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

Abstract. Mathematical constructs are necessary for computation on the underlying algebraic structures of cryptosystems. They are often written in assembly languages and optimized manually for efficiency. We develop a certified technique to verify 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. We report our case studies on verifying arithmetic computation over a large finite field and the Montgomery Ladderstep, a crucial loop in X25519.

1 Introduction

In order to take advantages 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 low-level assembly languages and manually optimized for efficiency. Security of cryptosystems could be compromised should programming errors in mathematical constructs be exploited by adversaries. Security guarantees of cryptographic programs thus 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 [29].

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 [22]. 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 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 $\mathbb{GF}(2^{255} - 19)$ [23]. 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. Mathematical constructs by their nature perform computation on underlying algebraic structures. We aim to verify whether they perform intended algebraic computation correctly. To this end, we propose the domain specific language CRYPTOLINE for low-level mathematical constructs. Algebraic pre- and post-conditions of programs in CRYPTOLINE are specified as Hoare triples [21]. Such an algebraic specification is converted to static single assignment form and then translated into an algebraic problem (called the modular polynomial equation entailment problem) [4,20]. We use the computer algebra system SINGULAR to solve the algebraic problem [19]. Program fragments irrelevant to algebraic properties are also removed by slicing to reduce the size of algebraic problems [28]. The proof assistant COQ is used to certify the correctness of translations, as well as solutions to algebraic problems computed by SINGULAR [12].

We report case studies on verifying mathematical constructs used in the X25519 ECDH key exchange protocol [9,10]. For each arithmetic operations (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 CRYPTOLINE 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 CRYPTOLINE 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 verify arithmetic computation over a finite field of order $2^{255} - 19$ and a critical program (the Montgomery Ladderstep) automatically.
– 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 [1,3,2,25,24]. 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]. Re-implementations of mathematical constructs in F* [16] have been verified using a combination of SMT solving and manual proofs. 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 [17]. 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 solver for the algebraic problem is 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 positive and all integers respectively. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ 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 $(F \backslash \{0\}, 1, \times)$ 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 $\{0, \ldots, \varrho\}$ 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 $\boldsymbol{x}$. $\mathcal{R}[\boldsymbol{x}]$ is the set of polynomials over $\boldsymbol{x}$ with coefficients in the ring $\mathcal{R}$. $\mathcal{R}[\boldsymbol{x}]$ is a ring. A set $I \subseteq \mathcal{R}[\boldsymbol{x}]$ is an *ideal* if

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

Given $G \subseteq \mathcal{R}[\boldsymbol{x}]$, $\langle G \rangle$ 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 $f \in \mathcal{R}[\boldsymbol{x}]$.

3 Domain Specific Language – CRYPTOLINE

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 are 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 uses the "evicted" data (to make room for the data of the eavesdropping thread) then has 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 CRYPTOLINE is designed based on the same principles. Conditional branches and indirect memory accesses are not admitted in CRYPTOLINE. A program is but a straight line of variable assignments on expressions. Consider the following syntactic classes:

$$\begin{array}{l} \mathit{Nat} ::= 1 \mid 2 \mid \cdots \qquad \mathit{Int} ::= \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \qquad \mathit{Var} ::= x \mid y \mid z \mid \cdots \\ \mathit{Expr} ::= \mathit{Int} \mid \mathit{Var} \mid -\mathit{Expr} \mid \mathit{Expr}+\mathit{Expr} \mid \mathit{Expr}-\mathit{Expr} \mid \mathit{Expr}\times\mathit{Expr} \mid \mathsf{Pow}(\mathit{Expr}, \mathit{Nat}) \end{array}$$

We allow exact integers as constants in the domain specific language. 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 $\mathsf{Pow}(e, n)$ denotes e^n for any expression e and positive integer n. More formally, let $\mathit{St} \triangleq \mathit{Var} \to \mathbb{Z}$ and $\nu \in \mathit{St}$ be a *state* (or *valuation*). That is, a state ν is a mapping from variables to integers. Define the semantic function $[\![e]\!]_\nu$ as follows.

$$\begin{array}{rr} [\![i]\!]_\nu \triangleq i \text{ for } i \in \mathit{Int} \quad [\![v]\!]_\nu \triangleq \nu(v) \text{ for } v \in \mathit{Var} & [\![-e]\!]_\nu \triangleq -_{\mathbb{Z}} [\![e]\!]_\nu \\ [\![e_0 + e_1]\!]_\nu \triangleq [\![e_0]\!]_\nu +_{\mathbb{Z}} [\![e_1]\!]_\nu & [\![e_0 - e_1]\!]_\nu \triangleq [\![e_0]\!]_\nu -_{\mathbb{Z}} [\![e_1]\!]_\nu \\ [\![e_0 \times e_1]\!]_\nu \triangleq [\![e_0]\!]_\nu \times_{\mathbb{Z}} [\![e_1]\!]_\nu & [\![\mathsf{Pow}(e, n)]\!]_\nu \triangleq ([\![e]\!]_\nu)^{[\![n]\!]_\nu} \end{array}$$

Note that there is no expression for quotients, or bitwise logical operations. Bitwise left shifting however can be modeled by multiplying $\mathsf{Pow}(2, n)$. Although CRYPTOLINE models a (very) small subset of assembly, it suffices to encode low-level mathematical constructs in X25519.

$$\mathit{Stmt} ::= \mathit{Var} \leftarrow \mathit{Expr} \mid [\mathit{Var}, \mathit{Var}] \leftarrow \mathsf{Split}(\mathit{Expr}, \mathit{Nat})$$

