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
We use the computer algebra system Singular to Coq, a fully certified integration of SMT solvers in the trusted computing base of our approach hence includes SMT implementation of the Montgomery Ladderstep is verified similarly. Their algebraic specifications with our automatic technique. The translation to SMT formulas is again certified by Coq. The results of SMT solvers however are not certified. The trusted computing base in the future [17].

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 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.

\[ \text{to SSA (Sec. 4.1)} \]
\[ \text{to zCryptoLine (Sec. 4.2)} \]
\[ \text{SSA form of algebraic specification in bvCryptoLine} \]
\[ \text{to entailment (Sec. 4.3)} \]
\[ \text{polynomial equation entailment solved by (Sec. 5.2)} \]
\[ \text{SINGULAR} \]
\[ \text{range specification} \]
\[ \text{overflow/underflow check solved by (Sec. 5.1)} \]
\[ \text{SMT solvers} \]

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-implementation 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 $B = \{0, \top\}$ for the Boolean domain. Let $\mathbb{N}$ and $\mathbb{Z}$ denote all natural numbers and all integers respectively. We use $\lbrack n \rbrack$ to denote the set $\{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$.

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

Given $G \subseteq \mathbb{N}$, $(\cdot)$ is commutative.

A ring $\mathbb{R} = \langle \mathbb{R}, 0, 1, +, \times \rangle$ is a commutative ring.

We write $\mathbb{F} = \{0, 1, +, \times\}$ for a finite field.

Given $G \subseteq \mathbb{N}$, $(\cdot)$ is a commutative ring.

If $x$ is commutative, $\mathbb{R}$ is a commutative ring.

A field $\mathbb{F} = (\mathbb{F}, 0, 1, +, \times)$ is a commutative ring.

For any prime number $p$, the set $\{0, \ldots, q - 1\}$ with the addition and multiplication modulo $q$ forms a finite field.

A ring $\mathbb{R}$ is a commutative ring if $x$ is commutative.

Fix a set of variables $\vec{x}$. $\mathbb{R}[\vec{x}]$ is the polynomial ring over $\mathbb{R}$ with coefficients in the ring $\mathbb{R}$. $\mathbb{R}[\vec{x}]$ is a ring.

A set $I \subseteq \mathbb{R}[\vec{x}]$ is an ideal if

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

Given $G \subseteq \mathbb{R}[\vec{x}]$, $(\cdot)$ is the ideal containing $G$; $G$ are the generators of $(\cdot)$. The ideal membership problem is to decide if $f \in I$ for a given ideal $I$ and $f \in \mathbb{R}[\vec{x}]$.

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 $w$.

Let $\forall w$ be the set of all bit-vectors with a bit-width $w$. The unsigned value of $b \in \forall w$ is denoted by $[b]$. For a natural number or an integer $n$, let $b_v(n)$ be the two’s complement representation of $n$ in a bit-width $w$. We use the following common operators for fixed-width bit-vectors: $\forall w_1 \uplus \forall w_2 : \forall w_1 \uplus \forall w_2$ for addition, $\forall w_1 \ominus \forall w_2 : \forall w_1 \ominus \forall w_2$ for subtraction, $\forall w_1 \times \forall w_2 : \forall w_1 \times \forall w_2$ for multiplication, $\forall w_1 \div \forall w : \forall w_1 \div \forall w$ for concatenation, $\forall w_1 \leftarrow n : \forall w_1 \leftarrow n$ for zero extension, $\forall w \ll j \leftarrow \forall n : \forall w \ll j \leftarrow \forall n$ for left-shifting, $\forall w \gg j \leftarrow \forall n : \forall w \gg j \leftarrow \forall n$ for logical right-shifting, and $\forall w[i, j] : \forall w[i, j]$ with $0 \leq j \leq i < w$ for bits extraction. We also assume comparison operators $<, > , \leq , \geq$ between unsigned values of bit-vectors.

Given a bit-vector $b \in \forall 2^w$, define $\text{h{i}j}(b) \triangleq [b[2^w - 1, 0]]$ for the extraction of higher $w$ bits, and $\text{i0}(b) \triangleq [b[w - 1, 0]]$ for the extraction of lower $w$ bits. For operations $\in \{+, -, \times, \div\}$, we define their extended version $\ast$ 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 $b_1, b_2, b_3 \in \forall w$, we have $b_1 + \ast b_2 \triangleq (b_1 \# \forall w) + (b_2 \# \forall w)$ and $b_1 \div \ast b_2 \div \ast b_3 \triangleq (b_1 \# \forall w) \div (b_2 \# \forall w) \div (b_3 \# \forall w)$.

4 IMPLEMENTATION
Define $v[u \leftrightarrow d](u) = \begin{cases} d & \text{if } u = v \\ v(u) & \text{otherwise} \end{cases}$. Define the semantic function $\llbracket \cdot \rrbracket(v)$ for variables and atoms as follows.

$$\llbracket v \rrbracket(v) \triangleq \begin{cases} v(v) & \text{for } v \in \text{Var} \\ v(v) & \text{if } a \text{ is a variable } v \\ b & \text{if } a \text{ is a bit-vector } b \end{cases}$$

Consider the transition relation $bTr \subseteq bSt \times bStmt \times bSt$ defined in Figure 2 where $v \Rightarrow v'$ denotes $(v, s, v') \in bTr$ for $v, v' \in bSt$ and $s \in bStmt$. Basically, $v \leftarrow a_1 + a_2$ is addition, $v \leftarrow a_1 + a_2 + y$ is addition with carry bit placed in $c$, $v \leftarrow a_1 + a_2 + y + y$ is addition of atoms plus a variable $y$, $v \leftarrow a_1 + a_2 + y$ 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, $v, v_1 \leftarrow a_1 \times a_2$ is full multiplication, $v \leftarrow a \ll n$ is left-shifting, $v_1, v_i \leftarrow a \ll n$ is splitting at position $n$, and $v_1, v_i \leftarrow (a_1, a_2) \ll n$ 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 $c \leftarrow v \leftarrow a_1 + a_2 + y$ is intended but not restricted to be carry bits.

A program is a sequence of statements. We denote the empty program by $\epsilon$.

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

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 $(v, \epsilon, v) \in bTr^*$ and $(v, s, p, v') \in bTr^*$ if there is a $v'$ with $(v, s, v') \in bTr$ and $(v', p, v') \in bTr^*$. We write $v \triangleright p$ when $(v, p, v') \in bTr^*$.

