Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs

Ming-Hsien Tsai
Academia Sinica
Taipei, Taiwan
mhtsai208@gmail.com

Bow-Yaw Wang
Academia Sinica
Taipei, Taiwan
bywang@iis.sinica.edu.tw

Bo-Yin Yang
Academia Sinica
Taipei, Taiwan
byyang@iis.sinica.edu.tw

ABSTRACT

Mathematical constructs are necessary for computation on the underlying algebraic structures of cryptosystems. They are often written in assembly language and optimized manually for efficiency. We develop a certified technique to verify low-level mathematical constructs in X25519, the default elliptic curve Diffie-Hellman key exchange protocol used in OpenSSH. Our technique translates an algebraic specification of mathematical constructs into an algebraic problem. The algebraic problem in turn is solved by the computer algebra system Singular. The proof assistant Coq certifies the translation and solution to algebraic problems. Specifications about output ranges and potential program overflows are translated to SMT problems and verified by SMT solvers. We report our case studies on verifying arithmetic computation over a large finite field and the Montgomery Ladderstep, a crucial loop in X25519.

CCS CONCEPTS

• **Security and privacy → Logic and verification**;

KEYWORDS

cryptography; verification; low-level implementation

1 INTRODUCTION

In order to take advantage of computer security offered by modern cryptography, cryptosystems must be realized by cryptographic programs where mathematical constructs are required to compute on the underlying algebraic structures of cryptosystems. Such mathematical constructs are frequently invoked in cryptographic programs; they are often written in assembly language and manually optimized for efficiency. Security of cryptosystems could be compromised should programming errors in mathematical constructs be exploited by adversaries. Subsequently, security guarantees of cryptographic programs depend heavily on the correctness of mathematical constructs. In order to build secure cryptosystems, we develop a certified technique to verify low-level mathematical constructs used in the security protocol X25519 automatically in this paper.

X25519 is an Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol; it is a high-performance cryptosystem designed to use the secure elliptic curve Curve25519 [8]. Curve25519 is an elliptic curve offering 128 bits of security when used with ECDH. In addition to allowing high-speed elliptic curve arithmetic, it is easier to implement properly, not covered by any known patents, and moreover less susceptible to implementation pitfalls such as weak random-number generators. Its parameters were also selected by easily described mathematical principles. These characteristics make Curve25519 a preferred choice for those who are leery of curves which might have intentionally inserted backdoors, such as those standardized by the United States National Institute of Standards and Technology (NIST). Indeed, Curve25519 is currently the de facto alternative to the NIST P-256 curve. Consequently, X25519 has a wide variety of applications including the default key exchange protocol in OpenSSH since 2014 [31].

Most of the computation in X25519, in trade parlance, is in a "variable base point multiplication," and the centerpiece is the Montgomery Ladderstep. This is usually a large constant-time assembly program performing the finite-field arithmetic that implements the mathematics on Curve25519. Should the implementation of Montgomery Ladderstep be incorrect, so would that of X25519. Obviously for all its virtues, X25519 would be pointless if its implementation is incorrect. This may be even more relevant in cryptography than most of engineering, because cryptography is one of the few disciplines with the concept of an omnipresent adversary, constantly looking for the smallest edge — and hence eager to trigger any unlikely event. Revising a cryptosystem due to rare failures potentially leading to a cryptanalysis is not unheard of [24]. Thus, it is important for security that we can show the computations comprising the Montgomery Ladderstep or (even better) the X25519 protocol to be correct.

Several obstacles need be overcome for the verification of mathematical constructs in X25519. The key exchange protocol is based on a group induced by Curve25519. The elliptic curve is in turn defined over the Galois field $\mathbb{GF}(2^{255} - 19)$. To compute on the elliptic curve group, arithmetic computation over $\mathbb{GF}(2^{255} - 19)$ needs to be correctly implemented. Particularly, 255-bit multiplications modulo $2^{255} - 19$ must be verified. Worse, commodity computing devices do not support 255-bit arithmetic computation directly. Arithmetic over the Galois field needs to be implemented by sequences of 32- or 64-bit instructions of the underlying architectures. One has to verify that a sequence of 32- or 64-bit instructions indeed computes, say, a 255-bit multiplication over the finite field. Yet this is only a

This work was supported by the Ministry of Science and Technology (MOST), Taiwan, through projects no 103-2221-E-001-020-MY3 and 105-2221-E-001-014-MY3; and by Academia Sinica Thematic Project Socially Accountable Privacy Framework for Secondary Data Usage.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134076

single step in the operation on the elliptic curve group. In order to compute the group operation, another sequence of arithmetic computation over $\mathbb{GF}(2^{255} - 19)$ is needed. Particularly, a crucial step, the Montgomery Ladderstep, requires 18 arithmetic computations over $\mathbb{GF}(2^{255} - 19)$ [25]. The entire Ladderstep must be verified to ensure security guarantees offered by Curve25519.

In this paper, we focus on algebraic properties about low-level implementations of mathematical constructs in cryptographic programs as well as range properties about program outputs. Mathematical constructs by their nature perform computation on underlying algebraic structures. We aim to verify whether they perform the intended algebraic computation correctly. To this end, we propose the domain specific language bvCryptoLine with operations on fixed-width bit-vectors for low-level mathematical constructs. Algebraic pre- and post-conditions of programs together with range information about inputs and outputs in bvCryptoLine are specified as Hoare triples [23]. Such a specification is converted to static single assignment form and then translated into (1) an algebraic problem (called the modular polynomial equation entailment problem) [4, 22] via zCryptoLine with operations on $\mathbb{Z}$, (2) a range problem, and (3) the absence of program overflows/underflows. We use the computer algebra system Singular to solve the algebraic problem [21]. The proof assistant Coq is used to certify the correctness of translations, as well as solutions to algebraic problems computed by Singular [12]. As range problems are hard to be solved automatically with proof assistants, the range problem and the absence of program overflows/underflows are verified by SMT (Satisfiability Modulo Theories) solvers. Correctness of the translation to SMT formulas is again certified by Coq. The results of SMT solvers however are not certified in our implementation. The trusted computing base of our approach hence includes SMT solvers and Coq. The translation to bvCryptoLine is also included in the base if the program to be verified is not in bvCryptoLine. A fully certified integration of SMT solvers in Coq can be used to reduce the trusted computing base in the future [17].

We report case studies on verifying mathematical constructs used in the X25519 ECDH key exchange protocol [9, 10]. For each arithmetic operation (such as addition, subtraction, and multiplication) over $\mathbb{GF}(2^{255} - 19)$, their low-level real-world implementations are converted to our domain specific language bvCryptoLine manually. We specify algebraic properties of mathematical constructs in Hoare triples. Mathematical constructs are then verified against their algebraic specifications with our automatic technique. The implementation of the Montgomery Ladderstep is verified similarly.

We have the following contributions:

- We propose a domain specific language bvCryptoLine for modeling low-level mathematical constructs used in cryptographic programs.
- We give a certified verification condition generator from algebraic specifications of programs to the modular polynomial equation entailment problem.
- We give a certified translation from range problems and overflow/underflow checks to SMT formulas.
- We verify arithmetic computation over a finite field of order $2^{255} - 19$ and a critical program (the Montgomery Ladderstep) automatically.

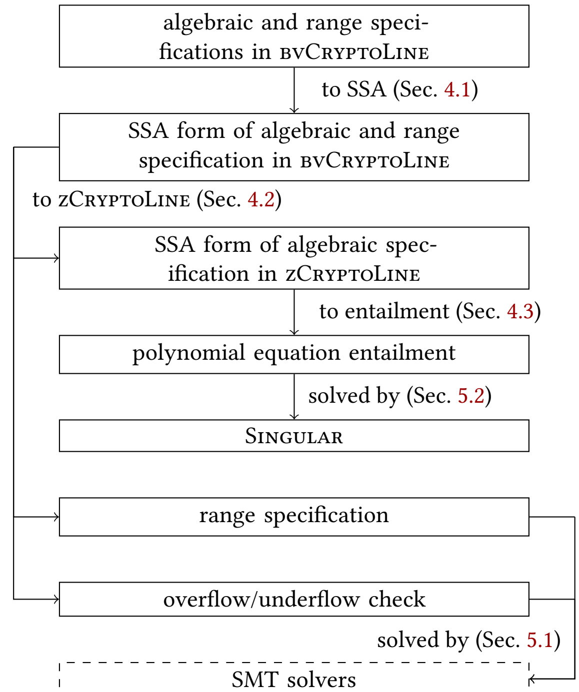

Figure 1: The verification flow. Except the answers from SMT solvers, all the translations and the answers from Singular are certified by Coq.

- To the best of our knowledge, our work is the first automatic and certified verification on real cryptographic programs with minimal human intervention.

Related Work. Low-level implementations of mathematical constructs have been formalized and manually proved in proof assistants [1–3, 26, 27]. A semi-automatic approach [14] has successfully verified a hand-optimized assembly implementation of the Montgomery Ladderstep with SMT solvers, manual program annotation, and a few Coq proofs. A C implementation of the Montgomery Ladderstep has been automatically verified with `gfverif` [11], which implements a specialized range analysis and translates verification problems to polynomial equations later solved by the Sage computer-algebra system [16]. Both the range analysis and the translation in `gfverif` are uncertified. Re-implementations of mathematical constructs in F* [18] have been verified using a combination of SMT solving and manual proofs. Vale [13] provides a meta language for defining syntax and semantics of assembly code. Several algorithms have been implemented in Vale and verified using SMT solvers with the help of manually constructed lemmas. Several cryptographic implementations in C and Java have been automatically verified by SAW to be equivalent to their reference implementations written in Cryptol [30] but the correctness of reference implementations is not proven and the verification results are not certified. The OpenSSL implementations of SHA-256 and HMAC have been formalized and manually proved in Coq [5, 6]. Synthesis of assembly codes for mathematical constructs has been proposed in [19]. Although the synthesized codes are correct by construction, they are not as efficient as hand-optimized assembly implementations.

This paper is organized as follows. After preliminaries (Section 2), our domain specific language is described in Section 3. Section 4 presents the translation to the algebraic problem. A certified translation from range and overflow/underflow checks to SMT formulas plus a certified solver for the algebraic problem are discussed in Section 5. Section 6 contains experimental results. It is followed by conclusions.

2 PRELIMINARIES

We write $\mathbb{B} = \{ff, tt\}$ for the Boolean domain. Let $\mathbb{N}$ and $\mathbb{Z}$ denote all natural numbers and all integers respectively. We use $[n]$ to denote the set $\{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$.

A *monoid* $\mathcal{M} = (M, \epsilon, \cdot)$ consists of a set M and an associative binary operator $\cdot$ on M with the *identity* $\epsilon \in M$. That is, $\epsilon \cdot m = m \cdot \epsilon = m$ for every $m \in M$. A *group* $\mathcal{G} = (G, 0, +)$ is an algebraic structure where $(G, 0, +)$ is a monoid and there is a $-a \in G$ such that $(-a) + a = a + (-a) = 0$ for every $a \in G$. The element $-a$ is called the *inverse* of a. $\mathcal{G}$ is *Abelian* if the operator $+$ is commutative. A *ring* $\mathcal{R} = (R, 0, 1, +, \times)$ with $0 \neq 1$ is an algebraic structure such that

- $(R, 0, +)$ is an Abelian group;
- $(R, 1, \times)$ is a monoid; and
- $\times$ is distributive over $+$: $a \times (b + c) = a \times b + a \times c$ for every $a, b, c \in R$.

If $\times$ is commutative, $\mathcal{R}$ is a *commutative* ring. A *field* $\mathcal{F} = (F, 0, 1, +, \times)$ is a commutative ring where $(F \setminus \{0\}, 1, \times)$ is also a group. $(\mathbb{N}, 1, \times)$ is a monoid. $(\mathbb{Z}, 0, 1, +, \times)$ is a commutative ring but not a field. For any prime number ϱ, the set $\{0, \ldots, \varrho - 1\}$ with the addition and multiplication modulo ϱ forms a *Galois field* of order ϱ (written $\mathbb{GF}(\varrho)$). We focus on Galois fields of very large orders, in particular, $\varrho = 2^{255} - 19$.

Fix a set of variables $\vec{x}$. $\mathcal{R}[\vec{x}]$ is the set of polynomials over $\vec{x}$ with coefficients in the ring $\mathcal{R}$. $\mathcal{R}[\vec{x}]$ is a ring. A set $I \subseteq \mathcal{R}[\vec{x}]$ is an *ideal* if

- $f + g \in I$ for every $f, g \in I$; and
- $h \times f \in I$ for every $h \in \mathcal{R}[\vec{x}]$ and $f \in I$.

Given $G \subseteq \mathcal{R}[\vec{x}]$, $\langle G \rangle$ is the minimal ideal containing G; G are the *generators* of $\langle G \rangle$. The *ideal membership* problem is to decide if $f \in I$ for a given ideal I and $f \in \mathcal{R}[\vec{x}]$.

Let $\mathbb{V}^w$ be the set of all bit-vectors with a bit-width w. The unsigned value of $b \in \mathbb{V}^w$ is denoted by $|b|$. For a natural number or an integer n, let $bv^w(n)$ be the two's complement representation of n in a bit-width w. We use the following common operators for fixed-width bit-vectors: $\mathbb{V}^w +_{\mathbb{V}} \mathbb{V}^w : \mathbb{V}^w$ for addition, $\mathbb{V}^w -_{\mathbb{V}} \mathbb{V}^w : \mathbb{V}^w$ for subtraction, $\mathbb{V}^w \times_{\mathbb{V}} \mathbb{V}^w : \mathbb{V}^w$ for multiplication, $\mathbb{V}^{w_1} ._{\mathbb{V}} \mathbb{V}^{w_2} : \mathbb{V}^{w_1+w_2}$ for concatenation, $\mathbb{V}^w \#_{\mathbb{V}} n : \mathbb{V}^{w+n}$ for zero extension, $\mathbb{V}^w \ll_{\mathbb{V}} n : \mathbb{V}^w$ for left-shifting, $\mathbb{V}^w \gg_{\mathbb{V}} n : \mathbb{V}^w$ for logical right-shifting, and $\mathbb{V}^w[i, j] : \mathbb{V}^{i-j+1}$ with $0 \leq j \leq i < w$ for bits extraction. We also assume comparison operators $<_{\mathbb{V}}$ and $\leq_{\mathbb{V}}$ between unsigned values of bit-vectors.