In CRYPTOLINE, 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 CRYPTOLINE. The statement $[v_h, v_l] \leftarrow \mathsf{Split}(e, n)$ 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. Define $\nu\{v \mapsto d\}(u) \triangleq \begin{cases} d & \text{if } u = v \\ \nu(u) & \text{otherwise} \end{cases}$. Consider the relation $Tr \subseteq St \times Stmt \times St$ defined by $(\nu, v \leftarrow e, \nu\{v \mapsto [\![e]\!]_\nu\}) \in Tr$, and $(\nu, [v_h, v_l] \leftarrow \mathsf{Split}(e, n), \nu\{v_h \mapsto hi\}\{v_l \mapsto lo\}) \in Tr$ where $hi = ([\![e]\!]_\nu - lo) \div 2^{[\![n]\!]_\nu}$ and $lo = [\![e]\!]_\nu \bmod 2^{[\![n]\!]_\nu}$. Intuitively, $(\nu, s, \nu') \in Tr$ denotes that the state ν transits to the state ν' after executing the statement s.

$$Prog ::= \epsilon \mid Stmt; Prog$$

A *program* is a sequence of statements. We denote the empty program by ϵ. 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 $Tr^* \subseteq St \times Prog \times St$ where $(\nu, \epsilon, \nu) \in Tr^*$ and $(\nu, s; p, \nu'') \in Tr^*$ if there is a ν' with $(\nu, s, \nu') \in Tr$ and $(\nu, p, \nu'') \in Tr^*$. We write $\nu \xRightarrow{p} \nu'$ when $(\nu, p, \nu') \in Tr^*$.

For algebraic specifications, $\top$ denotes the Boolean value tt. We admit polynomial equations $e = e'$ and modular polynomial equations $e \equiv e' \bmod n$ as atomic predicates. A *predicate* $q \in Pred$ is a conjunction of atomic predicates. For $\nu \in St$, write $\mathbb{Z} \models q[\nu]$ if q evaluates to tt after replacing each variable v with $\nu(v)$; ν is called a *q-state*.

$$Pred ::= \top \mid Expr = Expr \mid Expr \equiv Expr \bmod Nat \mid Pred \wedge Pred$$
$$Spec ::= (\!|Pred|\!)\, Prog\, (\!|Pred|\!)$$

We follow Hoare's formalism in algebraic specifications of mathematical constructs [21]. In an algebraic specification $(\!|q|\!)\ p\ (\!|q'|\!)$, q and q' are the *pre-* and *post-conditions* of p respectively. Given $q, q' \in Pred$ and $p \in Prog$, $(\!|q|\!)\ p\ (\!|q'|\!)$ is *valid* (written $\models (\!|q|\!)\ p\ (\!|q'|\!)$) if for every $\nu, \nu' \in St$, $\mathbb{Z} \models q[\nu]$ and $\nu \xRightarrow{p} \nu'$ imply $\mathbb{Z} \models q'[\nu']$. Less formally, $\models (\!|q|\!)\ p\ (\!|q'|\!)$ if executing p from a q-state always results in a q'-state.

1: $r_0 \leftarrow x_0$;	6: $r_0 \leftarrow r_0 + 4503599627370458$;	11: $r_0 \leftarrow r_0 - y_0$;
2: $r_1 \leftarrow x_1$;	7: $r_1 \leftarrow r_1 + 4503599627370494$;	12: $r_1 \leftarrow r_1 - y_1$;
3: $r_2 \leftarrow x_2$;	8: $r_2 \leftarrow r_2 + 4503599627370494$;	13: $r_2 \leftarrow r_2 - y_2$;
4: $r_3 \leftarrow x_3$;	9: $r_3 \leftarrow r_3 + 4503599627370494$;	14: $r_3 \leftarrow r_3 - y_3$;
5: $r_5 \leftarrow x_4$;	10: $r_4 \leftarrow r_4 + 4503599627370494$;	15: $r_4 \leftarrow r_4 - y_4$;

Fig. 1. Subtraction sub

Figure 1 gives a simple yet real implementation of subtraction over $\mathbb{GF}(\varrho)$. In the figure, a number in $\mathbb{GF}(\varrho)$ is represented by five 51-bit non-negative integers. 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. Given $0 \le x_0, x_1, x_2, x_3, x_4, y_0, y_1, y_2, y_3, y_4 < 2^{51}$,

the algebraic specification of the mathematical construct is therefore

$$(\!|\top|\!) \text{ sub } (\!| \begin{array}{c} radix51(x_4, x_3, x_2, x_1, x_0) - radix51(y_4, y_3, y_2, y_1, y_0) \\ \equiv radix51(r_4, r_3, r_2, r_1, r_0) \bmod \varrho \end{array} |\!).$$

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 $2\varrho = radix51$ (4503599627370494, 4503599627370494, 4503599627370494, 4503599627370494, 4503599627370458) after tedious computation. Hence $radix51(r_4, r_3, r_2, r_1, r_0) = radix51(x_4, x_3, x_2, x_1, x_0) + 2\varrho - radix51(y_4, y_3, y_2, y_1, y_0) \equiv radix51(x_4, x_3, x_2, x_1, x_0) - radix51(y_4, y_3, y_2, y_1, y_0) \bmod \varrho$. The program in Figure 1 is correct. 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. Please note that ranges are a complicated issue. The subtraction in Figure 1 gives results that are correct but possibly overflowing ($> \varrho$), which can and must be accounted for later.