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 $e_1 \equiv e_2$ and modular polynomial equations $e_1 \equiv e_2 \mod e_3$ 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.

$$bExp_a ::= \mathbb{Z} | \text{Var} \mid -bExp_a \mid bExp_a + bExp_a \mid \neg bExp_a \mid bExp_a \land bExp_a \mid bExp_a \lor bExp_a \mid bExp_a \Rightarrow bExp_a \mid bExp_a \Rightarrow bExp_a \mid bExp_a \Rightarrow bExp_a$$

An algebraic predicate $q_a \in bPred_a$ is then a conjunction of atomic algebraic predicates.

Given a state $v \in bSt$ and an expression $e \in bExp_a$, $\llbracket e \rrbracket_a(v)$ denotes the value of $e$ on $v$.

For an algebraic predicate $q_a \in bPred_a$, we write $\mathbb{V}^w \models q_a[v]$ if $q_a$ evaluates to $tt$ using the evaluation function $\llbracket e \rrbracket_a(v)$ 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.

$$bPred_r ::= \top \mid bAtom \land bAtom | bAtom \land bAtom$$

We use $a_1 \circ a_2, \ldots, a_n$, as a shorthand of the conjunction of $a_1 \circ a_1 \land a_2 \circ a_2 \land \cdots \land a_n \circ a_n$ and $a_1 \circ a_2 \land a_3 \circ a_3 \land \cdots \land a_n \circ a_n$, where $o, \circ \in \{<, \leq\}$. For $q_r \in bPred_r$ and $v \in bSt$, we write $\mathbb{V}^w \models q_r[v]$ if one of the following holds.

- $q$ is $\top$.
- $q$ is $a_1 < a_2$ and $\llbracket a_1 \rrbracket(v) < \llbracket a_2 \rrbracket(v)$.
- $q$ is $a_1 \leq a_2$ and $\llbracket a_1 \rrbracket(v) \leq \llbracket a_2 \rrbracket(v)$.
- $q$ is $q_1 \land q_2$. $\mathbb{V}^w \models q_1[v]$, and $\mathbb{V}^w \models q_2[v]$.

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

$$bPred ::= bPred_a \land bPred_r$$

For $v \in bSt$ and $q \in bPred$, we write $\mathbb{V}^w \models q[v]$ if $q$ evaluates to $tt$; $v$ is called a $q$-state. We follow Hoare’s formalism in specifications of mathematical constructs$^2$ and call $(q(p)(q'))$ a specification if $q(p)(q') \in bPred$, an algebraic specification if $q(p)(q') \in bPred_a$, and a range specification if $q(p)(q') \in bPred_r$. In $(q(p)(q'))$, $q$ and $q'$ are the pre- and post-conditions of $p$ respectively. Given $q, q' \in bPred$, $Q(p)(Q'(p'))$ is defined.

$^1$In our implementation, comparison between bit-vector expressions is allowed, not only between atoms.
Algorithm 1 Safety Test for Statements

1: function StmtSafe(s, v)
2:     match s with
3:         case v ← a: return tt
4:         case v ← a + a:
5:             return (v1 ⊕ v2) ⊕ (v1 ⊕ v2) = bv^0
6:         case v ← a + a + y:
7:             return (v1 ⊕ v2) ⊕ (v1 ⊕ v2) = bv^0
8:         case v ← a + a + y:
9:             return (v1 ⊕ v2) = bv^0
10:        case v ← a + a + y:
11:            return (v1 ⊕ v2) = bv^0
12:        case v ← a + a + y:
13:            return (v1 ⊕ v2) = bv^0
14:        case v ← a + a + y:
15:            return (v1 ⊕ v2) = bv^0
16:        case v ← a + a + y:
17:            return (v1 ⊕ v2) = bv^0
18:        case v ← a + a + y:
19:            return (v1 ⊕ v2) = bv^0
20: end function

Figure 3 gives a simple yet real implementation of subtraction over \( \mathbb{F}(q) \) 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 \( \mathbb{F}(q) \) is represented by five bit-vectors each with value less than or equal to \( 2^{51} + 2^{15} \). The variables \( x_0, x_1, x_2, x_3, x_4 \) for instance represent \( \text{radix51}(x_4, x_3, x_2, x_1, x_0) = (2^{51}x_4) + (2^{51}x_3) + (2^{51}x_2) + (2^{51}x_1) + (2^{51}x_0) \). 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 \( 2^{51} \). Let \( \text{radix51}(x_4, x_3, x_2, x_1, x_0) \) denote the representation of \( \text{radix51}(x_4, x_3, x_2, x_1, x_0) \) in \( b\text{Exp}_q \). Let \( q_a \triangleq T, q_r \triangleq 0 \leq x_0, x_1, x_2, x_3, x_4 \leq y_0 \). Let \( q_a = \text{radix51}(0x0FFFFFFFHFFFE), \) \( q_r = \text{radix51}(0x0FFFFFFFHFFFE), \) \( q_a = \text{radix51}(0x0FFFFFFFHFFFE), \) \( q_r = \text{radix51}(0x0FFFFFFFHFFFE), \) \( q_a = \text{radix51}(0x0FFFFFFFHFFFE), \) and \( q_r = \text{radix51}(0x0FFFFFFFHFFFE), \) which is not hard to see that

\[
\begin{align*}
2q &= \text{radix51}(0x0FFFFFFFHFFFE),
\end{align*}
\]

Figure 3: Subtraction bSub

Note that the variables \( r_j \)’s are added with constants after they are initialized with \( x_j \)’s but before \( y_j \)’s are subtracted from them. It is not hard to see that

\[
\begin{align*}
2q &= \text{radix51}(0x0FFFFFFFHFFFE),
\end{align*}
\]

after tedious computation. Hence

\[
\begin{align*}
\text{radix51}(x_4, x_3, x_2, x_1, x_0) &= \text{radix51}(y_4, y_3, y_2, y_1, y_0)
\end{align*}
\]

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 \( q_a, q_r \in b\text{Pred}_a, q_r, q_r' \in b\text{Pred}_r, \) and \( p \in b\text{Prog} \), we reduce the problem of checking \( \models [q_a; q_r; p] \) to (1) the entailment problem of modular polynomial equations over integer variables proving \( \models [q_a; p] \) via an intermediate language \( z\text{CRYPTOLINE} \), (2) a range problem \( \models [q_r; p] \) 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. \( z\text{CRYPTOLINE} \). The algebraic specification \( [q_a; p] \) in \( b\text{CRYPTOLINE} \) is transformed to a specification in \( z\text{CRYPTOLINE} \) so that the validity of the specification in \( z\text{CRYPTOLINE} \) implies the validity of \( [q_a; p] \) in \( b\text{CRYPTOLINE} \) if the program \( p \) is safe. (Section 4.2).
3. Modular polynomial equations. Validity of algebraic specifications in \( z\text{CRYPTOLINE} \) 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 zCryptOL and validity of specifications in zCryptOL 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 \( p \) 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 variables \( c \) and \( v \), the two variables are different variables; and
- every non-input program variable must be assigned to a value before being used.