Given a bit-vector $b \in \mathbb{V}^{2w}$, define $hi_{\mathbb{V}}(b) \triangleq b[2w - 1, w]$ for the extraction of higher w bits, and $lo_{\mathbb{V}}(b) \triangleq b[w - 1, 0]$ for the extraction of lower w bits. For operations $\bullet \in \{+_{\mathbb{V}}, -_{\mathbb{V}}, \times_{\mathbb{V}}\}$, we define their extended version $\bullet^{\#}$ which performs the original operation after doubling the width of operands by zero extension. In the extended operations, the width of operands is doubled only once. For example, given $b_1, b_2, b_3 \in \mathbb{V}^w$, we have $b_1 +^{\#}_{\mathbb{V}} b_1 \triangleq (b_1 \#_{\mathbb{V}} w) +_{\mathbb{V}} (b_2 \#_{\mathbb{V}} w)$ and $b_1 +^{\#}_{\mathbb{V}} b_2 +^{\#}_{\mathbb{V}} b_3 \triangleq (b_1 \#_{\mathbb{V}} w) +_{\mathbb{V}} (b_2 \#_{\mathbb{V}} w) +_{\mathbb{V}} (b_3 \#_{\mathbb{V}} w)$.

3 DOMAIN SPECIFIC LANGUAGE – BVCRYPTOLINE

One of the big issues with modern cryptography is how the assumptions match up with reality. In many situations, unexpected channels through which information can leak to the attacker may cause the cryptosystem to be broken. Typically this is about timing or electric power used. In side-channel resilient implementations, the execution time is kept constant (as much as possible) to prevent unexpected information leakage. Constant execution time however is harder to achieve than one would imagine. Modern processors have caches and multitasking. This makes it possible for one execution thread, even when no privilege is conferred, to affect the running time of another – simply by caching a sufficient amount of its own data in correct locations through repeated accesses, and then observing the running time of the other thread. The instructions in the other thread which use the "evicted" data (to make room for the data of the eavesdropping thread) then have to take more time getting its data back to the cache [7].

Thus, the innocuous actions of executing (a) a conditional branch instruction dependent on a secret bit, and (b) an indirect load instruction using a secret value in the register as the address, are both potentially dangerous leaks of information. Consequently, we are not often faced with secret-dependent branching or table-lookups in the assembly instructions, but a language describing cryptographic code might include pseudo-instructions to cover instruction sequences, phrases in the language if you will, that is used to achieve the same effect. The domain specific language bvCryptoLine is designed based on the same principles. Conditional branches and indirect memory accesses are not admitted in bvCryptoLine.

Assume some machine architecture with a positive wordsize w. A program is a straight line of instructions over bit-vectors with bit-width w.

$$
\begin{array}{rcl}
Var & ::= & x \mid y \mid z \mid \cdots \\
bAtom & ::= & Var \mid \mathbb{V}^w \\
bStmt & ::= & Var \leftarrow bAtom \\
& & \mid Var \leftarrow bAtom + bAtom \\
& & \mid Var\ Var \leftarrow bAtom + bAtom \\
& & \mid Var \leftarrow bAtom + bAtom + Var \\
& & \mid Var\ Var \leftarrow bAtom + bAtom + Var \\
& & \mid Var \leftarrow bAtom - bAtom \\
& & \mid Var \leftarrow bAtom \times bAtom \\
& & \mid Var\ Var \leftarrow bAtom \times bAtom \\
& & \mid Var \leftarrow bAtom \ll \mathbb{V}^w \\
& & \mid Var\ Var \leftarrow bAtom @ \mathbb{V}^w \\
& & \mid Var\ Var \leftarrow (bAtom.bAtom) \ll \mathbb{V}^w
\end{array}
$$

Let $bSt \triangleq Var \rightarrow \mathbb{V}^w$ and $\nu \in bSt$ be a *state* (or *valuation*). That is, a state ν is a mapping from variables to bit-vectors in $\mathbb{V}^w$.

$$
\begin{array}{lcl}
\nu & \overset{v \leftarrow a_1 + a_2}{\Longrightarrow} & \nu[v \leftarrow \llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)] \\
\nu & \overset{c\ v \leftarrow a_1 + a_2}{\Longrightarrow} & \nu[v \leftarrow lo_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu))][c \leftarrow hi_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu))] \\
\nu & \overset{v \leftarrow a_1 + a_2 + y}{\Longrightarrow} & \nu[v \leftarrow lo_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket y \rrbracket_{\mathbb{V}}(\nu))] \\
\nu & \overset{c\ v \leftarrow a_1 + a_2 + y}{\Longrightarrow} & \nu[v \leftarrow lo_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket y \rrbracket_{\mathbb{V}}(\nu))][c \leftarrow hi_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu) +^{\#}_{\mathbb{V}} \llbracket y \rrbracket_{\mathbb{V}}(\nu))] \\
\nu & \overset{v \leftarrow a_1 - a_2}{\Longrightarrow} & \nu[v \leftarrow \llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) -_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)] \\
\nu & \overset{v \leftarrow a_1 \times a_2}{\Longrightarrow} & \nu[v \leftarrow \llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) \times_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)] \\
\nu & \overset{v_h\ v_l \leftarrow a_1 \times a_2}{\Longrightarrow} & \nu[v_h \leftarrow hi_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) \times^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu))][v_l \leftarrow lo_{\mathbb{V}}(\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) \times^{\#}_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu))] \\
\nu & \overset{v \leftarrow a \ll n}{\Longrightarrow} & \nu[v \leftarrow \llbracket a \rrbracket_{\mathbb{V}}(\nu) \ll_{\mathbb{V}} |n|] \\
\nu & \overset{v_h\ v_l \leftarrow a @ n}{\Longrightarrow} & \nu[v_h \leftarrow \llbracket a \rrbracket_{\mathbb{V}}(\nu) \gg_{\mathbb{V}} |n|][v_l \leftarrow (\llbracket a \rrbracket_{\mathbb{V}}(\nu) \ll_{\mathbb{V}} (w -_{\mathbb{N}} |n|)) \gg_{\mathbb{V}} (w -_{\mathbb{N}} |n|)] \\
\nu & \overset{v_h\ v_l \leftarrow (a_1 . a_2) \ll n}{\Longrightarrow} & \nu[v_h \leftarrow hi_{\mathbb{V}}((\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) ._{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)) \ll_{\mathbb{V}} |n|)][v_l \leftarrow (lo_{\mathbb{V}}((\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) ._{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)) \ll_{\mathbb{V}} |n|)) \gg_{\mathbb{V}} |n|]
\end{array}
$$

Figure 2: Transition relation *bTr* for bvCryptoLine.

Define $\nu[v \leftarrow d](u) \triangleq \begin{cases} d & \text{if } u = v \\ \nu(u) & \text{otherwise} \end{cases}$. Define the semantic function $\llbracket \cdot \rrbracket_{\mathbb{V}}(\nu)$ for variables and atoms as follows.

$$
\begin{array}{rcl}
\llbracket v \rrbracket_{\mathbb{V}}(\nu) & \triangleq & \nu(v) \text{ for } v \in Var \\
\llbracket a \rrbracket_{\mathbb{V}}(\nu) & \triangleq & \begin{cases} \nu(v) & \text{if } a \text{ is a variable } v \\ b & \text{if } a \text{ is a bit-vector } b \end{cases}
\end{array}
$$

Consider the transition relation $bTr \subseteq bSt \times bStmt \times bSt$ defined in Figure 2 where $\nu \overset{s}{\Longrightarrow} \nu'$ denotes $(\nu, s, \nu') \in bTr$ for $\nu, \nu' \in bSt$ and $s \in bStmt$. Basically, $v \leftarrow a_1 + a_2$ is addition, $c\ v \leftarrow a_1 + a_2$ is addition with carry bit placed in c, $v \leftarrow a_1 + a_2 + y$ is addition of atoms plus a variable y, $c\ v \leftarrow a_1 + a_2 + y$ is addition of atoms plus a variable y with carry bit placed in c, $v \leftarrow a_1 - a_2$ is subtraction, $v \leftarrow a_1 \times a_2$ is multiplication, $v_h\ v_l \leftarrow a_1 \times a_2$ is full multiplication, $v \leftarrow a \ll n$ is left-shifting, $v_h\ v_l \leftarrow a@n$ is splitting at position n, and $v_h\ v_l \leftarrow (a_1.a_2) \ll n$ is left-shifting of higher n bits from a_2 to a_1. The variable y in $v \leftarrow a_1 + a_2 + y$ and $c\ v \leftarrow a_1 + a_2 + y$ is intended but not restricted to be carry bits.

A *program* is a sequence of statements. We denote the empty program by ϵ.

$$
bProg \quad ::= \quad \epsilon \mid bStmt; bProg
$$

Observe that conditional branches are not allowed in our domain specific language to prevent timing attacks. The semantics of a program is defined by the relation $bTr^* \subseteq bSt \times bProg \times bSt$ where $(\nu, \epsilon, \nu) \in bTr^*$ and $(\nu, s; p, \nu'') \in bTr^*$ if there is a ν' with $(\nu, s, \nu') \in bTr$ and $(\nu', p, \nu'') \in bTr^*$. We write $\nu \overset{p}{\Longrightarrow} \nu'$ when $(\nu, p, \nu') \in bTr^*$.

For specifications, $\top$ denotes the Boolean value *tt*. We allow two kinds of specifications, namely algebraic specifications evaluated on domain $\mathbb{Z}$ and range specifications evaluated on domain $\mathbb{V}^w$. Atomic predicates in an algebraic specification include polynomial equations $e_1 = e_2$ and modular polynomial equations $e_1 \equiv e_2 \bmod e_3$ where $e_i \in bExp_a$ is a polynomial expression for $i \in \{1, 2, 3\}$. An *algebraic predicate* $q_a \in bPred_a$ is then a conjunction of atomic algebraic predicates.

$$
\begin{array}{rcl}
bExp_a & ::= & \mathbb{Z} \mid Var \mid -\ bExp_a \mid bExp_a + bExp_a \\
 & & \mid bExp_a - bExp_a \mid bExp_a \times bExp_a \\
bPred_a & ::= & \top \mid bExp_a = bExp_a \mid bExp_a \equiv bExp_a \bmod bExp_a \\
 & & \mid bPred_a \wedge bPred_a
\end{array}
$$

Given a state $\nu \in bSt$ and an expression $e \in bExp_a$, $\llbracket e \rrbracket_{\mathbb{Z}}(\nu)$ denotes the value of e on ν.

$$
\begin{array}{rcl}
\llbracket n \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & n \text{ for } n \in \mathbb{Z} \\
\llbracket v \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & |\nu(v)| \text{ for } v \in Var \\
\llbracket -e \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & -_{\mathbb{Z}} \llbracket e \rrbracket_{\mathbb{Z}}(\nu) \\
\llbracket e_1 + e_2 \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & \llbracket e_1 \rrbracket_{\mathbb{Z}}(\nu) +_{\mathbb{Z}} \llbracket e_2 \rrbracket_{\mathbb{Z}}(\nu) \\
\llbracket e_1 - e_2 \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & \llbracket e_1 \rrbracket_{\mathbb{Z}}(\nu) -_{\mathbb{Z}} \llbracket e_2 \rrbracket_{\mathbb{Z}}(\nu) \\
\llbracket e_1 \times e_2 \rrbracket_{\mathbb{Z}}(\nu) & \triangleq & \llbracket e_1 \rrbracket_{\mathbb{Z}}(\nu) \times_{\mathbb{Z}} \llbracket e_2 \rrbracket_{\mathbb{Z}}(\nu)
\end{array}
$$

For an algebraic predicate $q_a \in bPred_a$, we write $\mathbb{V}^w \models q_a[\nu]$ if q_a evaluates to *tt* using the evaluation function $\llbracket e \rrbracket_{\mathbb{Z}}(\nu)$ for every subexpression e in q.

We admit comparison between atoms in range specifications as atomic range predicates[1]. A *range predicate* $q_r \in bPred_r$ is a conjunction of atomic range predicates.

$$
\begin{array}{rcl}
bPred_r & ::= & \top \mid bAtom < bAtom \mid bAtom \leq bAtom \\
 & & \mid bPred_r \wedge bPred_r
\end{array}
$$

We use $a_l \circ a_1, a_2, \ldots, a_n \bullet a_r$ as a shorthand of the conjunction of $a_l \circ a_1 \wedge a_l \circ a_2 \wedge \cdots \wedge a_l \circ a_n$ and $a_1 \bullet a_r \wedge a_2 \bullet a_r \wedge \cdots \wedge a_n \bullet a_r$ where $\circ, \bullet \in \{<, \leq\}$. For $q_r \in bPred_r$ and $\nu \in bSt$, we write $\mathbb{V}^w \models q_r[\nu]$ if one of the following holds.

- q is $\top$.
- q is $a_1 < a_2$ and $\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) <_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)$.
- q is $a_1 \leq a_2$ and $\llbracket a_1 \rrbracket_{\mathbb{V}}(\nu) \leq_{\mathbb{V}} \llbracket a_2 \rrbracket_{\mathbb{V}}(\nu)$.
- q is $q_1 \wedge q_2$, $\mathbb{V}^w \models q_1[\nu]$, and $\mathbb{V}^w \models q_2[\nu]$.

A *predicate* $q \in bPred$ consists of an algebraic predicate and a range predicate.

$$
\begin{array}{rcl}
bPred & ::= & bPred_a \,\vert\, bPred_r \\
bSpec & ::= & (\!| bPred |\!)\, bProg\, (\!| bPred |\!)
\end{array}
$$

For $\nu \in bSt$ and $q \in bPred$, we write $\mathbb{V}^w \models q[\nu]$ if q evaluates to *tt*; ν is called a *q-state*. We follow Hoare's formalism in specifications of mathematical constructs [23] and call $(\!| q |\!)\ p\ (\!| q' |\!)$ a *specification* if $q, q' \in bPred$, an *algebraic specification* if $q, q' \in bPred_a$, and a *range specification* if $q, q' \in bPred_r$. In $(\!| q |\!)\ p\ (\!| q' |\!)$, q and q' are the *pre-* and *post-conditions* of p respectively. Given $q, q' \in bPred$ ($q, q' \in bPred_a$, or $q, q' \in bPred_r$) and $p \in bProg$, $(\!| q |\!)\ p\ (\!| q' |\!)$ is

[1] In our implementation, comparison between bit-vector expressions is allowed, not only between atoms.