4 Translation to Algebraic Problems

Given $q, q' \in Pred$ and $p \in Prog$, we reduce the problem of checking $\models (\!|q|\!)\ p\ (\!|q'|\!)$ to the entailment problem of modular polynomial equations over integer variables. The reduction is carried out by the following three transformations:

1. Program slicing. To improve efficiency, fragments of the program p irrelevant to the post-condition q' are removed by program slicing (Section 4.1) [28].
2. Static single assignments. The sliced program is transformed into static single assignments. Variables in pre- and post-conditions are also renamed (Section 4.2) [4].
3. Modular polynomial equations. Validity of algebraic specifications is reduced to the entailment of modular polynomial equations (Section 4.3) [20].

For each transformation, we give an algorithm and establish the correctness of the algorithm in Coq [12]. Specifically, semantics for our domain specific language and validity of algebraic specifications are formalized. The correctness of transformations is then certified by the proof assistant Coq. For program slicing and static single assignments, we construct machine-checkable proofs for the soundness and completeness of the two transformations. 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 Program Slicing

Consider the problem of checking $\models (\!|q|\!)\ p\ (\!|q'|\!)$ for arbitrary $q, q' \in Pred$ and $p \in Prog$. Since q' is arbitrary, the program p may contain fragments irrelevant to q'. Program slicing is a simple yet effective technique to improve efficiency of verification through simplifying programs [28].

Here we slice programs backwardly. We initialize the set of *cared* variables to be the variables appeared in the post-condition q'. Starting from the last statement, we check if it assigns any cared variables. If so, we keep the statement and update the cared variables; otherwise, the statement is skipped. Then we continue to the previous statement. This process repeats until all statements of the program are examined.

Our program slicing algorithm requires several auxiliary functions. Algorithm 1 shows how to compute variables in an expression by structural induction. If the given expression is an integer, the empty set is returned; if it is a variable, the singleton with the variable is returned. Variables in other expressions are computed recursively.

Algorithm 1 Variables Occurred in Expressions

```
1:  function VarsInExpr(e)
2:      match e with
3:          case i: return ∅
4:          case v: return {v}
5:          case −e′: return VarsInExpr(e′)
6:          case e′ + e″: return VarsInExpr(e′) ∪ VarsInExpr(e″)
7:          case e′ − e″: return VarsInExpr(e′) ∪ VarsInExpr(e″)
8:          case e′ × e″: return VarsInExpr(e′) ∪ VarsInExpr(e″)
9:          case Pow(e′, _): return VarsInExpr(e′)
10: end function
```

Algorithm 2 Variables Occurred in Predicates

```
1:  function VarsInPred(q)
2:      match q with
3:          case ⊤: return ∅
4:          case e′ = e″: return VarsInExpr(e′) ∪ VarsInExpr(e″)
5:          case e′ ≡ e″ mod _: return VarsInExpr(e′) ∪ VarsInExpr(e″)
6:          case q′ ∧ q″: return VarsInPred(q′) ∪ VarsInPred(q″)
7:  end function
```

Similarly, Algorithm 2 computes the variables in a predicate. Using the Coq specification language Gallina, both algorithms are specified in the proof assistant.

To slice a statement, we check if the assigned variables are cared variables. If they are not, we leave the cared variables intact. If the assigned variables are cared variables, we update the cared variables by excluding the assigned variables but including the variables in the expression (Algorithm 3). In the algorithm, the parameter $vars$ denotes the set of cared variables. If the given statement assigns to a cared variable, it is returned with the updated cared variables. Otherwise, only cared variables are returned.

To slice a program, our algorithm proceeds from the last statement (Algorithm 4). It invokes SliceStmt to see if the current statement is relevant and update cared variables. The algorithm recurses with updated cared variables and remaining statements.

Algorithm 3 Slicing Statements

```
1: function SliceStmt(vars, s)
2:     match s with
3:         case v ← e:
4:             if v ∈ vars then return ⟨VarsInExpr(e) ∪ (vars \ {v}), s⟩ else return vars
5:         case [v_h, v_l] ← Split(e, _):
6:             if v_h or v_l ∈ vars then return ⟨VarsInExpr(e) ∪ (vars \ {v_h, v_l}), s⟩ else
   return vars
7: end function
```

Algorithm 4 Slicing Programs

```
1: function SliceProg(vars, p)
2:     match p with
3:         case ϵ: return ϵ
4:         case pp; s:
5:             match SliceStmt(vars, s) with
6:                 case vars′: return SliceProg(vars′, pp)
7:                 case ⟨vars′, s′⟩: return SliceProg(vars′, pp); s′
8: end function
```

To avoid ambiguity, we consider *well-formed* programs where

- for every statement $[v_h, v_l] \leftarrow \mathsf{Split}(e, n)$ in the program, $v_h \neq v_l$; and
- every non-input program variable must be assigned to a value before being used.

Algorithms 3 and 4 are specified in Gallina. Their properties are also specified and proven in Coq. Particularly, well-formedness is preserved by program slicing.

Lemma 1. SliceProg(VarsInPred(q'), p) *is well-formed for every* $q' \in Pred$ *and well-formed program* $p \in Prog$.

Moreover, the soundness and completeness of our program slicing algorithm have been certified. Formally, we have the following theorem:

Theorem 1. *For every* $q, q' \in Pred$ *and* $p \in Prog$,

$$\models (\!|q|\!)\ p\ (\!|q'|\!) \text{ if and only if } \models (\!|q|\!)\ \textsc{SliceProg}(\textsc{VarsInPred}(q'), p)\ (\!|q'|\!).$$

4.2 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 algebraic specification $(\!|q|\!)\ p\ (\!|q'|\!)$ to an algebraic specification of p in static single assignment form for any $q, q' \in Pred$ and $p \in Prog$. 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. Only the most recent copies of variables are referred in expressions. Algorithm 5 transforms expressions with the finite mapping θ

Algorithm 5 Static Single Assignment Transformation for Expressions

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

by structural induction. Integers are unchanged. For each variable, its most recent copy is returned by looking up the mapping θ. Other expressions are transformed recursively.