Our transformation maintains a finite mapping \( \theta \) from variables to non-negative integers. For any variable \( v \), \( v^\theta(v) \) is the most recently assigned copy of \( v \). For any atom \( a, a^\theta = v^\theta(v) \) when \( a \) is a variable \( v \), and otherwise \( b \) 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 \( \theta \) by structural induction. Integers are unchanged. For each variable, its most recent copy is returned by looking up the mapping \( \theta \). Other algebraic expressions are transformed recursively.

#### Algorithm 2 Static Single Assignment Transformation for Algebraic Expressions

```plaintext
1: function SSAExpra(\( \theta, e \))
2: match \( e \) with
3: case \( i \): return \( i \)
4: case \( v \): return \( v^\theta(v) \)
5: case \( -e' \): return \(- SSAExpra(\theta, e') \)
6: case \( e_1 + e_2 \):
7:     return SSAExpra(\( \theta, e_1 \)) + SSAExpra(\( \theta, e_2 \))
8: case \( e_1 - e_2 \):
9:     return SSAExpra(\( \theta, e_1 \)) - SSAExpra(\( \theta, e_2 \))
10: case \( e_1 \times e_2 \):
11:     return SSAExpra(\( \theta, e_1 \)) \times SSAExpra(\( \theta, e_2 \))
12: end function
```

Similarly, algebraic and range predicates must refer to most recent copies of variables. They are transformed according to the finite mapping \( \theta \). 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 separately with SSAPreda and SSAPredr, that is, given \( q_a \in bPred_a, q_r \in bPred_r \), and a mapping \( \theta \), \( SSAPred(\theta, q_a; q_r) = SSAPreda(\theta, q_a); SSAPredr(\theta, q_r) \).

#### Algorithm 3 Static Single Assignment Transformation for Statements

```plaintext
1: function SSAStmt(\( \theta, s \))
2: match \( s \) with
3: case \( v \leftarrow a_1 \):
4:     \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
5:     return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \)
6: case \( v \leftarrow a_1 + a_2 \):
7:     \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
8:     return \( 0', \theta^\theta(v) \leftarrow a_1^\theta + a_2^\theta \)
9: case \( c \ v \leftarrow a_1 + a_2 \):
10:    \( 0' \leftarrow 0[c \leftarrow \theta(c) + 1] \)
11:    return \( 0', \theta^\theta(c) \theta^\theta(v) \leftarrow a_1^\theta + a_2^\theta + y^\theta(y) \)
12: case \( c \ v \leftarrow a_1 + a_2 + y \):
13:    \( 0' \leftarrow 0[c \leftarrow \theta(c) + 1] \)
14:    return \( 0', \theta^\theta(c) \theta^\theta(v) \leftarrow a_1^\theta + a_2^\theta + y^\theta(y) \)
15: case \( v \leftarrow a_1 \times a_2 \):
16:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
17:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
18: case \( v \leftarrow a_1 \times a_2 \):
19:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
20:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
21: case \( v \leftarrow a_1 \times a_2 \):
22:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
23:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
24: case \( v \leftarrow a_1 \times a_2 \):
25:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
26:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
27: case \( v \leftarrow a_1 \times a_2 \):
28:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
29:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
30: case \( v \leftarrow a_1 \times a_2 \):
31:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
32:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
33: case \( v \leftarrow a_1 \times a_2 \):
34:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
35:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
36: case \( v \leftarrow a_1 \times a_2 \):
37:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
38:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
39: case \( v \leftarrow a_1 \times a_2 \):
40:    \( 0' \leftarrow 0[v \leftarrow \theta(v) + 1] \)
41:    return \( 0', \theta^\theta(v) \leftarrow a_1^\theta \times a_2^\theta \)
```

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 \( \theta \). The algorithm of statement transformation then updates \( \theta \) and replaces assigned variables with their latest copies.
It is straightforward to transform programs to static single assignment form (Algorithm 4). Using the initial mapping \( \theta_0 \) 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 \( \theta_0 \) to simplify our Coq proof.