1: $r_0 \leftarrow x_0;$
2: $r_1 \leftarrow x_1;$
3: $r_2 \leftarrow x_2;$
4: $r_3 \leftarrow x_3;$
5: $r_5 \leftarrow x_4;$
6: $r_0 \leftarrow r_0 + 0xFFFFFFFFFFFDA;$
7: $r_1 \leftarrow r_1 + 0xFFFFFFFFFFFFE;$
8: $r_2 \leftarrow r_2 + 0xFFFFFFFFFFFFE;$
9: $r_3 \leftarrow r_3 + 0xFFFFFFFFFFFFE;$
10: $r_4 \leftarrow r_4 + 0xFFFFFFFFFFFFE;$
11: $r_0 \leftarrow r_0 - y_0;$
12: $r_1 \leftarrow r_1 - y_1;$
13: $r_2 \leftarrow r_2 - y_2;$
14: $r_3 \leftarrow r_3 - y_3;$
15: $r_4 \leftarrow r_4 - y_4;$

Figure 3: Subtraction bSub

valid (written $\models (\!|q|\!)\ p\ (\!|q'|\!)$) if for every $v, v' \in bSt$, $\mathbb{V}^w \models q[v]$ and $v \overset{p}{\Longrightarrow} v'$ imply $\mathbb{V}^w \models q'[v']$. Less formally, $\models (\!|q|\!)\ p\ (\!|q'|\!)$ if executing p from a q-state always results in a q'-state.

Given a statement $s \in bStmt$ and a state $v \in bSt$, the function StmtSafe (Algorithm 1) checks if executing s from v neither overflows nor underflows. We call a statement s *safe* at a state v if StmtSafe(s, v) evaluates to tt. A program p is safe at a state v, denote by ProgSafe(p, v), if (1) $p = \epsilon$, or (2) $p = s; pp$, StmtSafe(s, v) $= tt$, and for all $v' \in bSt$, $v \overset{s}{\Longrightarrow} v'$ implies ProgSafe(pp, v'). A program is safe if it is safe at every state.

Algorithm 1 Safety Test for Statements

function StmtSafe(s, v)
  match s **with**
    case $v \leftarrow a$: **return** tt
    case $v \leftarrow a_1 + a_2$:
      return $hi_{\mathbb{V}}([\![a_1]\!]_{\mathbb{V}}(v) +^{\#}_{\mathbb{V}} [\![a_2]\!]_{\mathbb{V}}(v)) = bv^w(0)$
    case $c\ v \leftarrow a_1 + a_2$: **return** tt
    case $v \leftarrow a_1 + a_2 + y$:
      return $hi_{\mathbb{V}}([\![a_1]\!]_{\mathbb{V}}(v) +^{\#}_{\mathbb{V}} [\![a_2]\!]_{\mathbb{V}}(v) +^{\#}_{\mathbb{V}} [\![y]\!]_{\mathbb{V}}(v)) = bv^w(0)$
    case $c\ v \leftarrow a_1 + a_2 + y$: **return** tt
    case $v \leftarrow a_1 - a_2$:
      return $hi_{\mathbb{V}}([\![a_1]\!]_{\mathbb{V}}(v) -^{\#}_{\mathbb{V}} [\![a_2]\!]_{\mathbb{V}}(v)) = bv^w(0)$
    case $v \leftarrow a_1 \times a_2$:
      return $hi_{\mathbb{V}}([\![a_1]\!]_{\mathbb{V}}(v) \times^{\#}_{\mathbb{V}} [\![a_2]\!]_{\mathbb{V}}(v)) = bv^w(0)$
    case $v_h\ v_l \leftarrow a_1 \times a_2$: **return** tt
    case $v \leftarrow a \ll n$:
      return $[\![a]\!]_{\mathbb{V}}(v) <_{\mathbb{V}} (bv^w(1) \ll_{\mathbb{V}} (w -_{\mathbb{N}} |n|))$
    case $v_h\ v_l \leftarrow a@n$: **return** tt
    case $v_h\ v_l \leftarrow (a_1.a_2) \ll n$:
      return $[\![a_1]\!]_{\mathbb{V}}(v) <_{\mathbb{V}} (bv^w(1) \ll_{\mathbb{V}} (w -_{\mathbb{N}} |n|)) \wedge |n| \leq_{\mathbb{N}} w$
end function

Figure 3 gives a simple yet real implementation of subtraction over $\mathbb{GF}(\varrho)$ with a bit-width 64. In the figure, a constant bit-vector is written in hexadecimal format starting with the prefix $0x$ and a number in $\mathbb{GF}(\varrho)$ is represented by five bit-vectors each with value less than or equal to $2^{51} +_{\mathbb{Z}} 2^{15}$. The variables x_0, x_1, x_2, x_3, x_4 for instance represent $radix51(x_4, x_3, x_2, x_1, x_0) \triangleq (2^{51\times4})x_4 + (2^{51\times3})x_3 + (2^{51\times2})x_2 + (2^{51\times1})x_1 + (2^{51\times0})x_0$. The result of subtraction is stored in the variables r_0, r_1, r_2, r_3, r_4, which are all required to be in the range from 0 to 2^{54}. Let $radix51_{\mathbb{V}}(x_4, x_3, x_2, x_1, x_0)$ denote the representation of $radix51(x_4, x_3, x_2, x_1, x_0)$ in $bExp_a$. Let $q_a \triangleq \top$, $q_r \triangleq 0 \leq x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ y_0,\ y_1,\ y_2,\ y_3,\ y_4 \leq bv^{64}(2^{51} +_{\mathbb{Z}} 2^{15})$, $q'_a \triangleq radix51_{\mathbb{V}}(x_4, x_3, x_2, x_1, x_0) - radix51_{\mathbb{V}}(y_4, y_3, y_2, y_1, y_0) \equiv radix51_{\mathbb{V}}(r_4, r_3, r_2, r_1, r_0) \bmod \varrho$, and $q'_r \triangleq 0 \leq r_0, r_1, r_2, r_3, r_4 < bv^{64}(2^{54})$. The specification of the mathematical construct is therefore

$$(\!|q_a \wedge q_r|\!) \quad \text{bSub} \quad (\!|q'_a \wedge q'_r|\!).$$

Note that the variables r_i's are added with constants after they are initialized with x_i's but before y_i's are subtracted from them. It is not hard to see that

$$\begin{aligned} 2\varrho = radix51(&|0xFFFFFFFFFFFFE|, |0xFFFFFFFFFFFFE|, \\ &|0xFFFFFFFFFFFFE|, |0xFFFFFFFFFFFFE|, \\ &|0xFFFFFFFFFFFDA|) \end{aligned}$$

after tedious computation. Hence

$$\begin{aligned} &radix51(r_4, r_3, r_2, r_1, r_0) \\ =&radix51(x_4, x_3, x_2, x_1, x_0) + 2\varrho - radix51(y_4, y_3, y_2, y_1, y_0) \\ \equiv&radix51(x_4, x_3, x_2, x_1, x_0) - radix51(y_4, y_3, y_2, y_1, y_0) \bmod \varrho. \end{aligned}$$

The program in Figure 3 is correct assuming that it is safe. Characteristics of large Galois fields are regularly exploited in mathematical constructs for correctness and efficiency. Our domain specific language can easily model such specialized programming techniques. Indeed, the reason for adding constants is to prevent underflow. If the constants were not added, the subtraction in lines 11 to 15 could give negative and hence incorrect results. We will show how to prove that the program is safe later.

4 TRANSFORMATION OF SPECIFICATIONS

Given $q_a, q'_a \in bPred_a$, $q_r, q'_r \in bPred_r$, and $p \in bProg$, we reduce the problem of checking $\models (\!|q_a \mid q_r|\!)\ p\ (\!|q'_a \mid q'_r|\!)$ to (1) the entailment problem of modular polynomial equations over integer variables proving $\models (\!|q_a|\!)\ p\ (\!|q'_a|\!)$ via an intermediate language zCryptoLine, (2) a range problem $\models (\!|q_r|\!)\ p\ (\!|q'_r|\!)$, and (3) a safety check of program p. The reduction is carried out by the following three transformations:

(1) Static single assignments. The program is transformed into static single assignments. Variables in pre- and post-conditions are also renamed (Section 4.1) [4].
(2) zCryptoLine. The algebraic specification $(\!|q_a|\!)\ p\ (\!|q'_a|\!)$ in bvCryptoLine is transformed to a specification in zCryptoLine so that the validity of the specification in zCryptoLine implies the validity of $(\!|q_a|\!)\ p\ (\!|q'_a|\!)$ in bvCryptoLine if the program p is safe. (Section 4.2).
(3) Modular polynomial equations. Validity of algebraic specifications in zCryptoLine is reduced to the entailment of modular polynomial equations (Section 4.3) [22].

For each transformation, we give an algorithm and establish the correctness of the algorithm in Coq [12]. Specifically, semantics for zCryptoLine and validity of specifications in zCryptoLine are formalized. The correctness of transformations is then certified by the proof assistant Coq. For static single assignments, we construct machine-checkable proofs for the soundness and completeness of the transformation. For modular polynomial equations, another Coq-certified proof shows the soundness of the transformation from the validity of the algebraic specification to the entailment of modular polynomial equations. In the following subsections, transformations and their correctness are elaborated in details.

4.1 Static Single Assignments

A program is in *static single assignment* form if every non-input variable is assigned at most once and no input variable is assigned [4]. Our next task is to transform any specification $(\!|q|\!)\ p\ (\!|q'|\!)$ to a specification of p in static single assignment form for any $q, q' \in bPred$ and $p \in bProg$. To avoid ambiguity, we consider *well-formed* programs where

- for every statement in the program with two lvalues such as $c\ v \leftarrow a_1 + a_2 + y$ with lvalues c and v, the two lvalues are different variables; and
- every non-input program variable must be assigned to a value before being used.

Our transformation maintains a finite mapping θ from variables to non-negative integers. For any variable v, $v^{\theta(v)}$ is the most recently assigned copy of v. For any atom a, a^θ is $v^{\theta(v)}$ when a is a variable v, and otherwise is b when a is a constant bit-vector b. Only the most recent copies of variables are referred in expressions. Algorithm 2 transforms algebraic expressions with the finite mapping θ by structural induction. Integers are unchanged. For each variable, its most recent copy is returned by looking up the mapping θ. Other algebraic expressions are transformed recursively.

Algorithm 2 Static Single Assignment Transformation for Algebraic Expressions

function SSAExpra(θ, e)
  match e **with**
    case i: **return** i
    case v: **return** $v^{\theta(v)}$
    case $-e'$: **return** $-$SSAExpra(θ, e')
    case $e_1 + e_2$:
      return SSAExpra(θ, e_1) + SSAExpra(θ, e_2)
    case $e_1 - e_2$:
      return SSAExpra(θ, e_1) $-$ SSAExpra(θ, e_2)
    case $e_1 \times e_2$:
      return SSAExpra(θ, e_1) $\times$ SSAExpra(θ, e_2)
end function

Similarly, algebraic and range predicates must refer to most recent copies of variables. They are transformed according to the finite mapping θ. Thanks to the formalization of finite mappings in Coq. Both algorithms are easily specified in Gallina. Let SSAPreda and SSAPredr denote the transformations for $bPred_a$ and $bPred_r$ respectively. The function SSAPred then transforms the algebraic part and the range part of a predicate separatedly with SSAPreda and SSAPredr, that is, given $q_a \in bPred_a$, $q_r \in bPred_r$, and a mapping θ, SSAPred$(\theta, q_a \mid q_r) \triangleq$ SSAPreda$(\theta, q_a) \mid$ SSAPredr(θ, q_r).

Algorithm 3 Static Single Assignment Transformation for Statements

function SSAStmt(θ, s)
  match s **with**
    case $v \leftarrow a$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a^\theta \rangle$
    case $v \leftarrow a_1 + a_2$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a_1^\theta + a_2^\theta \rangle$
    case $c\ v \leftarrow a_1 + a_2$:
      $\theta' \leftarrow \theta[c \leftarrow \theta(c) + 1]$
      $\theta'' \leftarrow \theta'[v \leftarrow \theta(v) + 1]$
      return $\langle \theta'', c^{\theta'(c)}\ v^{\theta''(v)} \leftarrow a_1^\theta + a_2^\theta \rangle$
    case $v \leftarrow a_1 + a_2 + y$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a_1^\theta + a_2^\theta + y^{\theta(y)} \rangle$
    case $c\ v \leftarrow a_1 + a_2 + y$:
      $\theta' \leftarrow \theta[c \leftarrow \theta(c) + 1]$
      $\theta'' \leftarrow \theta'[v \leftarrow \theta(v) + 1]$
      return $\langle \theta'', c^{\theta'(c)}\ v^{\theta''(v)} \leftarrow a_1^\theta + a_2^\theta + y^{\theta(y)} \rangle$
    case $v \leftarrow a_1 - a_2$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a_1^\theta - a_2^\theta \rangle$
    case $v \leftarrow a_1 \times a_2$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a_1^\theta \times a_2^\theta \rangle$
    case $v_h\ v_l \leftarrow a_1 \times a_2$:
      $\theta' \leftarrow \theta[v_h \leftarrow \theta(v_h) + 1]$
      $\theta'' \leftarrow \theta'[v_l \leftarrow \theta(v_l) + 1]$
      return $\langle \theta'', v_h^{\theta'(v_h)}\ v_l^{\theta''(v_l)} \leftarrow a_1^\theta \times a_2^\theta \rangle$
    case $v \leftarrow a \ll n$:
      $\theta' \leftarrow \theta[v \leftarrow \theta(v) + 1]$
      return $\langle \theta', v^{\theta'(v)} \leftarrow a^\theta \ll n \rangle$
    case $v_h\ v_l \leftarrow a@n$:
      $\theta' \leftarrow \theta[v_h \leftarrow \theta(v_h) + 1]$
      $\theta'' \leftarrow \theta'[v_l \leftarrow \theta(v_l) + 1]$
      return $\langle \theta'', v_h^{\theta'(v_h)}\ v_l^{\theta''(v_l)} \leftarrow a^\theta\ @\ n \rangle$
    case $v_h\ v_l \leftarrow (a_1.a_2) \ll n$:
      $\theta' \leftarrow \theta[v_h \leftarrow \theta(v_h) + 1]$
      $\theta'' \leftarrow \theta'[v_l \leftarrow \theta(v_l) + 1]$
      return $\langle \theta'', v_h^{\theta'(v_h)}\ v_l^{\theta''(v_l)} \leftarrow (a_1^\theta\ .\ a_2^\theta)\ @\ n \rangle$
end function