Similarly, 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.

Statement transformation is slightly more complicated (Algorithm 6). For expressions on the right hand side, they are transformed by the given finite mapping θ. The algorithm then updates θ and replaces assigned variables with their latest copies.

Algorithm 6 Static Single Assignment Transformation for Statements

function SSASTMT(θ, s)
  match s **with**
    case $v \leftarrow e$:
      $\theta' \leftarrow \theta\{v \mapsto \theta(v) + 1\}$
      return $\langle \theta', v^{\theta'(v)} \leftarrow$ SSAEXPR(θ, e)$\rangle$
    case $[v_h, v_l] \leftarrow$ Split(e, n):
      $\theta_h \leftarrow \theta\{v_h \mapsto \theta(v_h) + 1\}$
      $\theta_l \leftarrow \theta_h\{v_l \mapsto \theta_h(v_l) + 1\}$
      return $\langle \theta_l, [v_h^{\theta_h(v_h)}, v_l^{\theta_l(v_l)}] \leftarrow$ Split(SSAEXPR(θ, e), n)$\rangle$
end function

It is straightforward to transform programs to static single assignment form (Algorithm 7). Using the initial mapping 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.

Using the specifications of Algorithm 6 and 7 in GALLINA, properties of these algorithms are formally proven in COQ. We first show that Algorithm 7 preserves well-formedness and produces a program in static single assignment form.

Lemma 2. *Let $\theta_0(v) = 0$ for every $v \in$ Var and $p \in$ Prog a well-formed program. If $\langle \hat{\theta}, \hat{p} \rangle =$ SSAPROG(θ_0, p), then $\hat{p}$ is well-formed and in static single assignment form.*

Algorithm 7 Static Single Assignment for Programs

```
1: function SSAPROG(θ, p)
2:     match p with
3:         case ϵ: return ⟨θ, ϵ⟩
4:         case s; pp:
5:             ⟨θ′, s′⟩ ← SSASTMT(θ, s)
6:             ⟨θ″, pp″⟩ ← SSAPROG(θ′, pp)
7:             return ⟨θ″, s′; pp″⟩
8: end function
```

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

Theorem 2. *Let $\theta_0(v) = 0$ for every $v \in Var$. For every $q, q' \in Pred$ and $p \in Prog$,*

$$\models (\!|q|\!)\ p\ (\!|q'|\!) \text{ if and only if } \models (\!|\text{SSAPRED}(\theta_0, q)|\!)\ \hat{p}\ (\!|\text{SSAPRED}(\hat{\theta}, q')|\!)$$

where $\langle \hat{\theta}, \hat{p} \rangle = \text{SSAPROG}(\theta_0, p)$.

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

Fig. 2. Subtraction subSSA in Static Single Assignment Form

Example. Figure 2 gives the subtraction program sub 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 $0 \leq i \leq 4$. Subsequently, variables in pre- and post-conditions of the algebraic specification for subtraction needs to be indexed. The corresponding algebraic specification is as follows.

$$(\!|\top|\!)\ \mathsf{subSSA}\ (\!|\begin{array}{c} radix51(x_4^0, x_3^0, x_2^0, x_1^0, x_0^0) - radix51(y_4^0, y_3^0, y_2^0, y_1^0, y_0^0) \\ \equiv radix51(r_4^3, r_3^3, r_2^3, r_1^3, r_0^3) \bmod \varrho \end{array}|\!).$$

4.3 Modular Polynomial Equation Entailment

The last step transforms any algebraic program specification to the modular polynomial equation entailment problem. For $q \in Pred$, we write $q(\boldsymbol{x})$ to signify the free

variables $\boldsymbol{x}$ occurred in q. Given $q(\boldsymbol{x}), q'(\boldsymbol{x}) \in \mathit{Pred}$, the *modular polynomial equation entailment* problem decides whether $q(\boldsymbol{x}) \implies q'(\boldsymbol{x})$ holds when $\boldsymbol{x}$ are substituted for arbitrary integers. That is, we want to check if $q(\boldsymbol{x}) \implies q'(\boldsymbol{x})$ evaluates to tt after each variable x is replaced by $\nu(x)$ for every valuation $\nu \in \mathit{St}$. We write $\mathbb{Z} \models \forall \boldsymbol{x}.q(\boldsymbol{x}) \implies q'(\boldsymbol{x})$ if it is indeed the case.

Programs in static single assignment form can easily be transformed to conjunctions of polynomial equations. An assignment statement is translated to a polynomial equation with a variable on the left hand side. A Split statement is transformed to an equation with a linear expression of the assigned variables on the left hand side (Algorithm 8).

Algorithm 8 Polynomial Equation Transformation for Statements

```
1: function StmtToPolyEQ(s)
2:     match s with
3:         case v ← e: return v = e
4:         case [v_h, v_l] ← Split(e, n): return v_l + 2^n v_h = e
5: end function
```

A program in static single assignment form is transformed to the conjunction of polynomial equations corresponding to its statements (Algorithm 9).

Algorithm 9 Polynomial Equation Transformation for Programs

```
1: function ProgToPolyEQ(p)
2:     match p with
3:         case ϵ: return ⊤
4:         case s; pp: return StmtToPolyEQ(s) ∧ ProgToPolyEQ(pp)
5: end function
```

Algorithm 8 and 9 are specified straightforwardly in Gallina. We use the proof assistant Coq to prove properties about the algorithms. Note that $\textsc{ProgToPolyEQ}(p) \in \mathit{Pred}$ for every $p \in \mathit{Prog}$. The following theorem shows that any behavior of the program p is a solution to the system of polynomial equations $\textsc{ProgToPolyEQ}(p)$. In other words, $\textsc{ProgToPolyEQ}(p)$ gives an abstraction of the program p.