**Algorithm 4 Static Single Assignment for Programs**

```plaintext
def SSAProg(θ, p)
  match p with
  | c : return (θ, c)
  | s : pp:
    (Ø', s') ← SSAStmt(θ, s)
    (Ø'', pp'') ← SSAProg(Ø', pp)
    return (Ø'', s'; pp'')

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 \( v \in \text{Var} \) and \( p \in b\text{Prog} \) a well-formed program. If \((\hat{θ}, \hat{p}) = SSAProc(θ_0, p)\), 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 \( θ_0(v) = 0 \) for every \( v \in \text{Var} \). For every \( q, q' \in b\text{Pred} \) and \( p \in \text{bProg} \),

\[
\{q \models p(q') \} \text{ if and only if } \{SSAPred}(θ_0, q)\} \models \{SSAPred}(\hat{θ}, q')\}
\]

where \((\hat{θ}, \hat{p}) = SSAProc(θ_0, p)\).

**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 \( x_i, y_j \)'s are indexed by 0 and \( r_i \)'s are indexed by 3 for \( 0 ≤ i ≤ 4 \). Subsequently, variables in pre- and post-conditions of the specification for subtraction need to be indexed. Let \( q_{θ} \triangleq T, q_{θ} = 0 ≤ x_{θ_{0}} x_{θ_{1}} x_{θ_{2}} x_{θ_{3}} x_{θ_{4}} x_{θ_{5}} \), \( y_{θ_{0}} y_{θ_{1}} y_{θ_{2}} y_{θ_{3}} y_{θ_{4}} y_{θ_{5}} ≤ \text{bv}^{64}(2^{51}+2^{15}) \), \( q_{θ} \triangleq \text{radix51}(x_{θ_{1}} x_{θ_{2}} x_{θ_{3}} x_{θ_{4}} x_{θ_{5}}) − \text{radix31}(y_{θ_{1}} y_{θ_{2}} y_{θ_{3}} y_{θ_{4}} y_{θ_{5}}) \equiv \text{radix51}(r_{θ_{1}} r_{θ_{2}} r_{θ_{3}} r_{θ_{4}} r_{θ_{5}}) \pmod{g} \), and \( q_{θ} = 0 ≤ r_{θ_{1}} r_{θ_{2}} r_{θ_{3}} r_{θ_{4}} r_{θ_{5}} ≤ \text{bv}^{64}(2^{34}) \). The corresponding specification of in static single assignment form is then

\[
\{q_θ ∧ q_θ \models bSubSSA[q_θ ∧ q_θ]\}.
\]

**4.2 zCRYPTOLINE**

Algorithmic 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:

\[
\text{zExpr} ::= \text{Z} \mid \text{Var} \mid \text{zExpr} + \text{zExpr} \mid \text{zExpr} \cdot \text{zExpr} \mid \text{Pow}(\text{zExpr}, n)\]

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 \( \text{Pow}(e, n) \) denotes \( e^n \) for any expression \( e \) and natural number \( n \). More formally, let \( z\text{St} \triangleq \text{Var} \rightarrow Z \) and \( μ \in z\text{St} \) be a state. That is, a state \( μ \) in zCRYPTOLINE is a mapping from variables to integers. Define the semantic function \( \mathbb{E}(Z)(μ) \) as follows.

\[
\mathbb{E}(Z)(μ) \triangleq \begin{cases} i \text{ for } i \in \mathbb{Z} \\ \mathbb{E}(μ(v)) \text{ for } v \in \text{Var} \\ \mathbb{E}(\bar{e}) \text{ for } e \in \mathbb{E} \\ \mathbb{E}(\mathbb{E}(e) + \mathbb{E}(μ)) \\ \mathbb{E}(\mathbb{E}(μ) + \mathbb{E}(e)) \\ \mathbb{E}(\mathbb{E}(μ)) \times \mathbb{E}(e) \\ \mathbb{E}(\text{Pow}(e, n)) \end{cases}
\]

In zCRYPTOLINE, only assignments are allowed. The statement \( v ← e \) 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 \( [v_1, v_2] ← \text{Split}(e, n) \) splits the value of \( e \) into two parts; the lowest \( n \) bits are stored in \( v_1 \) and the remaining higher bits are stored in \( v_2 \). Consider the relation \( z\text{Tr} \subseteq z\text{St} \times z\text{St} \times z\text{St} \) defined by \( (μ, ν, ε) \in z\text{Tr} \) if \( (μ(v_1), ν(v_2), ε) \in z\text{Tr} \), and \( (μ(v_1), ν(v_2), ε) \in \text{Split}(e, n, μ(v_1), ν(v_2), ε) \in z\text{Tr} \) where \( ε = \frac{\mathbb{E}(Z)(μ) - \mathbb{E}(Z)(ν)}{2^n} \) and \( ν = \mathbb{E}(Z)(μ) \mod 2^n \). Intuitively, \( (μ, s, ε) \in z\text{Tr} \) denotes that the state \( μ \) transits to the state \( μ' \) after executing the statement \( s \).

\[
\text{zStmt} ::= \text{Var} ← \text{zExpr} \mid [\text{Var}, \text{Var}] ← \text{Split}(\text{zExpr}, n)\]

\[
\text{zProg} ::= ε \mid \text{zStmt}; \text{zProg}
\]

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 \( z\text{Tr}^* \subseteq z\text{St} \times z\text{Prog} \times z\text{St} \) where \( (μ, ε, μ') \in z\text{Tr}^* \) and \( (μ, s; p), μ' \in z\text{Tr}^* \) if there is a \( μ' \) with \( (μ, s; p), μ' \in z\text{St} \) and \( (μ', p), μ'' \in z\text{Tr}^* \). We write \( μ \equiv ε μ' \) when \( (μ, p, μ') \in z\text{Tr}^* \).

The predicates \( z\text{Pred} \) in zCRYPTOLINE share the same syntax as the algebraic predicates in bvCRYPTOLINE but are evaluated on \( z\text{St} \).
rather than on bSt.

$$z_{\text{Pred}} \iff T \land z_{\text{Expr}} = z_{\text{Expr}} \equiv z_{\text{Expr}} \mod z_{\text{Expr}} \land z_{\text{Pred}} \land z_{\text{Pred}}$$

For $\mu \in Z$ and $q \in b_{\text{Pred}}$, write $Z \models q[\mu]$ if $q$ evaluates to $tt$ using the evaluation function $[[e]]_Z(\mu)$ for every subexpression $e$ in $q$. Given $q, q' \in z_{\text{Pred}}$ and $p \in z_{\text{Prog}}$, $(q)p \equiv (q')p$ is valid (written $\models (q)p \equiv (q')p$) if for every $\mu, \mu' \in Z$, $Z \models q[\mu] \land \mu \Rightarrow \mu'$ imply $Z \models q'[\mu']$.

Now we are ready to describe the transformation from an algebraic specification in $\text{bCryptoline}$ to a specification in $\text{zCryptoline}$. Given $v \in bSt$ and $\mu \in Z$, write $v \equiv \mu$ when $[v(\nu)] = \mu(\nu)$ for all variable $\nu \in Var$. For algebraic expressions, since $z_{\text{Expr}}$ subsumes $b_{\text{Expr}}$, we can easily define a function $\text{BVZ2Expr}$ that converts an algebraic expression $e_\alpha \in b_{\text{Expr}}$ to $z_{\text{Expr}}$ such that for every $v \in bSt$ and $\mu \in Z$ with $v \equiv \mu$, $[[e]]_Z(v) = [[\text{BVZ2Expr}(e_\alpha)]_Z(\mu)$. Similarly, we can define a function $\text{BVZ2Pred}$ such that for every $q_\alpha \in b_{\text{Pred}}$, $v \in bSt$, and $\mu \in Z$ with $v \equiv \mu$, $\forall^W \models q_\alpha[v]$ if and only if $\exists Z \models \text{BVZ2Pred}(q_\alpha)[\mu]$. Atoms are translated by the function $\text{BVZ2Atom}$.

