoin Source Code. https://github.com/litecoin-project/litecoin.
[23] The NaCl Developers. 2011. NaCl: Networking and Cryptography library. https://nacl.cr.yp.to/.
[24] The OpenSSL Developers. 2019. OpenSSL. https://www.openssl.org/.
[25] The Ripple Developers. 2019. Ripple Source Code. https://github.com/ripple/rippled.
[26] The wolfSSL Developers. 2019. wolfSSL Source Code. https://github.com/wolfSSL/wolfssl.
[27] The Zcash Developers. 2019. Zcash Source Code. https://github.com/zcash/zcash.
[28] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, New York, NY, USA, 1973–1987.
[29] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 2007–2020.
[30] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 1789–1806.