Theorem 3. *Let $p \in \mathit{Prog}$ be a well-formed program in static single assignment form. For every $\nu, \nu' \in \mathit{St}$ with $\nu \overset{p}{\Longrightarrow} \nu'$, we have $\mathbb{Z} \models \textsc{ProgToPolyEQ}(p)[\nu']$.*

Definition 1 gives the modular polynomial equation entailment problem corresponding to an algebraic program specification.

Definition 1. *For $q, q' \in \mathit{Pred}$ and $p \in \mathit{Prog}$ in static single assignment form, define*

$$\mathit{\Pi}(\langle\!|q|\!\rangle\ p\ \langle\!|q'|\!\rangle) \triangleq q(\boldsymbol{x}) \wedge \varphi(\boldsymbol{x}) \implies q'(\boldsymbol{x})$$

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

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

Fig. 3. Modular Polynomial Equation Entailment for subSSA

Example. The modular polynomial equation entailment problem corresponding to the algebraic specification of subtraction is shown in Figure 3. The problem has 15 polynomial equality constraints with 25 variables. We want to know if $\mathit{radix51}(r_4^3, r_3^3, r_2^3, r_1^3, r_0^3)$ is the difference between $\mathit{radix51}(x_4^0, x_3^0, x_2^0, x_1^0, x_0^0)$ and $\mathit{radix51}(y_4^0, y_3^0, y_2^0, y_1^0, y_0^0)$ in $\mathbb{GF}(\varrho)$ under the constraints.

The soundness of Algorithm 9 is certified in Coq (Theorem 4).

Theorem 4. *Let $q, q' \in \mathit{Pred}$ be predicates, and $p \in \mathit{Prog}$ a well-formed program in static single assignment form. If $\mathbb{Z} \models \forall \boldsymbol{x}. \Pi(\langle\!|q|\!\rangle\ p\ \langle\!|q'|\!\rangle)$, then $\models \langle\!|q|\!\rangle\ p\ \langle\!|q'|\!\rangle$.*

Summary of Translation Consider any predicates $q, q' \in \mathit{Pred}$ and well-formed program $p \in \mathit{Prog}$. Let $\theta_0(v) = 0$ for every $v \in \mathit{Var}$. By Theorem 1, 2, and 4, we have

$$\begin{array}{ll} & \models \langle\!|q|\!\rangle\ p\ \langle\!|q'|\!\rangle \\ \Leftrightarrow & \models \langle\!|q|\!\rangle\ \textsc{SliceProg}(\textsc{VarsInPred}(q'), p)\ \langle\!|q'|\!\rangle \qquad \text{(Theorem 1)} \\ \Leftrightarrow & \models \langle\!|\textsc{SSAPred}(\theta_0, q)|\!\rangle\ \hat{p}\ \langle\!|\textsc{SSAPred}(\hat{\theta}, q')|\!\rangle \qquad \text{(Theorem 2)} \\ & \text{where } \langle \hat{\theta}, \hat{p} \rangle = \textsc{SSAProg}(\theta_0, \textsc{SliceProg}(\textsc{VarsInPred}(q'), p)) \\ \Leftarrow & \mathbb{Z} \models \forall \boldsymbol{x}. \Pi(\langle\!|\textsc{SSAPred}(\theta_0, q)|\!\rangle\ \hat{p}\ \langle\!|\textsc{SSAPred}(\hat{\theta}, q')|\!\rangle) \qquad \text{(Theorem 4)} \end{array}$$

Observe that $\hat{p}$ is well-formed and in static single assignment form (Lemma 1 and 2). Theorem 4 is applicable in the last deduction. After the translations, an instance of the modular polynomial equation entailment problem is obtained from the given algebraic specification of a well-formed program in CryptoLine. To verify mathematical constructs against their algebraic specifications, we will solve the entailment problem.

5 Solving Modular Polynomial Equation Entailment Problem

It remains to show

$$\mathbb{Z} \models \forall \boldsymbol{x}. \bigwedge_{i \in [I]} e_i(\boldsymbol{x}) = e_i'(\boldsymbol{x}) \wedge \bigwedge_{j \in [J]} f_j(\boldsymbol{x}) \equiv f_j'(\boldsymbol{x}) \bmod n_j \implies \bigwedge_{k \in [K]} g_k(\boldsymbol{x}) = g_k'(\boldsymbol{x}) \wedge \bigwedge_{l \in [L]} h_l(\boldsymbol{x}) \equiv h_l'(\boldsymbol{x}) \bmod m_l$$

where $e_i(\boldsymbol{x}), e_i'(\boldsymbol{x}), f_j(\boldsymbol{x}), f_j'(\boldsymbol{x}), g_k(\boldsymbol{x}), g_k'(\boldsymbol{x}), h_l(\boldsymbol{x}), h_l'(\boldsymbol{x}) \in \mathbb{Z}[\boldsymbol{x}]$, $n_j, m_l \in \mathbb{N}$ 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

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

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

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

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

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

which in turn is equivalent to

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

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

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

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

where $e_i(\boldsymbol{x}), e'_i(\boldsymbol{x}), g(\boldsymbol{x}), g'(\boldsymbol{x}), h(\boldsymbol{x}), h'(\boldsymbol{x}) \in \mathbb{Z}[\boldsymbol{x}]$ for $i \in [I]$ and $m \in \mathbb{N}$ [20].