$$\text{BVZ2Atom}(a) = \begin{cases} \nu & \text{if } a \text{ is a variable } \nu \\ b & \text{if } a \text{ is a bit-vector } b \end{cases}$$

Let $\tilde{a}$ denote $\text{BVZ2Atom}(a)$ for $a \in b\text{Atom}$. The function $\text{BV2ZStmt}(s)$ (Algorithm 5) is defined to transform a statement in $\text{bCryptoline}$ to a statement in $\text{zCryptoline}$. Define a function $\text{BVZ2Prog}(\text{bProg})$ recursively such that $\text{BVZ2Prog}(\text{bProg}) \triangleq \text{bProg}$ and $\text{BVZ2Prog}(s; p) \triangleq \text{BVZ2Stmt}(s) \land \text{BVZ2Prog}(p)$. With these translation functions, the following soundness theorem holds.

Theorem 4.3. For every $q_\alpha, q'_\alpha \in b_{\text{Pred}}, q_r, q'_r \in b_{\text{Pred}},$ and $p \in b_{\text{Prog}}, \models (q_\alpha, q_r)p \equiv (q'_\alpha, q'_r)p$ if and only if the following conditions hold:

\begin{enumerate}
  \item $\forall^W \models q_\alpha[v]$ implies $\text{Procsafe}(p, v) \equiv \text{tt}$ for all $v \in bSt$.
  \item $\models (q_r) \equiv (q'_r)$.
  \item $\models (\text{BVZ2Pred}(q_\alpha)) \land (\text{BVZ2Prog}(p) \land (\text{BVZ2Pred}(q'_\alpha))$.
\end{enumerate}

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 $\models (q_r)p \equiv (q'_r)p$ may require that $q_\alpha$ holds initially but we do not consider any algebraic predicates in verifying range specifications.

The function $\text{BVZ2Prog}$ preserves well-formedness and static single assignment form. This is shown by the following lemma.

Lemma 4.4. Given a well-formed program $p \in b_{\text{Prog}}$ in static single assignment form, $\text{BVZ2Prog}(p) \in \text{zProg}$ is well-formed and in static single assignment form.

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

1. function $\text{BV2ZStmt}(s)$
2.    match $s$ with
3.     case $\nu \leftarrow a$: return $\nu \leftarrow \tilde{a}$
4.     case $\nu \leftarrow a + b$: return $\nu \leftarrow \tilde{a} + \tilde{b}$
5.     case $c \equiv a + b$: return $\nu \leftarrow \tilde{a} + \tilde{b}$
6.     case $\nu \leftarrow a + b + y$: return $\nu \leftarrow \tilde{a} + \tilde{b} + y$
7.     case $c \equiv a + b + y$: return $\nu \leftarrow \tilde{a} + \tilde{b} + y$
8.     case $[c, v] \leftarrow \text{Split}(\tilde{a} + \tilde{b} + y, w)$
9.     case $\nu \leftarrow a - b$: return $\nu \leftarrow \tilde{a} - \tilde{b}$
10.    case $\nu \leftarrow a - b$: return $\nu \leftarrow \tilde{a} - \tilde{b}$
11.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
12.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
13.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
14.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
15.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
16.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
17.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$
18.    case $\nu \leftarrow a \times b$: return $\nu \leftarrow \tilde{a} \times \tilde{b}$

Figure 5 shows the result of transforming the subtraction program $b\text{SubSSA}$ to $\text{zCryptoline}$.

4.3 Modular Polynomial Equation Entailment

The last step transforms any algebraic program specification in $\text{zCryptoline}$ to the modular polynomial equation entailment problem. For $q \in z_{\text{Pred}}$, we write $q(\bar{x})$ to signify the free variables $\bar{x}$ occurring in $q$. Given $q(\bar{x}), q'(\bar{x}) \in z_{\text{Pred}},$ the modular polynomial equation entailment problem decides whether $q(\bar{x}) \equiv q'(\bar{x})$ holds when $\bar{x}$ are substituted for arbitrary integers. That is, we want to check if for every valuation $\nu \in Z$, $q(\bar{x})$ evaluates to $tt$ implies $q'(\bar{x})$ evaluates to $tt$ after each variable $x$ is replaced by $\nu(x)$. We write $\exists \nu. q(\bar{x}) \equiv q'(\bar{x})$ if it is indeed the case.

Programs in static single assignment form can easily be transformed to conjunctions of polynomial equations. The function $\text{StmtToPolyEq}$ (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 $\text{ProgToPolyEq}$, which is recursively defined such that $\text{ProgToPolyEq}(e) \triangleq T$ and $\text{ProgToPolyEq}(s; p) \triangleq \text{StmtToPolyEq}(s) \land \text{ProgToPolyEq}(p)$.
Algorithm 6 Polynomial Equation Transformation for Statements

1: function STMTToPolyEQ(s)
2:   match s with
3:     case v ← e: return v = e
4:     case [v₁, v₂] ← Split(e, n):
5:       return v₁ + Pow(2, n) × v₂ = e
6: 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) ∈ zPred for every p ∈ 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 ∈ zProg be a well-formed program in static single assignment form. For every μ, μ′ ∈ zSt with μ ⋵ p ⋵ μ′, we have Z ⋵ ProgToPolyEQ(p)[μ′].

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

Definition 4.6. For q, q′ ∈ zPred and p ∈ zProg in static single assignment form, define

\[ \Pi([q]) p \equiv q'(x) \]

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