Statement transformation is slightly more complicated (Algorithm 3). For atoms and variables on the right hand side, they are transformed by the given finite mapping θ. The algorithm of statement transformation then updates θ and replaces assigned variables with their latest copies.

$$
\begin{array}{lll}
r_0^1 \leftarrow x_0^0; & r_0^2 \leftarrow r_0^1 + 0xFFFFFFFFFFFDA; & r_0^3 \leftarrow r_0^2 - y_0^0; \\
r_1^1 \leftarrow x_1^0; & r_1^2 \leftarrow r_1^1 + 0xFFFFFFFFFFFFE; & r_1^3 \leftarrow r_1^2 - y_1^0; \\
r_2^1 \leftarrow x_2^0; & r_2^2 \leftarrow r_2^1 + 0xFFFFFFFFFFFFE; & r_2^3 \leftarrow r_2^2 - y_2^0; \\
r_3^1 \leftarrow x_3^0; & r_3^2 \leftarrow r_3^1 + 0xFFFFFFFFFFFFE; & r_3^3 \leftarrow r_3^2 - y_3^0; \\
r_4^1 \leftarrow x_4^0; & r_4^2 \leftarrow r_4^1 + 0xFFFFFFFFFFFFE; & r_4^3 \leftarrow r_4^2 - y_4^0;
\end{array}
$$

Figure 4: Subtraction bSubSSA in Static Single Assignment Form

It is straightforward to transform programs to static single assignment form (Algorithm 4). Using the initial mapping θ_0 from variables to 0, the algorithm starts from the first statement and obtains an updated mapping with the statement in static single assignment form. It continues to transform the next statement with the updated mapping. Note that our algorithm works for any initial mapping but we choose θ_0 to simplify our Coq proof.

Algorithm 4 Static Single Assignment for Programs

```
function SSAProg(θ, p)
    match p with
        case ϵ: return ⟨θ, ϵ⟩
        case s; pp:
            ⟨θ′, s′⟩ ← SSAStmt(θ, s)
            ⟨θ′′, pp′′⟩ ← SSAProg(θ′, pp)
            return ⟨θ′′, s′; pp′′⟩
end function
```

Using the specifications of Algorithm 3 and 4 in Gallina, properties of these algorithms are formally proven in Coq. We first show that Algorithm 4 preserves well-formedness and produces a program in static single assignment form.

Lemma 4.1. *Let $\theta_0(v) = 0$ for every $v \in Var$ and $p \in bProg$ a well-formed program. If $\langle \hat{\theta}, \hat{p} \rangle = SSAProg(\theta_0, p)$, then $\hat{p}$ is well-formed and in static single assignment form.*

The next theorem shows that our transformation is both sound and complete. That is, a specification is valid if and only if its corresponding specification in static single assignment form is valid.

Theorem 4.2. *Let $\theta_0(v) = 0$ for every $v \in Var$. For every $q, q' \in bPred$ and $p \in bProg$,*

$\models (\!|q|\!)\ p\ (\!|q'|\!)$ if and only if $\models (\!|SSAPred(\theta_0, q)|\!)\ \hat{p}\ (\!|SSAPred(\hat{\theta}, q')|\!)$ where $\langle \hat{\theta}, \hat{p} \rangle = SSAProg(\theta_0, p)$.

Example. Figure 4 gives the subtraction program bSub in static single assignment form. Starting from 0, the index of a variable is incremented when the variable is assigned to an expression. After the static single assignment translation, the variables x_i's, y_i's are indexed by 0 and r_i's are indexed by 3 for $0 \le i \le 4$. Subsequently, variables in pre- and post-conditions of the specification for subtraction need to be indexed. Let $\hat{q}_a \triangleq \top$, $\hat{q}_r \triangleq 0 \le x_0^0, x_1^0, x_2^0, x_3^0, x_4^0, y_0^0, y_1^0, y_2^0, y_3^0, y_4^0 \le bv^{64}(2^{51} +_{\mathbb{Z}} 2^{15})$, $\hat{q}'_a \triangleq radix51_{\mathbb{V}}(x_4^0, x_3^0, x_2^0, x_1^0, x_0^0) - radix51_{\mathbb{V}}(y_4^0, y_3^0, y_2^0, y_1^0, y_0^0) \equiv radix51_{\mathbb{V}}(r_4^3, r_3^3, r_2^3, r_1^3, r_0^3) \bmod \varrho$, and $q'_r \triangleq 0 \le r_0^3, r_1^3, r_2^3, r_3^3, r_4^3 < bv^{64}(2^{54})$. The corresponding specification of in static single assignment form is then

$$(\!|\hat{q}_a \wedge \hat{q}_r|\!)\mathrm{bSubSSA}(\!|\hat{q}'_a \wedge \hat{q}'_r|\!).$$

4.2 zCryptoLine

Algebraic specifications in bvCryptoLine are transformed to modular polynomial equation entailment problems via an intermediate language zCryptoLine. A program in zCryptoLine is but a straight line of variable assignments on expressions. Consider the following syntactic classes:

$$
\begin{array}{rcl}
zExpr & ::= & \mathbb{Z} \mid Var \mid -zExpr \mid zExpr{+}zExpr \mid zExpr{-}zExpr \\
& & \mid zExpr{\times}zExpr \mid \mathrm{Pow}(zExpr, \mathbb{N})
\end{array}
$$

We allow exact integers as constants in zCryptoLine. Variables are thus integer variables. An expression can be a constant, a variable, or a negative expression. Additions, subtractions, and multiplications of expressions are available. The expression $\mathrm{Pow}(e, n)$ denotes e^n for any expression e and natural number n. More formally, let $zSt \triangleq Var \to \mathbb{Z}$ and $\mu \in zSt$ be a state. That is, a state μ in zCryptoLine is a mapping from variables to integers. Define the semantic function $[\![e]\!]_{\mathbb{Z}}(\mu)$ as follows.

$$
\begin{array}{rcl}
[\![i]\!]_{\mathbb{Z}}(\mu) & \triangleq & i \text{ for } i \in \mathbb{Z} \\
[\![v]\!]_{\mathbb{Z}}(\mu) & \triangleq & \mu(v) \text{ for } v \in Var \\
[\![-e]\!]_{\mathbb{Z}}(\mu) & \triangleq & -_{\mathbb{Z}}[\![e]\!]_{\mathbb{Z}}(\mu) \\
[\![e_0 + e_1]\!]_{\mathbb{Z}}(\mu) & \triangleq & [\![e_0]\!]_{\mathbb{Z}}(\mu) +_{\mathbb{Z}} [\![e_1]\!]_{\mathbb{Z}}(\mu) \\
[\![e_0 - e_1]\!]_{\mathbb{Z}}(\mu) & \triangleq & [\![e_0]\!]_{\mathbb{Z}}(\mu) -_{\mathbb{Z}} [\![e_1]\!]_{\mathbb{Z}}(\mu) \\
[\![e_0 \times e_1]\!]_{\mathbb{Z}}(\mu) & \triangleq & [\![e_0]\!]_{\mathbb{Z}}(\mu) \times_{\mathbb{Z}} [\![e_1]\!]_{\mathbb{Z}}(\mu) \\
[\![\mathrm{Pow}(e, n)]\!]_{\mathbb{Z}}(\mu) & \triangleq & ([\![e]\!]_{\mathbb{Z}}(\mu))^n
\end{array}
$$

In zCryptoLine, only assignments are allowed. The statement $v \leftarrow e$ assigns the value of e to the variable v. For bounded additions, multiplications, and right shifting, they are modeled by the construct Split in zCryptoLine. The statement $[v_h, v_l] \leftarrow \mathrm{Split}(e, n)$ splits the value of e into two parts; the lowest n bits are stored in v_l and the remaining higher bits are stored in v_h. Consider the relation $zTr \subseteq zSt \times zStmt \times zSt$ defined by $(\mu, v \leftarrow e, \mu[v \leftarrow [\![e]\!]_{\mathbb{Z}}(\mu)]) \in zTr$, and $(\mu, [v_h, v_l] \leftarrow \mathrm{Split}(e, n), \mu[v_h \leftarrow hi][v_l \leftarrow lo]) \in zTr$ where $hi = ([\![e]\!]_{\mathbb{Z}}(\mu) - lo) \div 2^n$ and $lo = [\![e]\!]_{\mathbb{Z}}(\mu) \bmod 2^n$. Intuitively, $(\mu, s, \mu') \in zTr$ denotes that the state μ transits to the state μ' after executing the statement s.

$$
\begin{array}{rcl}
zStmt & ::= & Var \leftarrow zExpr \mid [Var, Var] \leftarrow \mathrm{Split}(zExpr, \mathbb{N}) \\
zProg & ::= & \epsilon \mid zStmt; zProg
\end{array}
$$

A program is a sequence of statements. Again, we denote the empty program by ϵ. The semantics of a program is defined by the relation $zTr^* \subseteq zSt \times zProg \times zSt$ where $(\mu, \epsilon, \mu) \in zTr^*$ and $(\mu, s; p, \mu'') \in zTr^*$ if there is a μ' with $(\mu, s, \mu') \in zTr$ and $(\mu', p, \mu'') \in zTr^*$. We write $\mu \overset{p}{\Longrightarrow} \mu'$ when $(\mu, p, \mu') \in zTr^*$.

The predicates $zPred$ in zCryptoLine share the same syntax as the algebraic predicates in bvCryptoLine but are evaluated on zSt

$$
\begin{array}{lll}
r_0^1 \leftarrow x_0^0; & r_0^2 \leftarrow r_0^1 + 4503599627370458; & r_0^3 \leftarrow r_0^2 - y_0^0; \\
r_1^1 \leftarrow x_1^0; & r_1^2 \leftarrow r_1^1 + 4503599627370494; & r_1^3 \leftarrow r_1^2 - y_1^0; \\
r_2^1 \leftarrow x_2^0; & r_2^2 \leftarrow r_2^1 + 4503599627370494; & r_2^3 \leftarrow r_2^2 - y_2^0; \\
r_3^1 \leftarrow x_3^0; & r_3^2 \leftarrow r_3^1 + 4503599627370494; & r_3^3 \leftarrow r_3^2 - y_3^0; \\
r_4^1 \leftarrow x_4^0; & r_4^2 \leftarrow r_4^1 + 4503599627370494; & r_4^3 \leftarrow r_4^2 - y_4^0;
\end{array}
$$

Figure 5: Subtraction BV2ZProg(bSubSSA)

rather than on *bSt*.

$$
\begin{array}{rcl}
zPred & ::= & \top \mid zExpr = zExpr \mid zExpr \equiv zExpr \bmod zExpr \\
 & & \mid zPred \wedge zPred
\end{array}
$$

For $\mu \in zSt$ and $q \in bPred_a$, write $\mathbb{Z} \models q[\mu]$ if q evaluates to tt using the evaluation function $[\![e]\!]_{\mathbb{Z}}(\mu)$ for every subexpression e in q. Given $q, q' \in zPred$ and $p \in zProg$, $(\!|q|\!)\ p\ (\!|q'|\!)$ is valid (written $\models (\!|q|\!)\ p\ (\!|q'|\!)$) if for every $\mu, \mu' \in zSt$, $\mathbb{Z} \models q[\mu]$ and $\mu \xRightarrow{p} \mu'$ imply $\mathbb{Z} \models q'[\mu']$.

Now we are ready to describe the transformation from an algebraic specification in bvCryptoLine to a specification in zCryptoLine. Given $\nu \in bSt$ and $\mu \in zSt$, write $\nu \simeq \mu$ when $|\nu(v)| = \mu(v)$ for all variable $v \in Var$. For algebraic expressions, since *zExpr* subsumes $bExp_a$, we can easily define a function BV2ZExpr that converts an algebraic expression $e_a \in bExp_a$ to *zExpr* such that for every $\nu \in bSt$ and $\mu \in zSt$ with $\nu \simeq \mu$, $[\![e_a]\!]_{\mathbb{Z}}(\nu) = [\![\text{BV2ZExpr}(e_a)]\!]_{\mathbb{Z}}(\mu)$. Similarly, we can define a function BV2ZPred such that for every $q_a \in bPred_a$, $\nu \in bSt$, and $\mu \in zSt$ with $\nu \simeq \mu$, $\mathbb{V}^w \models q_a[\nu]$ if and only if $\mathbb{Z} \models \text{BV2ZPred}(q_a)[\mu]$. Atoms are translated by the function BV2ZAtom.

$$
\text{BV2ZAtom}(a) = \begin{cases} v & \text{if } a \text{ is a variable } v \\ |b| & \text{if } a \text{ is a bit-vector } b \end{cases}
$$

Let $\tilde{a}$ denote BV2ZAtom(a) for $a \in bAtom$. The function BV2ZStmt (Algorithm 5) is defined to transform a statement in bvCryptoLine to a statement in zCryptoLine. Define a function BV2ZProg recursively such that BV2ZProg$(\epsilon) \triangleq \epsilon$ and BV2ZProg$(s;p) \triangleq$ BV2ZStmt(s); BV2ZProg(p). With these translation functions, the following soundness theorem holds.

Theorem 4.3. *For every $q_a, q'_a \in bPred_a$, $q_r, q'_r \in bPred_r$, and $p \in bProg$, $\models (\!|q_a \mid q_r|\!)\ p\ (\!|q'_a \mid q'_r|\!)$ if all the following conditions hold.*

- C1 $\mathbb{V}^w \models q_r[\nu]$ *implies* ProgSafe$(p, \nu) = tt$ *for all* $\nu \in bSt$.
- C2 $\models (\!|q_r|\!)\ p\ (\!|q'_r|\!)$.
- C3 $\models (\!|BV2ZPred(q_a)|\!)\ BV2ZProg(p)\ (\!|BV2ZPred(q'_a)|\!)$.