We solve the polynomial equation entailment problems (1) and (2) via the ideal membership problem [20,11]. For (1), consider the ideal $I = \langle e_i(\boldsymbol{x}) - e'_i(\boldsymbol{x})\rangle_{i\in[I]}$. Suppose $g(\boldsymbol{x}) - g'(\boldsymbol{x}) \in I$. Then there are $u_i(\boldsymbol{x}) \in \mathbb{Z}[\boldsymbol{x}]$ (called *coefficients*) such that

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

Hence $g(\boldsymbol{x}) - g'(\boldsymbol{x}) = 0$ follows from the polynomial equations $e_i(\boldsymbol{x}) = e'_i(\boldsymbol{x})$ for $i \in [I]$. Similarly, it suffices to check if $h(\boldsymbol{x}) - h'(\boldsymbol{x}) \in \langle m, e_i(\boldsymbol{x}) - e'_i(\boldsymbol{x})\rangle_{i\in[I]}$ for (2). If so, there are $u, u_i(\boldsymbol{x}) \in \mathbb{Z}[\boldsymbol{x}]$ such that

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

Thus $h(\boldsymbol{x}) \equiv h'(\boldsymbol{x}) \bmod m$ 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 $x^2 + x \notin \langle 2\rangle$ [20].

Two Coq tactics are available to find formal proofs for the polynomial equation entailment problems [26,27]. The tactic nsatz proves the entailment problem of the form in (1); the tactic gbarith is able to prove 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 [18]. 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 9 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 [19]. 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:

$$y \cdot_{\mathbb{K}} y =_{\mathbb{K}} x \cdot_{\mathbb{K}} x \cdot_{\mathbb{K}} x +_{\mathbb{K}} 486662 \cdot_{\mathbb{K}} x \cdot_{\mathbb{K}} x +_{\mathbb{K}} x. \tag{5}$$

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

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

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 1 (Section 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 properties, and performed certified verification with our technique. Table 1 summarizes the results without and with applying the two heuristics in Section 5. 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.

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

	number of lines	time (seconds)			remark
		without	with	range	
addition	10	45.40	3.24	0.49	$a +_{\mathbb{K}} b$
subtraction	15	52.18	3.65	0.83	$a -_{\mathbb{K}} b$
multiplication	169	-	24.98	19.82	$a \cdot_{\mathbb{K}} b$
multiplication by 121666	21	110.39	3.87	0.62	$121666 \cdot_{\mathbb{K}} a$
square	124	-	13.08	14.33	$a \cdot_{\mathbb{K}} a$

We also perform range checks with the SMT solver BOOLECTOR on the five mathematical constructs in Table 1 [13]. The most complicated case (multiplication) takes 19.82 seconds. Although our range check is not yet certified, low-level mathematical constructs in X25519 are formally verified automatically.

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.

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

Algorithm 10 Montgomery Ladderstep

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

2: $t_1 \leftarrow x_m +_{\mathbb{K}} z_m$
3: $t_2 \leftarrow x_m -_{\mathbb{K}} z_m$
4: $t_7 \leftarrow t_2 \cdot_{\mathbb{K}} t_2$
5: $t_6 \leftarrow t_1 \cdot_{\mathbb{K}} t_1$
6: $t_5 \leftarrow t_6 -_{\mathbb{K}} t_7$
7: $t_3 \leftarrow x_{m+1} +_{\mathbb{K}} z_{m+1}$
8: $t_4 \leftarrow x_{m+1} -_{\mathbb{K}} z_{m+1}$
9: $t_9 \leftarrow t_3 \cdot_{\mathbb{K}} t_2$
10: $t_8 \leftarrow t_4 \cdot_{\mathbb{K}} t_1$
11: $x_{m+1} \leftarrow t_8 +_{\mathbb{K}} t_9$
12: $z_{m+1} \leftarrow t_8 -_{\mathbb{K}} t_9$
13: $x_{m+1} \leftarrow x_{m+1} \cdot_{\mathbb{K}} x_{m+1}$
14: $z_{m+1} \leftarrow z_{m+1} \cdot_{\mathbb{K}} z_{m+1}$
15: $z_{m+1} \leftarrow z_{m+1} \cdot_{\mathbb{K}} x_1$
16: $x_m \leftarrow t_6 \cdot_{\mathbb{K}} t_7$
17: $z_m \leftarrow 121666 \cdot_{\mathbb{K}} t_5$
18: $z_m \leftarrow z_m +_{\mathbb{K}} t_7$
19: $z_m \leftarrow z_m \cdot_{\mathbb{K}} t_5$
20: **return** $(x_m, z_m, x_{m+1}, z_{m+1})$
21: **end function**

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 10).

Let unprimed and primed variables denote their values before and after computation respectively. The Montgomery Ladderstep has the following specification [23]:[1]

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

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 CRYPTOLINE, translate the above specification into an algebraic specification, and apply our technique to verify the Ladderstep (containing 1462 statements). The verification takes 82 hours. The SMT solver finishes range checks on the Ladderstep in 19 minutes. For production releases of low-level mathematical constructs, we believe 3.5 days in verification time are well invested.

7 Conclusion

We have developed techniques to verify algebraic 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 certified techniques for range checks. We also plan to apply our techniques to more low-level mathematical constructs in real cryptographic programs.

[1] Amusingly, we find for ourselves the factor of 4 in both the numerator and denominator of the addition formulas during verification, noted on [23, p. 261].