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 \( \text{radix51}(x_4, x_3, x_2, x_1, x_0) \equiv \text{Pow}(2, 51 \times 2^4) \times x_4 + \text{Pow}(2, 51 \times 2^3) \times x_3 + \text{Pow}(2, 51 \times 2^2) \times x_2 + \text{Pow}(2, 51 \times 2^1) \times x_1 + \text{Pow}(2, 51 \times 2^0) \times x_0 \) for \( x_4, x_3, x_2, x_1, x_0 \) ∈ Var. We want to know if \( \text{radix51}(r_1^3, r_2^3, r_3^3, r_4^3) \) is the difference between \( \text{radius51}(x_4^3, x_3^3, x_2^3, x_1^3, x_0^3) \) and \( \text{radius51}(q_4^3, y_3^3, y_2^3, y_1^3, y_0^3) \) in \( \mathbb{G}(q) \) 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 \([v_h, v_1] \leftarrow \text{Split}(e, n)\), the polynomial equation \( v_1 + \text{Pow}(2, n) \times v_h = e \) does not guarantee that \( v_1 \) exactly represents the lower \( n \) bits of \( e \).

Theorem 4.7. Let q, q′ ∈ zPred be predicates, and p ∈ zProg a well-formed program in static single assignment form. If \( Z \models \forall x. \Pi([q]) p \equiv q'(x) \), then \( \models [q] p \equiv [q'] \).
the predicates in QF_BV. An expression $e \in qExp$ can be a constant $bvconst(n, b)$, a variable $v \in Var$, an addition $bvadd(e_1, e_2)$, a subtraction $bvsub(e_1, e_2)$, a multiplication $bvmul(e_1, e_2)$, a concatenation $concat(e_1, e_2)$, a zero extension $zero\_extend(e', i)$, a left-shifting $bvshl(e_1, e_2)$, a logical right-shifting $bshr(e_1, e_2)$, or an extraction $bvextract(e', i, j)$ 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-or-equal $bvue(e_1, e_2)$, a negation $\neg q'$, a conjunction $q_1 \land q_2$, or a disjunction $q_1 \lor q_2$ 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 $\neg q_1 \lor q_2$.

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

\[
\begin{align*}
  bvhigh(e) & \triangleq bvextract(e, 2w - 1, n) \\
  b-low(e) & \triangleq bvextract(e, w - 1, 0)
\end{align*}
\]

Similar to the bit-vector operations $+\nu$, $-\nu$, and $\times_{\nu}$ extended with zero extension in Section 2, for $e \in \{bvadd, bvsub, bvmul\}$, we define their extended versions $\equiv^*$. For example, $bvadd^*(e_1, e_2) \equiv bvadd(bvzerextend(e_1, w), zero\_extend(e_2, w))$ and $bvadd^*(e_1, e_2, e_3) \equiv bvadd(bvadd^*(e_1, w), zero\_extend(e_2, w))$.

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 $e$.

\[
width(bvconst(n, b)) = n \\
width(v) = w \\
width(bvadd(e_1, e_2)) = \max(width(e_1), width(e_2)) \\
width(bvsub(e_1, e_2)) = \min(width(e_1), width(e_2)) \\
width(bvmul(e_1, e_2)) = \max(width(e_1), width(e_2)) \\
width(concat(e_1, e_2)) = width(e_1) + j \cdot width(e_2) \\
width(zero\_extend(e_1, i)) = width(e_1) + i \\
width(bvshl(e_1, e_2)) = width(e_1) \\
width(bvshr(e_1, e_2)) = width(e_1) \\
width(bvextract(e, i, j)) = i - j \cdot width(e')
\]

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 $bvadd(e_1, e_2)$, a subtraction $bvsub(e_1, e_2)$, or a multiplication $bvmul(e_1, e_2)$ with $width(e_1) = width(e_2)$ and both $e_1$ and $e_2$ well-formed, or (3) an extraction $bvextract(e', i, j)$ with $0 \leq j < i < width(e')$ and $e'$ well-formed. A predicate $q \in qPred$ is well-formed if all subexpressions are well-formed.

Let $v \in bStmt$ be a state. Define the semantic function $\llbracket e \rrbracket(v)$ for well-formed expressions $e \in qExp$. For a predicate $q \in qPred$, we write $\exists v \models q[v]$ if $e$ evaluates to $tt$ using the evaluation function $\llbracket e \rrbracket(v)$ for every subexpression $e$ in $q$, using $\llbracket v \rrbracket$ for $bvul$, and using $\llbracket \nu \rrbracket$ for $bvu$.

\[
\begin{align*}
\llbracket bvconst(n, b) \rrbracket(v) & \triangleq b \\
\llbracket v \rrbracket(v) & \triangleq \llbracket z \rrbracket(v) \\
\llbracket bvadd(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) + \llbracket e_2 \rrbracket(v) \\
\llbracket bvsub(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) - \llbracket e_2 \rrbracket(v) \\
\llbracket bvmul(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) \times \llbracket e_2 \rrbracket(v) \\
\llbracket concat(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) \llbracket e_2 \rrbracket(v) \\
\llbracket zero\_extend(e_1, i) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) \times i & i \\
\llbracket bvshl(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) \times \llbracket e_2 \rrbracket(v) \\
\llbracket bvshr(e_1, e_2) \rrbracket(v) & \triangleq \llbracket e_1 \rrbracket(v) \times \llbracket e_2 \rrbracket(v) \\
\llbracket bvextract(e, i, j) \rrbracket(v) & \triangleq \llbracket e \rrbracket(v)[i, j]
\end{align*}
\]

Let $q_r, q'_r \in bPred$ be two range predicates and $p \in bProg$ a well-formed program in static single assignment form. Both an safety check $\llbracket \forall \nu \models q_r \rrbracket \implies \mathit{ProjSafe}(p, v) = tt$ for all $v \in bStmt$ and a range problem $\llbracket \forall (q_r, p) \models \nu \rrbracket \implies \mathit{ProjQFBV}(p)$ involves 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 $q_r$, $q'_r$, and the safety check to QF_BV formulas.