As conditions C1 and C2 involve only bit-vector operations, both conditions can be verified by translations to the QF_BV fragment (quantifier-free formulas over the theory of fixed-size bit-vectors) of SMT (Section 5.1). Condition C3 is verified by a transformation to polynomial equation entailment (Section 5.2). Note that the inverse implication of Theorem 4.3 does not hold because for example, proving $\models (\!|q_r|\!)\ p\ (\!|q'_r|\!)$ may require that q_a holds initally but we do not consider any algebraic predicates in verifying range specifications.

The function BV2ZProg preserves well-formedness and static single assignment form. This is showed by the following lemma.

Lemma 4.4. *Given a well-formed program $p \in bProg$ in static single assignment form, BV2ZProg$(p) \in zProg$ is well-formed and in static single assignment form.*

Algorithm 5 Transformation from *bStmt* to *zStmt* (w is the assumed wordsize)

```
function BV2ZStmt(s)
    match s with
        case v ← a: return v ← ã
        case v ← a1 + a2: return v ← ã1 + ã2
        case c v ← a1 + a2:
            return [c, v] ← Split(ã1 + ã2, w)
        case v ← a1 + a2 + y: return v ← ã1 + ã2 + y
        case c v ← a1 + a2 + y:
            return [c, v] ← Split(ã1 + ã2 + y, w)
        case v ← a1 − a2: return v ← ã1 − ã2
        case v ← a1 × a2: return v ← ã1 × ã2
        case vh vl ← a1 × a2:
            return [vh, vl] ← Split(ã1 × ã2, w)
        case v ← a ≪ n: return v ← ã × Pow(2, |n|)
        case vh vl ← a@n: return [vh, vl] ← Split(ã, |n|)
        case vh vl ← (a1.a2) ≪ n:
            return [vh, vl] ← Split(ã1×Pow(2, w)+ã2, w−|n|)
end function
```

Figure 5 shows the result of transforming the subtraction program bSubSSA to zCryptoLine.

4.3 Modular Polynomial Equation Entailment

The last step transforms any algebraic program specification in zCryptoLine to the modular polynomial equation entailment problem. For $q \in zPred$, we write $q(\vec{x})$ to signify the free variables $\vec{x}$ occurring in q. Given $q(\vec{x}), q'(\vec{x}) \in zPred$, the *modular polynomial equation entailment* problem decides whether $q(\vec{x}) \implies q'(\vec{x})$ holds when $\vec{x}$ are substituted for arbitrary integers. That is, we want to check if for every valuation $\mu \in zSt$, $q(\vec{x})$ evaluates to tt implies $q'(\vec{x})$ evaluates to tt after each variable x is replaced by $\mu(x)$. We write $\mathbb{Z} \models \forall \vec{x}.q(\vec{x}) \implies q'(\vec{x})$ if it is indeed the case.

Programs in static single assignment form can easily be transformed to conjunctions of polynomial equations. The function StmtToPolyEq (Algorithm 6) translates (1) an assignment statement to a polynomial equation with a variable on the left hand side and (2) a Split statement to an equation with a linear expression of the assigned variables on the left hand side. A program in static single assignment form is then transformed to the conjunction of polynomial equations corresponding to its statements by the function ProgToPolyEq, which is recursively defined such that ProgToPolyEq$(\epsilon) \triangleq \top$ and ProgToPolyEq$(s;p) \triangleq$ StmtToPolyEq$(s) \wedge$ ProgToPolyEq(p).

$$
\top \wedge \left(\begin{array}{ccccccccccccccc}
r_0^1 & = & x_0^0 & \wedge & r_0^2 & = & r_0^1 + 4503599627370458 & \wedge & r_0^3 & = & r_0^2 - y_0^0 & \wedge \\
r_1^1 & = & x_1^0 & \wedge & r_1^2 & = & r_1^1 + 4503599627370494 & \wedge & r_1^3 & = & r_1^2 - y_1^0 & \wedge \\
r_2^1 & = & x_2^0 & \wedge & r_2^2 & = & r_2^1 + 4503599627370494 & \wedge & r_2^3 & = & r_2^2 - y_2^0 & \wedge \\
r_3^1 & = & x_3^0 & \wedge & r_3^2 & = & r_3^1 + 4503599627370494 & \wedge & r_3^3 & = & r_3^2 - y_3^0 & \wedge \\
r_4^1 & = & x_4^0 & \wedge & r_4^2 & = & r_4^1 + 4503599627370494 & \wedge & r_4^3 & = & r_4^2 - y_4^0 &
\end{array}\right) \Longrightarrow
$$
$$
\mathit{radix51}_{\mathbb{Z}}(x_4^0, x_3^0, x_2^0, x_1^0, x_0^0) - \mathit{radix51}_{\mathbb{Z}}(y_4^0, y_3^0, y_2^0, y_1^0, y_0^0) \equiv \mathit{radix51}_{\mathbb{Z}}(r_4^3, r_3^3, r_2^3, r_1^3, r_0^3) \bmod \varrho
$$

Figure 6: Modular Polynomial Equation Entailment for BV2ZProg(bSubSSA)

Algorithm 6 Polynomial Equation Transformation for Statements

---

function StmtToPolyEQ(s)
  match s **with**
    case $v \leftarrow e$: **return** $v = e$
    case $[v_h, v_l] \leftarrow \mathsf{Split}(e, n)$:
      return $v_l + \mathsf{Pow}(2, n) \times v_h = e$
end function

The functions StmtToPolyEq and ProgToPolyEq are specified straightforwardly in Gallina. We use the proof assistant Coq to prove properties about the functions. Note that ProgToPolyEQ(p) $\in$ *zPred* for every $p \in$ *zProg*. The following theorem shows that any behavior of the program p is a solution to the system of polynomial equations ProgToPolyEQ(p). In other words, ProgToPolyEQ(p) gives an abstraction of the program p.

Theorem 4.5. *Let $p \in$ zProg be a well-formed program in static single assignment form. For every $\mu, \mu' \in$ zSt with $\mu \stackrel{p}{\Longrightarrow} \mu'$, we have $\mathbb{Z} \models$ ProgToPolyEQ(p)$[\mu']$.*

Definition 4.6 gives the modular polynomial equation entailment problem corresponding to an algebraic program specification.

Definition 4.6. For $q, q' \in$ *zPred* and $p \in$ *zProg* in static single assignment form, define

$$\Pi(\llparenthesis q \rrparenthesis\ p\ \llparenthesis q' \rrparenthesis) \quad \triangleq \quad q(\vec{x}) \wedge \varphi(\vec{x}) \implies q'(\vec{x})$$

where $\varphi(\vec{x}) =$ ProgToPolyEQ(p).

Example. The modular polynomial equation entailment problem corresponding to the algebraic specification of subtraction is shown in Figure 6. The problem has 15 polynomial equality constraints with 25 variables. Define $\mathit{radix51}_{\mathbb{Z}}(x_4, x_3, x_2, x_1, x_0) \triangleq \mathsf{Pow}(2, 51 \times_{\mathbb{Z}} 4) \times x_4 + \mathsf{Pow}(2, 51 \times_{\mathbb{Z}} 3) \times x_3 + \mathsf{Pow}(2, 51 \times_{\mathbb{Z}} 2) \times x_2 + \mathsf{Pow}(2, 51 \times_{\mathbb{Z}} 1) \times x_1 + \mathsf{Pow}(2, 51 \times_{\mathbb{Z}} 0) \times x_0$ for $x_0, x_1, x_2, x_3, x_4 \in$ *Var*. We want to know if $\mathit{radix51}_{\mathbb{Z}}(r_4^3, r_3^3, r_2^3, r_1^3, r_0^3)$ is the difference between $\mathit{radix51}_{\mathbb{Z}}(x_4^0, x_3^0, x_2^0, x_1^0, x_0^0)$ and $\mathit{radix51}_{\mathbb{Z}}(y_4^0, y_3^0, y_2^0, y_1^0, y_0^0)$ in $\mathbb{GF}(\varrho)$ under the constraints.

The soundness of ProgToPolyEq is certified in Coq (Theorem 4.7). It is not complete because in the transformation of the statement $[v_h, v_l] \leftarrow \mathsf{Split}(e, n)$, the polynomial equation $v_l + \mathsf{Pow}(2, n) \times v_h = e$ does not guarantee that v_l exactly represents the lower n bits of e.

Theorem 4.7. *Let $q, q' \in$ zPred be predicates, and $p \in$ zProg a well-formed program in static single assignment form. If $\mathbb{Z} \models \forall \vec{x}.\Pi(\llparenthesis q \rrparenthesis\ p\ \llparenthesis q' \rrparenthesis)$, then $\models \llparenthesis q \rrparenthesis\ p\ \llparenthesis q' \rrparenthesis$.*

$$
\begin{array}{lll}
 & \models \llparenthesis q_a \mid q_r \rrparenthesis\ p\ \llparenthesis q'_a \mid q'_r \rrparenthesis & \\
\Leftrightarrow & \models \llparenthesis \hat{q}_a \mid \hat{q}_r \rrparenthesis\ \hat{p}\ \llparenthesis \hat{q}'_a \mid \hat{q}'_r \rrparenthesis & \text{(Theorem 4.2)} \\
 & \text{where } \langle \hat{\theta}, \hat{p} \rangle = \mathrm{SSAProg}(\theta_0, p), & \\
 & \hat{q}_a = \mathrm{SSAPredA}(\theta_0, q_a), & \\
 & \hat{q}_r = \mathrm{SSAPredR}(\theta_0, a_r), & \\
 & \hat{q}'_a = \mathrm{SSAPredA}(\hat{\theta}, q'_a), \text{ and} & \\
 & \hat{q}'_r = \mathrm{SSAPredR}(\hat{\theta}, q'_r) & \\
\Leftarrow & \mathbb{V}^w \models \hat{q}_r[v] \text{ implies } \mathrm{ProgSafe}(\hat{p}) = tt \text{ for all } v \in bSt, & \\
 & \models \llparenthesis \hat{q}_r \rrparenthesis\ \hat{p}\ \llparenthesis \hat{q}'_r \rrparenthesis, \text{ and} & \\
 & \models \llparenthesis \tilde{q}_a \rrparenthesis\ \tilde{p}\ \llparenthesis \tilde{q}'_a \rrparenthesis & \text{(Theorem 4.3)} \\
 & \text{where } \tilde{p} = \mathrm{BV2ZProg}(\hat{p}), & \\
 & \tilde{q}_a = \mathrm{BV2ZPred}(\hat{q}_a), \text{ and} & \\
 & \tilde{q}'_a = \mathrm{BV2ZPred}(\hat{q}'_a) & \\
\Leftarrow & \mathbb{V}^w \models \hat{q}_r[v] \text{ implies } \mathrm{ProgSafe}(\hat{p}) = tt \text{ for all } v \in bSt, & \\
 & \models \llparenthesis \hat{q}_r \rrparenthesis\ \hat{p}\ \llparenthesis \hat{q}'_r \rrparenthesis, \text{ and} & \\
 & \mathbb{Z} \models \forall \vec{x}.\Pi(\llparenthesis \tilde{q}_a \rrparenthesis\ \tilde{p}\ \llparenthesis \tilde{q}'_a \rrparenthesis) & \text{(Theorem 4.7)}
\end{array}
$$

Figure 7: Summary of Translations

Summary of Translation. Consider any algebraic predicates q_a, $q'_a \in bPred_a$, range predicates q_r, $q'_r \in bPred_r$, and well-formed program $p \in bProg$. Let $\theta_0(v) = 0$ for every $v \in Var$. By Theorem 4.2, 4.3, and 4.7, we have a summary of translation in Figure 7. Observe that $\tilde{p}$ in Figure 7 is well-formed and in static single assignment form (Lemma 4.1 and 4.4). Theorem 4.7 is applicable in the last deduction. After the translations, a safety check, a range problem, and an instance of the modular polynomial equation entailment problem are obtained from the given specification of a well-formed program in bvCryptoLine. To verify mathematical constructs against their specifications, we will solve all the three problems in the next section.

5 VERIFICATION OF SPECIFICATIONS

We show how to solve a range problem, check if a program is safe, and solve modular polynomial equation entailment problem in this section. The first two problems are reduced to QF_BV formulas and solved by an SMT solver. The last problem is reduced to an ideal membership problem and solve by the computer algebra system Singular.

5.1 Solving Range and Overflow Checks

First define the syntax of a fragment of QF_BV with function names taken from the standard format SMTLIB2. In this fragment, a variable always represents a bit-vector of width w (the assumed word-size). Let *qExp* and *qPred* respectively denote the expressions and the predicates in QF_BV. An expression $e \in qExp$ can be a constant $\mathsf{bvconst}(n, b)$, a variiable $v \in Var$, an addition $\mathsf{bvadd}(e_1, e_2)$, a subtraction $\mathsf{bvsub}(e_1, e_2)$, a multiplication $\mathsf{bvmul}(e_1, e_2)$, a concatenation $\mathsf{concat}(e_1, e_2)$, a zero extension $\mathsf{zero\_extend}(e', i)$, a left-shifting $\mathsf{bvshl}(e_1, e_2)$, a logical right-shifting $\mathsf{bvlshr}(e_1, e_2)$, or an extraction $\mathsf{bvextract}(e', i, j)$ where $n, i, j \in \mathbb{N}$, $b \in \mathbb{V}^n$, and $e_1, e_2, e' \in qExp$. A predicate $q \in qPred$ can be $\top$, an equality $e_1 = e_2$, a less-than $\mathsf{bvult}(e_1, e_2)$, a less-then-or-equal $\mathsf{bvule}(e_1, e_2)$, a negation $\neg q'$, a conjunction $q_1 \wedge q_2$, or a disjunction $q_1 \vee q_2$ where $e_1, e_2 \in qExp$ and $q', q_1, q_2 \in qPred$. An implication $q_1 \Rightarrow q_2$ is defined as $\neg q_1 \vee q_2$.