References

1. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innovations in Systems and Software Engineering 9(2), 59–77 (2013) 3
2. Affeldt, R., Marti, N.: An approach to formal verification of arithmetic functions in assembly. In: Okada, M., Satoh, I. (eds.) Advances in Computer Science. LNCS, vol. 4435, pp. 346–360. Springer (2007) 3
3. Affeldt, R., Nowak, D., Yamada, K.: Certifying assembly with formal security proofs: The case of BBS. Science of Computer Programming 77(10–11), 1058–1074 (2012) 3
4. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL. pp. 1–11. ACM, New York, NY, USA (1988) 2, 6, 8
5. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Transactions on Programming Languages and Systems 37(2), 7:1–7:31 (2015) 3
6. Beringer, L., Petcher, A., Ye, K.Q., Appel, A.W.: Verified correctness and security of openssl HMAC. In: USENIX Security Symposium 2015. pp. 207–221. USENIX Association (2015) 3
7. Bernstein, D.J.: Cache timing attacks on AES (2005), https://cr.yp.to/antiforgery/cachetiming-20050414.pdf 4
8. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography. LNCS, vol. 3958, pp. 207–228. Springer (2006) 1
9. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. In: Preneel, B., Takagi, T. (eds.) CHES. LNCS, vol. 6917, pp. 124–142. Springer (2011) 2, 15
10. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering 2(2), 77–89 (2012) 2, 15
11. Bernstein, D.J., Schwabe, P.: gfverif: Fast and easy verification of finite-field arithmetic (2016), http://gfverif.cryptojedi.org 3, 13
12. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004) 2, 6
13. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS. LNCS, vol. 5505, pp. 174–177. Springer (2009) 15
14. Chen, Y.F., Hsu, C.H., Lin, H.H., Schwabe, P., Tsai, M.H., Wang, B.Y., Yang, B.Y., Yang, S.Y.: Verifying Curve25519 software. In: CCS. pp. 299–309. ACM (2014) 3
15. Cohen, H.: A Course in Computational Algebraic Number Theory, GTM, vol. 138. Springer, 3rd edn. (1996) 13, 14
16. Project Everest, https://project-everest.github.io 3
17. Fiat-crypto, https://github.com/mit-plv/fiat-crypto 3
18. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and company (1979) 13
19. Greuel, G.M., Pfister, G.: A Singular Introduction to Commutative Algebra. Springer, 2nd edn. (2008) 2, 14
20. Harrison, J.: Automating elementary number-theoretic proofs using Gröbner bases. In: Pfenning, F. (ed.) CADE. LNCS, vol. 4603, pp. 51–66. Springer (2007) 2, 6, 13
21. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM 12(10), 576–580 (1969) 2, 5
22. Howgrave-Graham, N., Nguyen, P.Q., Pointcheval, D., Proos, J., Silverman, J.H., Singer, A., Whyte, W.: The impact of decryption failures on the security of NTRU encryption. In: Boneh, D. (ed.) CRYPTO. LNCS, vol. 2729, pp. 226–246. Springer (2003) 2

23. Montgomery, P.L.: Speeding the Pollard and elliptic ccurve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987) 2, 16
24. Myreen, M.O., Curello, G.: Proof pearl: A verified bignum implementation in x86-64 machine code. In: Certified Programs and Proofs. LNCS, vol. 8307, pp. 66–81. Springer (2013) 3
25. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS. LNCS, vol. 4424, pp. 568–582. Springer (2007) 3
26. Pottier, L.: Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Knowledge Exchange: Automated Provers and Proof Assistants. p. 418 (2008) 13
27. Pottier, L.: Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. Tech. Rep. abs/1007.3615, CoRR (2010) 13
28. Weiser, M.: Program slicing. In: ICSE. pp. 439–449. IEEE Press (1981) 2, 6
29. Wikipedia: Curve25519, https://en.wikipedia.org/wiki/Curve25519 1

A Appendix

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

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

$$
\begin{array}{rll}
1 : & mulrax \leftarrow & x3; \\
2 : & mulrax \leftarrow & mulrax \times 19; \\
3 : & mulx319 \leftarrow & mulrax; \\
4 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y2, 64); \\
5 : & r0 \leftarrow & mulrax; \\
6 : & mulr01 \leftarrow & mulrdx; \\
7 : & mulrax \leftarrow & x4; \\
8 : & mulrax \leftarrow & mulrax \times 19; \\
9 : & mulx419 \leftarrow & mulrax; \\
10 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y1, 64); \\
11 : & r0 \leftarrow & r0 + mulrax; \\
12 : & [carry, r0] \leftarrow & \mathsf{Split}(r0, 64); \\
13 : & mulr01 \leftarrow & mulr01 + mulrdx + carry; \\
14 : & mulrax \leftarrow & x0; \\
15 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y0, 64); \\
16 : & r0 \leftarrow & r0 + mulrax; \\
17 : & [carry, r0] \leftarrow & \mathsf{Split}(r0, 64); \\
18 : & mulr01 \leftarrow & mulr01 + mulrdx + carry; \\
19 : & mulrax \leftarrow & x0; \\
20 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y1, 64); \\
21 : & r1 \leftarrow & mulrax; \\
22 : & mulr11 \leftarrow & mulrdx; \\
23 : & mulrax \leftarrow & x0; \\
24 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y2, 64); \\
25 : & r2 \leftarrow & mulrax; \\
26 : & mulr21 \leftarrow & mulrdx; \\
27 : & mulrax \leftarrow & x0; \\
28 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y3, 64); \\
29 : & r3 \leftarrow & mulrax; \\
30 : & mulr31 \leftarrow & mulrdx; \\
31 : & mulrax \leftarrow & x0; \\
32 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y4, 64); \\
33 : & r4 \leftarrow & mulrax; \\
34 : & mulr41 \leftarrow & mulrdx; \\
35 : & mulrax \leftarrow & x1; \\
36 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y0, 64); \\
37 : & r1 \leftarrow & r1 + mulrax; \\
38 : & [carry, r1] \leftarrow & \mathsf{Split}(r1, 64); \\
39 : & mulr11 \leftarrow & mulr11 + mulrdx + carry; \\
40 : & mulrax \leftarrow & x1; \\
41 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y1, 64); \\
42 : & r2 \leftarrow & r2 + mulrax; \\
43 : & [carry, r2] \leftarrow & \mathsf{Split}(r2, 64); \\
44 : & mulr21 \leftarrow & mulr21 + mulrdx + carry; \\
45 : & mulrax \leftarrow & x1; \\
46 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y2, 64); \\
47 : & r3 \leftarrow & r3 + mulrax; \\
48 : & [carry, r3] \leftarrow & \mathsf{Split}(r3, 64); \\
49 : & mulr31 \leftarrow & mulr31 + mulrdx + carry; \\
50 : & mulrax \leftarrow & x1; \\
51 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y3, 64); \\
52 : & r4 \leftarrow & r4 + mulrax; \\
53 : & [carry, r4] \leftarrow & \mathsf{Split}(r4, 64); \\
54 : & mulr41 \leftarrow & mulr41 + mulrdx + carry; \\
55 : & mulrax \leftarrow & x1; \\
56 : & mulrax \leftarrow & mulrax \times 19; \\
57 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y4, 64); \\
58 : & r0 \leftarrow & r0 + mulrax; \\
59 : & [carry, r0] \leftarrow & \mathsf{Split}(r0, 64); \\
60 : & mulr01 \leftarrow & mulr01 + mulrdx + carry; \\
61 : & mulrax \leftarrow & x2; \\
62 : & [mulrdx, mulrax] \leftarrow & \mathsf{Split}(mulrax \times y0, 64); \\
63 : & r2 \leftarrow & r2 + mulrax; \\
64 : & [carry, r2] \leftarrow & \mathsf{Split}(r2, 64); \\
65 : & mulr21 \leftarrow & mulr21 + mulrdx + carry;
\end{array}
$$