Define $\bar{v}$ as $v$ when the atom $a$ is a variable $v$ and otherwise $bvconst(w, b)$ when $a$ is a constant $b$. The function $\mathit{StmtQFBV}$ (Algorithm 7) transforms a statement in $bStmt$ to a QF_BV formula. Recursively define the function $\mathit{ProjQFBV}$ for programs in $bProg$ such that $\mathit{ProjQFBV}(e) \equiv \top$ and $\mathit{ProjQFBV}(s; p) \equiv \mathit{StmtQFBV}(s) \land \mathit{ProjQFBV}(p)$. Note that the formulas returned by $\mathit{StmtQFBV}$ and $\mathit{ProjQFBV}$ are well-formed QF_BV formulas. The following theorem states that $\mathit{ProjQFBV}(p)$ 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 $v, v' \in bStmt$, $v \models p \land v' \models p$ implies $\forall \nu \models \mathit{ProjQFBV}(p)[v']$.

**Algorithm 7 Transformation from bStmt to qPred**

1. function $\mathit{StmtQFBV}(s)$
2. 
3. match $s$ with
4. 
5. case $u \leftarrow a: \text{return } v = \bar{a}$
6. 
7. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
8. 
9. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
10. 
11. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
12. 
13. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
14. 
15. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
16. 
17. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
18. 
19. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
20. 
21. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
22. 
23. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
24. 
25. case $u \leftarrow a + a: \text{return } v = bvadd(\bar{a}, \bar{a})$
26. 
27. end function

For the transformation from range predicates to QF_BV formulas, recursively define a function $\mathit{PredQFBV}$ such that $\mathit{PredQFBV}(\top)$
Theorem 5.2. Let q ∈ bPreds be a range predicate. Then, for all v ∈ bSt, \( \forall v \models q[v] \) if and only if \( \forall \mathcal{V} \models \text{ProgSafeQFBV}(q)[v] \).

Define a function \( \text{StmtSafeQFBV} \) (Algorithm 8) which transforms safety checks for statements to QF_BV. Recursively define a function \( \text{ProgSafeQFBV} \) such that \( \text{ProgSafeQFBV}(v) \equiv \top \) and \( \text{ProgSafeQFBV}(s; p) \equiv \text{StmtSafeQFBV}(s) \land \text{ProgSafeQFBV}(p) \). 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 assignment form,

- \( \forall \mathcal{V} \models q_r[v] \) implies \( \text{ProgSafeQFBV}(p; v) = \top \) for all \( v \in bSt \) if \( (\text{ProgSafeQFBV}(q_r) \land \text{ProgSafeQFBV}(p)) \Rightarrow \text{ProgSafeQFBV}(p) \) is valid, and
- \( \models (q_r; p; q'_r) \) if the QF_BV formula \( \text{ProgSafeQFBV}(q_r) \land \text{ProgSafeQFBV}(p) \) \( \Rightarrow \text{ProgSafeQFBV}(q'_r) \) is valid.

\[
\begin{align*}
\text{Algorithm 8: Transformation from Safety Checks to QF_BV} \\
&\text{function } \text{StmtSafeQFBV}(s) \text{ with} \\
&\begin{align*}
o &\leftarrow \text{bvconst}(w, \text{bv}\mathcal{V}(0)) \\
\text{match } s \text{ with} \\
&\begin{cases} \\
&\text{\text{case } } v \leftarrow a: \text{return } \top \\
&\text{case } v \leftarrow a_1 + a_2: \text{return } \text{bvhigh(bvadd}(w, \text{bv}\mathcal{V}(0))) \\
&\text{case } v \leftarrow a_1 \cdot a_2: \text{return } \text{bvhigh(bvmul}(w, \text{bv}\mathcal{V}(0))) \\
&\text{case } v \leftarrow a_1 + a_2 + y: \text{return } \text{bvhigh(bvadd}(w, \text{bv}\mathcal{V}(0))) \\
&\text{case } v \leftarrow \text{bfunc}(n): \text{return } \text{bvconst}(w, \text{bv}\mathcal{V}(0)) \\
&\text{case } v \leftarrow \text{bvfunc}(n, m): \text{return } \text{bvconst}(w, \text{bv}\mathcal{V}(0)) \\
\end{cases}
\end{align*}
\end{align*}
\]
We evaluate our techniques in real-world low-level mathematical
h'(\vec{x}) \in (m(\vec{x}), e_i(\vec{x}) - e'_i(\vec{x}))_{i \in [l]} for (2). If so, there are u, u_i(\vec{x}) \in
\mathbb{Z}[\vec{x}] such that

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

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

Two Coq tactics are available to find formal proofs for the polyno-
mial equation entailment problems \([28, 29]\). The tactic nsatz proves the entailment problem of the form in (1); the tactic gathir 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 \( x + 2^f y = e \) (from Split statements).
Such polynomial equations can safely be removed after every occurrences
of \( x \) are replaced with \( e \) or \( e - 2^f y \) 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 \( y^2 = x^3 + 486662 x^2 + x \) is over the Galois field \( \mathbb{K} = \mathbb{GF}(\varrho) \)
with \( \varrho = 2^{255} - 19 \). To make the field explicit, we rewrite its defini-
tion as:

\[ y \in \mathbb{K} \quad y = \mathbb{K} \cdot x \cdot \mathbb{K} \cdot x + \mathbb{K} \cdot 486662 \cdot \mathbb{K} \cdot x + \mathbb{K} \cdot x. \quad (5) \]

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
\( \mathbb{G} = (\mathbb{G}, +, \cdot, 0) \) with \( \mathbb{G} = \{(x, y) : x, y \text{ satisfying (5)}\} \). Let \( P_0 = (x_0, y_0) \), \( P_1 = (x_1, y_1) \) \( \in \mathbb{G}. \) We have \(-P_0 = (x_0, -y_0)\) and \( P_0 +_\mathbb{G} P_1 = (x, y) \) where

\[ m = (y_1 - \mathbb{K} \cdot y_0) \div \mathbb{K} \cdot (x_1 - \mathbb{K} \cdot x_0) \]
\[ x = m \cdot \mathbb{K} \cdot m - \mathbb{K} \cdot 486662 - \mathbb{K} \cdot x_0 - \mathbb{K} \cdot x_1 \]
\[ y = (2 \cdot \mathbb{K} \cdot x_0 + \mathbb{K} \cdot x_1 + \mathbb{K} \cdot 486662) \cdot m - \mathbb{K} \cdot m \cdot \mathbb{K} \cdot m \cdot \mathbb{K} \cdot y_0 \]