Based on the basic expressions, we define two shorthands for extracting the higher bits and the lower bits of an expression.

$$\begin{array}{rcl}
\mathsf{bvhigh}(e) & \triangleq & \mathsf{bvextract}(e, 2w-1, n) \\
\mathsf{bvlow}(e) & \triangleq & \mathsf{bvextract}(e, w-1, 0)
\end{array}$$

Similar to the bit-vector operations $+_{\mathbb{V}}$, $-_{\mathbb{V}}$, and $\times_{\mathbb{V}}$ extended with zero extension in Section 2, for $\bullet \in \{\mathsf{bvadd}, \mathsf{bvsub}, \mathsf{bvmul}\}$, we define their extended versions $\bullet^{\#}$. For example, $\mathsf{bvadd}^{\#}(e_1, e_2) \triangleq \mathsf{bvadd}(\mathsf{zero\_extend}(e_1, w), \mathsf{zero\_extend}(e_2, w))$ and $\mathsf{bvadd}^{\#}(e_1, e_2, e_3) \triangleq \mathsf{bvadd}(\mathsf{bvadd}(\mathsf{zero\_extend}(e_1, w), \mathsf{zero\_extend}(e_2, w)), \mathsf{zero\_extend}(e_3, w))$.

Let $\max(n, m)$ return the maximal number in n and m. Given an expression $e \in qExp$, $width(e)$ denotes the maximal bit-width of e.

$$\begin{array}{rcl}
width(\mathsf{bvconst}(n, b)) & = & n \\
width(v) & = & w \\
width(\mathsf{bvadd}(e_1, e_2)) & = & max(width(e_1), width(e_2)) \\
width(\mathsf{bvsub}(e_1, e_2)) & = & max(width(e_1), width(e_2)) \\
width(\mathsf{bvmul}(e_1, e_2)) & = & max(width(e_1), width(e_2)) \\
width(\mathsf{concat}(e_1, e_2)) & = & width(e_1) +_{\mathbb{N}} width(e_2) \\
width(\mathsf{zero_extend}(e, i)) & = & width(e) +_{\mathbb{N}} i \\
width(\mathsf{bvshl}(e_1, e_2)) & = & width(e_1) \\
width(\mathsf{bvlshr}(e_1, e_2)) & = & width(e_1) \\
width(\mathsf{bvextract}(e, i, j)) & = & i -_{\mathbb{N}} j +_{\mathbb{N}} 1
\end{array}$$

The expression e is called *well-formed* if e is (1) a constant, a variable, a concatenation, a zero extension, a left-shifting, or a logical right-shifting, (2) an addition $\mathsf{bvadd}(e_1, e_2)$, a subtraction $\mathsf{bvsub}(e_1, e_2)$, or a multiplication $\mathsf{bvmul}(e_1, e_2)$ with $width(e_1) = width(e_2)$ and both e_1 and e_2 well-formed, or (3) an extraction $\mathsf{bvextract}(e', i, j)$ with $0 \leq j \leq i < width(e')$ and e' well-formed. A predicate $q \in qPred$ is well-formed if all subexpressions are well-formed.

Let $\nu \in bSt$ be a state. Define the semantic function $[\![e]\!]_{\mathbb{V}}(\nu)$ for well-formed expressions $e \in qExp$. For a predicate $q \in qPred$, we write $\mathbb{V}^w \models q[\nu]$ if q evaluates to tt using the evaluation function $[\![e]\!]_{\mathbb{V}}(\nu)$ for every subexpression e in q, using $<_{\mathbb{V}}$ for bvult, and using $\leq_{\mathbb{V}}$ for bvule.

$$\begin{array}{rcl}
[\![\mathsf{bvconst}(n, b)]\!]_{\mathbb{V}}(\nu) & \triangleq & b \\
[\![v]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![v]\!]_{\mathbb{Z}}(\nu) \\
[\![\mathsf{bvadd}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) +_{\mathbb{V}} [\![e_2]\!]_{\mathbb{V}}(\nu) \\
[\![\mathsf{bvsub}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) -_{\mathbb{V}} [\![e_2]\!]_{\mathbb{V}}(\nu) \\
[\![\mathsf{bvmul}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) \times_{\mathbb{V}} [\![e_2]\!]_{\mathbb{V}}(\nu) \\
[\![\mathsf{concat}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) ._{\mathbb{V}} [\![e_2]\!]_{\mathbb{V}}(\nu) \\
[\![\mathsf{zero_extend}(e, i)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e]\!]_{\mathbb{V}}(\nu) \#_{\mathbb{V}} i \\
[\![\mathsf{bvshl}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) \ll_{\mathbb{V}} |[\![e_2]\!]_{\mathbb{V}}(\nu)| \\
[\![\mathsf{bvlshr}(e_1, e_2)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e_1]\!]_{\mathbb{V}}(\nu) \gg_{\mathbb{V}} |[\![e_2]\!]_{\mathbb{V}}(\nu)| \\
[\![\mathsf{bvextract}(e, i, j)]\!]_{\mathbb{V}}(\nu) & \triangleq & [\![e]\!]_{\mathbb{V}}(\nu)[i, j]
\end{array}$$

Let $q_r, q_r' \in bPred_r$ be two range predicates and $p \in bProg$ a well-formed program in static single assignment form. Both an safety check ($\mathbb{V}^w \models q_r[\nu]$ implies $\textsc{ProgSafe}(p, \nu) = tt$ for all $\nu \in bSt$) and a range problem ($\models (\!|q_r|\!)\ p\ (\!|q_r'|\!)$) involve only bit-vector operations and can be modeled by QF_BV expressions. To show that, we first define functions to transform the program p, the predicates q_r and q_r', and the safety check to QF_BV formulas.

Define $\overline{a}$ as v when the atom a is a variable v and otherwise $\mathsf{bvconst}(w, b)$ when a is a constant b. The function $\textsc{StmtQFBV}$ (Algorithm 7) transforms a statement in *bStmt* to a QF_BV formula. Recursively define the function $\textsc{ProgQFBV}$ for programs in *bProg* such that $\textsc{ProgQFBV}(\epsilon) \triangleq \top$ and $\textsc{ProgQFBV}(s; p) \triangleq \textsc{StmtQFBV}(s) \wedge \textsc{ProgQFBV}(p)$. Note that the formulas returned by $\textsc{StmtQFBV}$ and $\textsc{ProgQFBV}$ are well-formed QF_BV formulas. The following theorem states that $\textsc{ProgQFBV}(p)$ gives an abstraction of the program p.

Theorem 5.1. *Let $p \in bProg$ be a well-formed program in static single assignment form. Then, for all $\nu, \nu' \in bSt$, $\nu \overset{p}{\Longrightarrow} \nu'$ implies $\mathbb{V}^w \models \textsc{ProgQFBV}(p)[\nu']$.*

Algorithm 7 Transformation from *bStmt* to *qPred*

```
function StmtQFBV(s)
    match s with
        case v ← a: return v = ā
        case v ← a₁ + a₂: return v = bvadd(ā₁, ā₂)
        case c v ← a₁ + a₂:
            r ← bvadd#(ā₁, ā₂)
            return c = bvhigh(r) ∧ v = bvlow(r)
        case v ← a₁ + a₂ + y:
            return v = bvadd(bvadd(ā₁, ā₂), y)
        case c v ← a₁ + a₂ + y:
            r ← bvadd#(bvadd#(ā₁, ā₂), y)
            return c = bvhigh(r) ∧ v = bvlow(r)
        case v ← a₁ − a₂: return v = bvsub(ā₁, ā₂)
        case v ← a₁ × a₂: return v = bvmul(ā₁, ā₂)
        case v_h v_l ← a₁ × a₂:
            r ← bvmul#(ā₁, ā₂)
            return v_h = bvhigh(r) ∧ v_l = bvlow(r)
        case v ← a ≪ n: return v = bvshl(ā, bvconst(w, n))
        case v_h v_l ← a@n:
            m_h ← bvconst(w, n)
            m_l ← bvconst(w, bv^w(w − |n|))
            return v_h = bvlshr(ā, m_h)∧
                    v_l = bvlshr(bvshl(ā, m_l), m_l)
        case v_h v_l ← (a₁.a₂) ≪ n:
            m_n ← bvconst(w, n)
            r ← bvshl(concat(ā₁, ā₂), m_n)
            return v_h = bvhigh(r)∧
                    v_l = bvlshr(bvlow(r), m_n)
end function
```

For the transformation from range predicates to QF_BV formulas, recursively define a function $\textsc{PredrQFBV}$ such that $\textsc{PredrQFBV}(\top)$

$\triangleq \top$, PredrQFBV($a_1 < a_2$) $\triangleq$ bvult($\overline{a_1}, \overline{a_2}$), PredrQFBV($a_1 \leq a_2$) $\triangleq$ bvule($\overline{a_1}, \overline{a_2}$), and PredrQFBV($p_1 \land p_2$) $\triangleq$ PredrQFBV(p_1) $\land$ PredrQFBV(p_2). We have the following theorem for the transformation of range predicates.

Theorem 5.2. *Let $q \in bPred_r$ be a range predicate. Then, for all $v \in bSt$, $\mathbb{V}^w \models q[v]$ if and only if $\mathbb{V}^w \models$ PredrQFBV$(q)[v]$.*

Define a function StmtSafeQFBV (Algorithm 8) which transforms safety checks for statements to QF_BV. Recursively define a function ProgSafeQFBV such that ProgSafeQFBV(ϵ) $\triangleq \top$ and ProgSafeQFBV($s;p$) $\triangleq$ StmtSafeQFBV(s) $\land$ ProgSafeQFBV(p). The following theorem states the soundness of our translation from range problems and safety checks to QF_BV.

Theorem 5.3. *Given two range predicates $q_r, q'_r \in bPred_r$ and a well-formed program $p \in bProg$ in static single assignment form,*

- *$\mathbb{V}^w \models q_r[v]$ implies ProgSafe$(p, v) = tt$ for all $v \in bSt$ if, (PredrQFBV$(q_r) \land$ ProgQFBV$(p)) \Rightarrow$ ProgSafeQFBV(p) is valid, and*
- *$\models (\!| q_r |\!)\ p\ (\!| q'_r |\!)$ if the QF_BV formula PredrQFBV$(q_r) \land$ ProgQFBV$(p) \Rightarrow$ PredrQFBV(q'_r) is valid.*

Algorithm 8 Transformation from Safety Checks to QF_BV

function StmtSafeQFBV(s)
  $o \leftarrow$ bvconst(w, $bv^w(0)$)
  match s **with**
    case $v \leftarrow a$: **return** $\top$
    case $v \leftarrow a_1 + a_2$: **return** bvhigh(bvadd$^\#$($\overline{a_1}, \overline{a_2}$)) $= o$
    case $c\ v \leftarrow a_1 + a_2$: **return** $\top$
    case $v \leftarrow a_1 + a_2 + y$:
      return bvhigh(bvadd$^\#$($\overline{a_1}, \overline{a_2}$)) $= o \land$
        bvhigh(bvadd$^\#$(bvadd$^\#$($\overline{a_1}, \overline{a_2}$)), y) $= o$
    case $c\ v \leftarrow a_1 + a_2 + y$: **return** $\top$
    case $v \leftarrow a_1 - a_2$: **return** bvhigh(bvsub$^\#$($\overline{a_1}, \overline{a_2}$)) $= o$
    case $v \leftarrow a_1 \times a_2$: **return** bvhigh(bvmul$^\#$($\overline{a_1}, \overline{a_2}$)) $= o$
    case $v_h\ v_l \leftarrow a_1 \times a_2$: **return** $\top$
    case $v \leftarrow a \ll n$:
      $one \leftarrow$ bvconst(w, $bv^w(1)$)
      $m \leftarrow$ bvconst(w, $bv^w(w - |n|)$)
      return bvult($\overline{a}$, bvshl(one, m))
    case $v_h\ v_l \leftarrow a@n$: **return** $\top$
    case $v_h\ v_l \leftarrow (a_1.a_2) \ll n$:
      $one \leftarrow$ bvconst(w, $bv^w(1)$)
      $m \leftarrow$ bvconst(w, $bv^w(w - |n|)$)
      return bvult($\overline{a_1}$, bvshl(one, m))$\land$
        bvule(bvconst(w, n), bvconst(w, $bv^w(w)$))
end function

5.2 Solving Modular Polynomial Equation Entailment Problem

To solve a modular polynomial equation entailment problem $\mathbb{Z} \models \forall \vec{x}.\Pi((\!|\tilde{q}_a|\!)\ \tilde{p}\ (\!|\tilde{q}'_a|\!))$, it remains to show

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \land \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv f'_j(\vec{x}) \bmod n_j(\vec{x})$$
$$\implies \bigwedge_{k \in [K]} g_k(\vec{x}) = g'_k(\vec{x}) \land \bigwedge_{l \in [L]} h_l(\vec{x}) \equiv h'_l(\vec{x}) \bmod m_l(\vec{x})$$

where $e_i(\vec{x})$, $e'_i(\vec{x})$, $f_j(\vec{x})$, $f'_j(\vec{x})$, $n_j(\vec{x})$, $g_k(\vec{x})$, $g'_k(\vec{x})$, $h_l(\vec{x})$, $h'_l(\vec{x})$, $m_l(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for $i \in [I]$, $j \in [J]$, $k \in [K]$, and $l \in [L]$. Since the consequence is a conjunction of (modular) equations, it suffices to prove one conjunct at a time. That is, we aim to show

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \land \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv f'_j(\vec{x}) \bmod n_j(\vec{x})$$
$$\implies g(\vec{x}) = g'(\vec{x}); \text{ or}$$

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \land \bigwedge_{j \in [J]} f_j(\vec{x}) \equiv f'_j(\vec{x}) \bmod n_j(\vec{x})$$
$$\implies h(\vec{x}) \equiv h'(\vec{x}) \bmod m(\vec{x})$$

where $e_i(\vec{x}), e'_i(\vec{x}), f_j(\vec{x}), f'_j(\vec{x}), n_j(\vec{x}), g(\vec{x}), g'(\vec{x}), h(\vec{x}), h'(\vec{x}), m(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for $i \in [I], j \in [J]$.

It is not hard to rewrite modular polynomial equations in antecedents of the above implications. For instance, the first implication is equivalent to

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \land \bigwedge_{j \in [J]} [\exists d_j. f_j(\vec{x}) = f'_j(\vec{x}) + d_j \cdot n_j(\vec{x})]$$
$$\implies g(\vec{x}) = g'(\vec{x}),$$

which in turn is equivalent to

$$\mathbb{Z} \models \forall \vec{x} \forall \vec{d}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \land \bigwedge_{j \in [J]} f_j(\vec{x}) = f'_j(\vec{x}) + d_j \cdot n_j(\vec{x})$$
$$\implies g(\vec{x}) = g'(\vec{x}).$$

It hence suffices to consider the following *polynomial equation entailment* problem:

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \implies g(\vec{x}) = g'(\vec{x}); \text{ or} \tag{1}$$

$$\mathbb{Z} \models \forall \vec{x}. \bigwedge_{i \in [I]} e_i(\vec{x}) = e'_i(\vec{x}) \implies h(\vec{x}) \equiv h'(\vec{x}) \bmod m(\vec{x}) \tag{2}$$

where $e_i(\vec{x})$, $e'_i(\vec{x})$, $g(\vec{x})$, $g'(\vec{x})$, $h(\vec{x})$, $h'(\vec{x})$, $m(\vec{x}) \in \mathbb{Z}[\vec{x}]$ for $i \in [I]$ [22].

We solve the polynomial equation entailment problems (1) and (2) via the ideal membership problem [11, 22]. For (1), consider the ideal $I = \langle e_i(\vec{x}) - e'_i(\vec{x}) \rangle_{i \in [I]}$. Suppose $g(\vec{x}) - g'(\vec{x}) \in I$. Then there are $u_i(\vec{x}) \in \mathbb{Z}[\vec{x}]$ (called *coefficients*) such that

$$g(\vec{x}) - g'(\vec{x}) = \sum_{i \in [I]} u_i(\vec{x})[e_i(\vec{x}) - e'_i(\vec{x})]. \tag{3}$$

Hence $g(\vec{x}) - g'(\vec{x}) = 0$ follows from the polynomial equations $e_i(\vec{x}) = e'_i(\vec{x})$ for $i \in [I]$. Similarly, it suffices to check if $h(\vec{x})$ –

$h'(\vec{x}) \in \langle m(\vec{x}), e_i(\vec{x}) - e'_i(\vec{x})\rangle_{i\in[I]}$ for (2). If so, there are $u, u_i(\vec{x}) \in \mathbb{Z}[\vec{x}]$ such that

$$h(\vec{x}) - h'(\vec{x}) = u(\vec{x}) \cdot m(\vec{x}) + \sum_{i\in[I]} u_i(\vec{x})[e_i(\vec{x}) - e'_i(\vec{x})]. \quad (4)$$

Thus $h(\vec{x}) \equiv h'(\vec{x}) \bmod m(\vec{x})$ as required. The reduction to the ideal membership problem however is incomplete. Consider $\mathbb{Z} \models \forall x.x^2 + x \equiv 0 \bmod 2$ but $x^2 + x \notin \langle 2\rangle$ [22].

Two Coq tactics are available to find formal proofs for the polynomial equation entailment problems [28, 29]. The tactic nsatz proves the entailment problem of the form in (1); the tactic gbarith proves the form in (2). The ideal membership problem can be solved by finding a Gröbner basis for the ideal [15]. Both tactics solve the polynomial equation entailment problem by computing Gröbner bases for induced ideals. Finding Gröbner bases for ideals however is NP-hard because it allows us to solve a system of equations over the Boolean field [20]. Low-level mathematical constructs can have hundreds of polynomial equations in (1) or (2). Both Coq tactics fail to solve such problems in a reasonable amount of time.

We develop two heuristics to solve the polynomial equation entailment problem more effectively. Note that the polynomial equations generated by Algorithm 6 are of the forms: $x = e$ (from assignment statements) or $x + 2^c y = e$ (from Split statements). Such polynomial equations can safely be removed after every occurrences of x are replaced with e or $e - 2^c y$ respectively. The number of generators of the induced ideal is hence reduced. We define a Coq tactic to simplify polynomial equation entailment problems by rewriting variables and then removing polynomial equations.

To further improve scalability, we use the computer algebra system Singular to solve the ideal membership problem [21]. Our tactic submits the membership problem to Singular and obtains coefficients from the computer algebra system. Since algorithms used in Singular might be implemented incorrectly, our Coq tactic then certifies the coefficients by checking the equation (3) or (4) to ensure the polynomial equation entailment problem is correctly solved. Soundness of our technique therefore does not rely on the external solver Singular.

6 EVALUATION

We evaluate our techniques in real-world low-level mathematical constructs in X25519. In elliptic curve cryptography, arithmetic computation over large finite fields is required. For instance, Curve25519 defined by $y^2 = x^3 + 486662x^2 + x$ is over the Galois field $\mathbb{K} = \mathbb{GF}(\varrho)$ with $\varrho = 2^{255} - 19$. To make the field explicit, we rewrite its definition as:

$$y \cdot_{\mathbb{K}} y =_{\mathbb{K}} x \cdot_{\mathbb{K}} x \cdot_{\mathbb{K}} x +_{\mathbb{K}} 486662 \cdot_{\mathbb{K}} x \cdot_{\mathbb{K}} x +_{\mathbb{K}} x. \quad (5)$$

Since arithmetic computation is over $\mathbb{K}$ whose elements can be represented by 255-bit numbers, any pair (x, y) satisfying (5) (called a *point* on the curve) can be represented by a pair of 255-bit numbers. It can be shown that points on Curve25519 with the point at infinity as the unit (denoted $0_{\mathbb{G}}$) form a commutative group $\mathbb{G} = (G, +_{\mathbb{G}}, 0_{\mathbb{G}})$ with $G = \{(x, y) : x, y \text{ satisfying (5)}\}$. Let $P_0 = (x_0, y_0), P_1 = (x_1, y_1) \in G$. We have $-P_0 = (x_0, -y_0)$ and $P_0 +_{\mathbb{G}} P_1 = (x, y)$ where

$$\begin{aligned} m &= (y_1 -_{\mathbb{K}} y_0) \div_{\mathbb{K}} (x_1 -_{\mathbb{K}} x_0) \\ x &= m \cdot_{\mathbb{K}} m -_{\mathbb{K}} 486662 -_{\mathbb{K}} x_0 -_{\mathbb{K}} x_1 \\ y &= (2 \cdot_{\mathbb{K}} x_0 +_{\mathbb{K}} x_1 +_{\mathbb{K}} 486662) \cdot_{\mathbb{K}} m -_{\mathbb{K}} m \cdot_{\mathbb{K}} m \cdot_{\mathbb{K}} m -_{\mathbb{K}} y_0 \end{aligned} \quad (6)$$

when $P_0 \neq \pm P_1$. Other cases ($P_0 = \pm P_1$) are defined similarly [15]. $\mathbb{G}$ and similar elliptic curve groups are the main objects in elliptic curve cryptography. It is essential to implement the commutative binary operation $+_{\mathbb{G}}$ very efficiently in practice.

6.1 Arithmetic Computation over $\mathbb{GF}(2^{255} - 19)$

The operation $+_{\mathbb{G}}$ is defined by arithmetic computation over $\mathbb{K}$. Mathematical constructs for arithmetic over $\mathbb{K}$ are hence necessary. Recall that an element in $\mathbb{K}$ is represented by a 256-bit number. Arithmetic computation for 255-bit integers however is not yet available in commodity computing devices as of the year 2017; it has to be carried out by limbs where a *limb* is a 32- or 64-bit number depending on the underlying computer architectures. Figure 3 is such an implementation of subtraction for the AMD64 architecture.

Multiplication is another interesting but much more complicated computation. The naïve implementation for 255-bit multiplication would compute a 510-bit product and then find the corresponding 255-bit representation by division. An efficient implementation for 255-bit multiplication avoids division by performing modulo operations aggressively. For instance, an intermediate result of the form $c \cdot_{\mathbb{K}} 2^{255}$ is immediately replaced by $c \cdot_{\mathbb{K}} 19$ since $2^{255} =_{\mathbb{K}} 19$ in $\mathbb{GF}(\varrho)$. This is indeed how the most efficient multiplication for the AMD64 architecture is implemented (Appendix A.1) [9, 10].

In our experiment, we took real-world efficient and secure low-level implementations of arithmetic computation over $\mathbb{GF}(\varrho)$ from [9, 10], manually translated source codes to our domain specific language, specified their algebraic and range properties, and performed certified verification with our technique. Table 1 summarizes the results without and with applying the two heuristics in Section 5.2. The column "safe" shows the time used by the SMT solver Boolector to verify if the input program is safe. The column "range" shows the time used by Boolector to verify the range specification of the input program. The columns "algebraic" show the time used by Singular to verify the algebraic specification of the input program. The columns "total" show the total verification time including safety check, verification of range and algebraic specifications, rewriting, proof certification, etc. The columns "without heuristics" and "with heuristics" respectively show the time information without and with the two heuristics. The results show that without the two heuristics, multiplication and square cannot be verified because the computation of Gröbner bases was killed by the OS after running for days. With the heuristics, all the implementations can be verified in seconds.

We also tried to verify buggy implementations such as the buggy implementation of multiplication mentioned in [14]. In such cases, our verification tactic in Coq just failed without giving any counterexample. Note that when our tactic fails to verify a program, we cannot conclude that the program is buggy because our approach is sound but not complete.

Table 1: Certified Verification of Arithmetic Operations over $\mathbb{GF}(\varrho)$

	number of lines	time (seconds)						remark
		safe	range	without heuristics		with heuristics		
				algebraic	total	algebraic	total	
addition	10	0.162	0.249	30.545	41.55	0.401	4.14	$a +_{\mathbb{K}} b$
subtraction	15	0.140	0.389	35.646	48.47	0.208	4.93	$a -_{\mathbb{K}} b$
multiplication	144	3.904	41.070	-	-	2.312	81.93	$a \cdot_{\mathbb{K}} b$
multiplication by 121666	26	0.266	0.852	1112.311	1125.41	0.315	7.70	$121666 \cdot_{\mathbb{K}} a$
square	109	3.722	19.905	-	-	1.087	47.44	$a \cdot_{\mathbb{K}} a$

6.2 The Montgomery Ladderstep

Recall that X25519 is based on the Abelian group $\mathbb{G} = (G, +_{\mathbb{G}}, 0_{\mathbb{G}})$ induced by the curve Curve25519. As aforementioned, the binary operation $+_{\mathbb{G}}$ requires another sequence of arithmetic computation over $\mathbb{GF}(\varrho)$. Errors could still be introduced or even implanted in any sequence of computation proclaimed to implement $+_{\mathbb{G}}$. Our next experiment verifies a critical low-level program implementing the group operation [9, 10].

Let $P \in G$ be a point on Curve25519. We write $[n]P$ for the n-fold addition $P +_{\mathbb{G}} \cdots +_{\mathbb{G}} P \in G$ for $n \in \mathbb{N}$. In X25519, we want to compute a *point multiplication*, that is, the point $[n]P$ for given n and P. The standard iterative squaring method computes $[n]P$ by examining each bit of n iteratively. For each iteration, $[2m]P$ is computed from $[m]P$ and added with another P when the current bit is 1. Although the method is reasonably efficient, it is not constant-time and hence insecure.

Algorithm 9 Montgomery Ladderstep

function LADDERSTEP($x_1, x_m, z_m, x_{m+1}, z_{m+1}$)

$t_1 \leftarrow x_m +_{\mathbb{K}} z_m$
$t_2 \leftarrow x_m -_{\mathbb{K}} z_m$
$t_7 \leftarrow t_2 \cdot_{\mathbb{K}} t_2$
$t_6 \leftarrow t_1 \cdot_{\mathbb{K}} t_1$
$t_5 \leftarrow t_6 -_{\mathbb{K}} t_7$
$t_3 \leftarrow x_{m+1} +_{\mathbb{K}} z_{m+1}$
$t_4 \leftarrow x_{m+1} -_{\mathbb{K}} z_{m+1}$
$t_9 \leftarrow t_3 \cdot_{\mathbb{K}} t_2$
$t_8 \leftarrow t_4 \cdot_{\mathbb{K}} t_1$
$x_{m+1} \leftarrow t_8 +_{\mathbb{K}} t_9$
$z_{m+1} \leftarrow t_8 -_{\mathbb{K}} t_9$
$x_{m+1} \leftarrow x_{m+1} \cdot_{\mathbb{K}} x_{m+1}$
$z_{m+1} \leftarrow z_{m+1} \cdot_{\mathbb{K}} z_{m+1}$
$z_{m+1} \leftarrow z_{m+1} \cdot_{\mathbb{K}} x_1$
$x_m \leftarrow t_6 \cdot_{\mathbb{K}} t_7$
$z_m \leftarrow 121666 \cdot_{\mathbb{K}} t_5$
$z_m \leftarrow z_m +_{\mathbb{K}} t_7$
$z_m \leftarrow z_m \cdot_{\mathbb{K}} t_5$
return $(x_m, z_m, x_{m+1}, z_{m+1})$
end function

To have constant execution time, the key idea is to compute *both* $[2m]P$ and $[2m+1]P$ at each iteration. The Montgomery Ladderstep is an efficient algorithm computing $[2m]P$ and $[2m+1]P$ from P, $[m]P$, and $[m+1]P$ on Montgomery curves (including Curve25519). The algorithm uses only x coordinates of the points. Furthermore, expensive divisions are avoided in the Ladderstep by projective representations. That is, the algorithm represents $x \div_{\mathbb{K}} z$ by the pair $x : z$ and works with fractions (Algorithm 9).