when \( P_0 \neq +P_1 \). Other cases \((P_0 = +P_1)\) 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 naive 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 \( c \cdot \mathbb{K} \cdot 2^{255} \) is immediately replaced by \( c \cdot \mathbb{K} \) since \( 2^{255} \equiv \mathbb{K} \) 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 Boole-
tor to verify if the input program is safe. The column “range” shows the
time used by Booletor 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 counter-
example. 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.
### 6.2 The Montgomery Ladderstep

Recall that X25519 is based on the Abelian group \( \mathbb{G} = (G, +, 0_G) \) induced by the curve Curve25519. As aforementioned, the binary operation \(+_G\) requires another sequence of arithmetic computation over \( \mathbb{F}(q) \). Errors could still be introduced or even implanted in any sequence of computation proclaimed to implement \(+_G\). 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 \( P +_G \cdots +_G P \) for \( P \in G \) 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 3 Montgomery Ladderstep

1. \( \text{function Ladderstep}(x_1, x_m, z_m, x_{m+1}, z_{m+1}) \)

2. \( t_1 \leftarrow x_m +_G z_m \)
3. \( t_2 \leftarrow x_m -_G z_m \)
4. \( t_7 \leftarrow t_2 \cdot 2 \)
5. \( t_6 \leftarrow t_1 \cdot t_1 \)
6. \( t_5 \leftarrow t_6 - t_7 \)
7. \( t_3 \leftarrow x_m \cdot t_5 +_G z_m \cdot t_5 \)
8. \( t_4 \leftarrow x_m \cdot t_3 +_G z_m \cdot t_3 \)
9. \( t_9 \leftarrow t_3 \cdot t_2 \)
10. \( t_8 \leftarrow t_4 \cdot t_2 \)
11. \( x_{m+1} \leftarrow t_8 +_G t_9 \)
12. \( z_{m+1} \leftarrow t_5 - t_9 \)
13. \( x_{m+1} \leftarrow x_{m+1} +_G x_m \cdot t_1 \)
14. \( z_{m+1} \leftarrow z_{m+1} +_G z_m \cdot t_1 \)
15. \( x_{m+1} \leftarrow x_{m+1} +_G x_m \cdot t_1 \)
16. \( z_{m+1} \leftarrow z_{m+1} +_G z_m \cdot t_1 \)
17. \( z_{m+1} \leftarrow 121666x_m +_G z_m \cdot t_1 \)
18. \( z_m \leftarrow 121666x_m +_G z_m \cdot t_1 \)
19. \( z_{m+1} \leftarrow z_{m+1} +_G z_m \cdot t_1 \)
20. \( \text{return} (x_{m+1}, z_{m+1}, x_m, z_m) \)
21. \( \text{end function} \)

#### Table 1: Certified Verification of Arithmetic Operations over \( \mathbb{F}(q) \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of Lines</th>
<th>Time (seconds)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safe</td>
<td>Range</td>
<td>Without Heuristics</td>
</tr>
<tr>
<td></td>
<td>Algebraic</td>
<td>Total</td>
<td>Algebraic</td>
</tr>
<tr>
<td>Addition</td>
<td>10</td>
<td>0.162</td>
<td>0.249</td>
</tr>
<tr>
<td>Subtraction</td>
<td>15</td>
<td>0.140</td>
<td>0.389</td>
</tr>
<tr>
<td>Multiplication</td>
<td>144</td>
<td>3.904</td>
<td>41.070</td>
</tr>
<tr>
<td>Multiplication by 121666</td>
<td>26</td>
<td>0.266</td>
<td>0.852</td>
</tr>
<tr>
<td>Square</td>
<td>109</td>
<td>3.722</td>
<td>19.905</td>
</tr>
</tbody>
</table>

For the range specification, the unsigned value of each limb used to represent an output \( x_m, z_m, x_{m+1}, z_{m+1} \), or \( z_{m+1} \) must be in the range from 0 to \( 2^{31} + 2^{15} \). In our experiment, we replace all arithmetical 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, 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 will be 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


[²] Amazingly, 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.
A APPENDIX

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

The following \texttt{bcCRYPTOline} code implements multiplications over \( \mathbb{G}(2^{255} - 19) \) for the AMD64 architecture:

\begin{verbatim}
1:  mulrax ← x3;
2:  mulrax ← mulrax × bx4(19);
3:  mulxl319 ← mulrax;
4:  muladx mulrax ← mulrax × y2;
5:  r0 ← mulrax;
6:  mulr01 ← muladx;
7:  mulx ← x4;
8:  mulrax ← mulrax × bx4(19);
9:  mulxl419 ← mulrax;
10: muladx mulrax ← mulrax × y3;
11: carry r0 ← r0 + mulrax;
12: mulr01 ← mulr01 + muladx + carry;
13: mulrax ← x0;
14: muladx mulrax ← mulrax × y0;
15: carry r0 ← r0 + mulrax;
16: mulr01 ← mulr01 + muladx + carry;
17: mulrax ← x0;
18: muladx mulrax ← mulrax × y1;
19: r1 ← mulrax;
20: mulr11 ← muladx;
21: mulrax ← x0;
22: muladx mulrax ← mulrax × y2;
23: r2 ← mulrax;
24: mulr21 ← muladx;
25: mulrax ← x0;
26: muladx mulrax ← mulrax × y3;
27: r3 ← mulrax;
28: mulr31 ← muladx;
29: mulrax ← x0;
30: muladx mulrax ← mulrax × y4;
31: r4 ← mulrax;
32: mulr41 ← muladx;
33: mulrax ← x1;
34: muladx mulrax ← mulrax × y0;
35: carry r1 ← r1 + mulrax;
36: mulr11 ← mulr11 + muladx + carry;
37: mulrax ← x1;
38: muladx mulrax ← mulrax × y1;
39: carry r2 ← r2 + mulrax;
40: mulr21 ← mulr21 + muladx + carry;
41: mulrax ← x1;
42: muladx mulrax ← mulrax × y2;
43: carry r3 ← r3 + mulrax;
44: mulr31 ← mulr31 + muladx + carry;
45: mulrax ← x1;
46: muladx mulrax ← mulrax × y3;
47: carry r4 ← r4 + mulrax;
48: mulr41 ← mulr41 + muladx + carry;
49: mulrax ← x1;
50: muladx mulrax ← mulrax × bx4(19);
51: mulrax ← mulrax × y4;
52: carry r0 ← r0 + mulrax;
\end{verbatim}
Let bMul denote the above program. Define \( q_a \triangleq T \), \( q_r \triangleq 0 \leq x_0, x_1, x_2, x_3, x_4, y_0, y_1, y_2, y_3, y_4 \leq \text{radix}^{64}(2^{52}) \), \( q_a \triangleq \text{radix}^{51}(x_4, x_3, x_2, x_1, x_0) \times \text{radix}^{51}(y_4, y_3, y_2, y_1, y_0) \approx \text{radix}^{51}(z_4, z_3, z_2, z_1, z_0) \mod \text{radix}^{64}(2^{55} - 19) \), and \( q_r \triangleq 0 \leq z_1, z_2, z_3, z_4 \leq \text{radix}^{64}(2^{52}) \). Its specification is

\[ \langle q_a, q_r \rangle \text{ bMul } \langle q'_a, q'_r \rangle. \]