Let unprimed and primed variables denote their values before and after computation respectively. Write xy to denote $x \cdot_{\mathbb{K}} y$ for short. The Montgomery Ladderstep has the following algebraic specification [25]:[2]

$$
\begin{aligned}
x'_m &=_{\mathbb{K}} 4(x_m x_{m+1} -_{\mathbb{K}} z_m z_{m+1})(x_m x_{m+1} -_{\mathbb{K}} z_m z_{m+1}) \\
z'_m &=_{\mathbb{K}} 4x_1(x_m z_{m+1} -_{\mathbb{K}} z_m x_{m+1})(x_m z_{m+1} -_{\mathbb{K}} z_m x_{m+1}) \\
x'_{m+1} &=_{\mathbb{K}} (x_m x_m -_{\mathbb{K}} z_m z_m)(x_m x_m -_{\mathbb{K}} z_m z_m) \\
z'_{m+1} &=_{\mathbb{K}} 4x_m z_m(x_m x_m +_{\mathbb{K}} 121666 x_m z_m +_{\mathbb{K}} z_m z_m)
\end{aligned}
$$

For the range specification, the unsigned value of each limb used to represent an output x_m, z_m, x_{m+1}, or z_{m+1} must be in the range from 0 to $2^{51} + 2^{15}$. In our experiment, we replace all arithmetic computation over $\mathbb{K}$ with corresponding mathematical constructs (4 additions, 4 subtractions, 4 squares, 5 multiplications, and 1 multiplication by 121666) written in BVCRYPTOLINE, translate the above specification into an algebraic specification, a range specification, and a safety check, and then apply our technique to verify the Ladderstep (containing 1282 statements). The verification takes 131 hours, including 77 hours in safety check [3], 33 hours in checking range specification, 16 hours in checking algebraic specification, and the remaining hours in term rewriting, proof validation, etc. For production releases of low-level mathematical constructs, we believe 5.5 days in verification time are well invested.

7 CONCLUSION

We have developed techniques to verify algebraic and range specifications of low-level mathematical constructs in cryptographic programs. Our case studies on real low-level implementations of X25519 suggest the applicability and scalability of our techniques. Currently, we are working on automatic translation from assembly languages to our domain specific language. Such translation will make our verification techniques more accessible to assembly programmers. We are also applying our techniques to more low-level mathematical constructs in industrial cryptographic programs. Communication with assembly programmers will further improve the proposed techniques in practice.

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, 346–360.

[2] Amusingly, we find for ourselves the factor of 4 in both the numerator and denominator of the addition formulas during verification, noted on [25, p. 261].

[3] We verified the four equations in the algebraic specification separately and our COQ tactic checked the safety of the same program four times. The time needed to check program safety once is roughly 19 hours.

[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] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *POPL*. ACM, New York, NY, USA, 1–11.

[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. https://doi.org/10.1145/2701415

[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] Daniel J. Bernstein. 2005. Cache Timing Attacks on AES. (2005). https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[8] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography (LNCS)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.), Vol. 3958. Springer, 207–228.

[9] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-Speed High-Security Signatures. In *CHES (LNCS)*, Bart Preneel and Tsuyoshi Takagi (Eds.), Vol. 6917. Springer, 124–142.

[10] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-Speed High-Security Signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.

[11] Daniel J. Bernstein and Peter Schwabe. 2016. gfverif: Fast and Easy Verification of Finite-Field Arithmetic. (2016). http://gfverif.cryptojedi.org.

[12] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer.

[13] 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.

[14] 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 *CCS*. ACM, 299–309. https://doi.org/10.1145/2660267.2660370

[15] Henri Cohen. 1996. *A Course in Computational Algebraic Number Theory* (3rd ed.). GTM, Vol. 138. Springer.

[16] The Sage Developers. 2017. SageMath, the Sage Mathematics Software System. (2017). http://www.sagemath.org.

[17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *CAV (LNCS)*, Viktor Kuncak and Rupak Majumdar (Eds.). Springer.

[18] everest 2016. Project Everest. https://project-everest.github.io. (2016). Accessed: 2017-05-19.

[19] fiat 2015. Fiat-Crypto. https://github.com/mit-plv/fiat-crypto. (2015). Accessed: 2017-05-19.

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

[21] Gert-Martin Greuel and Gerhard Pfister. 2008. *A Singular Introduction to Commutative Algebra* (2nd ed.). Springer.

[22] John Harrison. 2007. Automating Elementary Number-Theoretic Proofs Using Gröbner Bases. In *CADE (LNCS)*, Frank Pfenning (Ed.), Vol. 4603. Springer, 51–66.

[23] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *CACM* 12, 10 (1969), 576–580.

[24] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. 2003. The Impact of Decryption Failures on the Security of NTRU Encryption. In *CRYPTO (LNCS)*, Dan Boneh (Ed.), Vol. 2729. Springer, 226–246.

[25] Peter L. Montgomery. 1987. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. Comp.* 48, 177 (1987), 243–264.

[26] 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, 66–81. https://doi.org/10.1007/978-3-319-03545-1_5

[27] Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *TACAS (LNCS)*, Orna Grumberg and Michael Huth (Eds.), Vol. 4424. Springer, 568–582.

[28] Loïc Pottier. 2008. Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. In *Knowledge Exchange: Automated Provers and Proof Assistants*, G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz (Eds.). 418.

[29] Loïc Pottier. 2010. *Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics.* Technical Report abs/1007.3615. CoRR.

[30] Aaron Tomb. 2016. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security & Privacy* 14, 6 (2016), 26–33. https://doi.org/10.1109/MSP.2016.125

[31] Wikipedia. 2017. Curve25519. https://en.wikipedia.org/wiki/Curve25519. (2017). Accessed: 2017-05-19.

A APPENDIX

A.1 Multiplication over $\mathbb{GF}(2^{255} - 19)$

The following BVCRYPTOLINE code implements multiplications over $\mathbb{GF}(2^{255} - 19)$ for the AMD64 architecture:

$$
\begin{array}{ll}
\mathit{mulrax} \leftarrow & \mathit{x3};\\
\mathit{mulrax} \leftarrow & \mathit{mulrax} \times bv^{64}(19);\\
\mathit{mulx319} \leftarrow & \mathit{mulrax};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y2};\\
\mathit{r0} \leftarrow & \mathit{mulrax};\\
\mathit{mulr01} \leftarrow & \mathit{mulrdx};\\
\mathit{mulrax} \leftarrow & \mathit{x4};\\
\mathit{mulrax} \leftarrow & \mathit{mulrax} \times bv^{64}(19);\\
\mathit{mulx419} \leftarrow & \mathit{mulrax};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y1};\\
\mathit{carry}\ \mathit{r0} \leftarrow & \mathit{r0} + \mathit{mulrax};\\
\mathit{mulr01} \leftarrow & \mathit{mulr01} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x0};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y0};\\
\mathit{carry}\ \mathit{r0} \leftarrow & \mathit{r0} + \mathit{mulrax};\\
\mathit{mulr01} \leftarrow & \mathit{mulr01} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x0};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y1};\\
\mathit{r1} \leftarrow & \mathit{mulrax};\\
\mathit{mulr11} \leftarrow & \mathit{mulrdx};\\
\mathit{mulrax} \leftarrow & \mathit{x0};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y2};\\
\mathit{r2} \leftarrow & \mathit{mulrax};\\
\mathit{mulr21} \leftarrow & \mathit{mulrdx};\\
\mathit{mulrax} \leftarrow & \mathit{x0};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y3};\\
\mathit{r3} \leftarrow & \mathit{mulrax};\\
\mathit{mulr31} \leftarrow & \mathit{mulrdx};\\
\mathit{mulrax} \leftarrow & \mathit{x0};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y4};\\
\mathit{r4} \leftarrow & \mathit{mulrax};\\
\mathit{mulr41} \leftarrow & \mathit{mulrdx};\\
\mathit{mulrax} \leftarrow & \mathit{x1};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y0};\\
\mathit{carry}\ \mathit{r1} \leftarrow & \mathit{r1} + \mathit{mulrax};\\
\mathit{mulr11} \leftarrow & \mathit{mulr11} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x1};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y1};\\
\mathit{carry}\ \mathit{r2} \leftarrow & \mathit{r2} + \mathit{mulrax};\\
\mathit{mulr21} \leftarrow & \mathit{mulr21} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x1};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y2};\\
\mathit{carry}\ \mathit{r3} \leftarrow & \mathit{r3} + \mathit{mulrax};\\
\mathit{mulr31} \leftarrow & \mathit{mulr31} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x1};\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \ast \mathit{y3};\\
\mathit{carry}\ \mathit{r4} \leftarrow & \mathit{r4} + \mathit{mulrax};\\
\mathit{mulr41} \leftarrow & \mathit{mulr41} + \mathit{mulrdx} + \mathit{carry};\\
\mathit{mulrax} \leftarrow & \mathit{x1};\\
\mathit{mulrax} \leftarrow & \mathit{mulrax} \times bv^{64}(19);\\
\mathit{mulrdx}\ \mathit{mulrax} \leftarrow & \mathit{mulrax} \times \mathit{y4};\\
\mathit{carry}\ \mathit{r0} \leftarrow & \mathit{r0} + \mathit{mulrax};
\end{array}
$$

```
53 : mulr01 ← mulr01 + mulrdx + carry;
54 : mulrax ← x2;
55 : mulrdx mulrax ← mulrax × y0;
56 : carry r2 ← r2 + mulrax;
57 : mulr21 ← mulr21 + mulrdx + carry;
58 : mulrax ← x2;
59 : mulrdx mulrax ← mulrax × y1;
60 : carry r3 ← r3 + mulrax;
61 : mulr31 ← mulr31 + mulrdx + carry;
62 : mulrax ← x2;
63 : mulrdx mulrax ← mulrax × y2;
64 : carry r4 ← r4 + mulrax;
65 : mulr41 ← mulr41 + mulrdx + carry;
66 : mulrax ← x2;
67 : mulrax ← mulrax × bv^64(19);
68 : mulrdx mulrax ← mulrax × y3;
69 : carry r0 ← r0 + mulrax;
70 : mulr01 ← mulr01 + mulrdx + carry;
71 : mulrax ← x2;
72 : mulrax ← mulrax × bv^64(19);
73 : mulrdx mulrax ← mulrax × y4;
74 : carry r1 ← r1 + mulrax;
75 : mulr11 ← mulr11 + mulrdx + carry;
76 : mulrax ← x3;
77 : mulrdx mulrax ← mulrax × y0;
78 : carry r3 ← r3 + mulrax;
79 : mulr31 ← mulr31 + mulrdx + carry;
80 : mulrax ← x3;
81 : mulrdx mulrax ← mulrax × y1;
82 : carry r4 ← r4 + mulrax;
83 : mulr41 ← mulr41 + mulrdx + carry;
84 : mulrax ← mulx319;
85 : mulrdx mulrax ← mulrax × y3;
86 : carry r1 ← r1 + mulrax;
87 : mulr11 ← mulr11 + mulrdx + carry;
88 : mulrax ← mulx319;
89 : mulrdx mulrax ← mulrax × y4;
90 : carry r2 ← r2 + mulrax;
91 : mulr21 ← mulr21 + mulrdx + carry;
92 : mulrax ← x4;
93 : mulrdx mulrax ← mulrax × y0;
94 : carry r4 ← r4 + mulrax;
95 : mulr41 ← mulr41 + mulrdx + carry;
96 : mulrax ← mulx419;
97 : mulrdx mulrax ← mulrax × y2;
98 : carry r1 ← r1 + mulrax;
99 : mulr11 ← mulr11 + mulrdx + carry;
100 : mulrax ← mulx419;
101 : mulrdx mulrax ← mulrax × y3;
102 : carry r2 ← r2 + mulrax;
103 : mulr21 ← mulr21 + mulrdx + carry;
104 : mulrax ← mulx419;
105 : mulrdx mulrax ← mulrax × y4;
106 : carry r3 ← r3 + mulrax;
107 : mulr31 ← mulr31 + mulrdx + carry;
108 : mulr01 r0 ← (mulr01.r0) ≪ 13;
109 : mulr11 r1 ← (mulr11.r1) ≪ 13;
110 : r1 ← r1 + mulr01;
111 : mulr21 r2 ← (mulr21.r2) ≪ 13;
112 : r2 ← r2 + mulr11;
113 : mulr31 r3 ← (mulr31.r3) ≪ 13;
114 : r3 ← r3 + mulr21;
115 : mulr41 r4 ← (mulr41.r4) ≪ 13;
116 : r4 ← r4 + mulr31;
117 : mulr41 ← mulr41 × bv^64(19);
118 : r0 ← r0 + mulr41;
119 : mult ← r0;
120 : mult tmp ← mult@51;
121 : mult ← mult + r1;
122 : r1 ← mult;
123 : multtmp2 ← mult@51;
124 : r0 ← tmp;
125 : mult ← mult + r2;
126 : r2 ← mult;
127 : mult tmp ← mult@51;
128 : r1 ← tmp2;
129 : mult ← mult + r3;
130 : r3 ← mult;
131 : mult tmp2 ← mult@51;
132 : r2 ← tmp;
133 : mult ← mult + r4;
134 : r4 ← mult;
135 : mult tmp ← mult@51;
136 : r3 ← tmp2;
137 : mult ← mult × bv^64(19);
138 : r0 ← r0 + mult;
139 : r4 ← tmp;
140 : z0 ← r0;
141 : z1 ← r1;
142 : z2 ← r2;
143 : z3 ← r3;
144 : z4 ← r4;
```

Let bMul denote the above program. Define $q_a \triangleq \top$, $q_r \triangleq 0 \leq \mathit{x0}, \mathit{x1}, \mathit{x2}, \mathit{x3}, \mathit{x4}, \mathit{y0}, \mathit{y1}, \mathit{y2}, \mathit{y3}, \mathit{y4} \leq bv^{64}(2^{52})$, $q'_a \triangleq \mathit{radix51}_{\mathbb{V}}(\mathit{x4}, \mathit{x3}, \mathit{x2}, \mathit{x1}, \mathit{x0}) \times \mathit{radix51}_{\mathbb{V}}(\mathit{y4}, \mathit{y3}, \mathit{y2}, \mathit{y1}, \mathit{y0}) \equiv \mathit{radix51}_{\mathbb{V}}(\mathit{z4}, \mathit{z3}, \mathit{z2}, \mathit{z1}, \mathit{z0}) \bmod bv^{64}(2^{255}-19)$, and $q'_r \triangleq 0 \leq \mathit{z0}, \mathit{z1}, \mathit{z2}, \mathit{z3}, \mathit{z4} \leq bv^{64}(2^{52})$. Its specification is

$$(\!|q_a \mid q_r|\!) \quad \mathsf{bMul} \quad (\!|q'_a \mid q'_r|\!).$$