$66 : mulrax \leftarrow x2;$
$67 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y1, 64);$
$68 : r3 \leftarrow r3 + mulrax;$
$69 : [carry, r3] \leftarrow \mathsf{Split}(r3, 64);$
$70 : mulr31 \leftarrow mulr31 + mulrdx + carry;$
$71 : mulrax \leftarrow x2;$
$72 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y2, 64);$
$73 : r4 \leftarrow r4 + mulrax;$
$74 : [carry, r4] \leftarrow \mathsf{Split}(r4, 64);$
$75 : mulr41 \leftarrow mulr41 + mulrdx + carry;$
$76 : mulrax \leftarrow x2;$
$77 : mulrax \leftarrow mulrax \times 19;$
$78 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y3, 64);$
$79 : r0 \leftarrow r0 + mulrax;$
$80 : [carry, r0] \leftarrow \mathsf{Split}(r0, 64);$
$81 : mulr01 \leftarrow mulr01 + mulrdx + carry;$
$82 : mulrax \leftarrow x2;$
$83 : mulrax \leftarrow mulrax \times 19;$
$84 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y4, 64);$
$85 : r1 \leftarrow r1 + mulrax;$
$86 : [carry, r1] \leftarrow \mathsf{Split}(r1, 64);$
$87 : mulr11 \leftarrow mulr11 + mulrdx + carry;$
$88 : mulrax \leftarrow x3;$
$89 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y0, 64);$
$90 : r3 \leftarrow r3 + mulrax;$
$91 : [carry, r3] \leftarrow \mathsf{Split}(r3, 64);$
$92 : mulr31 \leftarrow mulr31 + mulrdx + carry;$
$93 : mulrax \leftarrow x3;$
$94 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y1, 64);$
$95 : r4 \leftarrow r4 + mulrax;$
$96 : [carry, r4] \leftarrow \mathsf{Split}(r4, 64);$
$97 : mulr41 \leftarrow mulr41 + mulrdx + carry;$
$98 : mulrax \leftarrow mulx319;$
$99 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y3, 64);$
$100 : r1 \leftarrow r1 + mulrax;$
$101 : [carry, r1] \leftarrow \mathsf{Split}(r1, 64);$
$102 : mulr11 \leftarrow mulr11 + mulrdx + carry;$
$103 : mulrax \leftarrow mulx319;$
$104 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y4, 64);$
$105 : r2 \leftarrow r2 + mulrax;$
$106 : [carry, r2] \leftarrow \mathsf{Split}(r2, 64);$
$107 : mulr21 \leftarrow mulr21 + mulrdx + carry;$
$108 : mulrax \leftarrow x4;$
$109 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y0, 64);$
$110 : r4 \leftarrow r4 + mulrax;$
$111 : [carry, r4] \leftarrow \mathsf{Split}(r4, 64);$
$112 : mulr41 \leftarrow mulr41 + mulrdx + carry;$
$113 : mulrax \leftarrow mulx419;$
$114 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y2, 64);$
$115 : r1 \leftarrow r1 + mulrax;$
$116 : [carry, r1] \leftarrow \mathsf{Split}(r1, 64);$
$117 : mulr11 \leftarrow mulr11 + mulrdx + carry;$
$118 : mulrax \leftarrow mulx419;$
$119 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y3, 64);$
$120 : r2 \leftarrow r2 + mulrax;$
$121 : [carry, r2] \leftarrow \mathsf{Split}(r2, 64);$
$122 : mulr21 \leftarrow mulr21 + mulrdx + carry;$
$123 : mulrax \leftarrow mulx419;$
$124 : [mulrdx, mulrax] \leftarrow \mathsf{Split}(mulrax \times y4, 64);$
$125 : r3 \leftarrow r3 + mulrax;$
$126 : [carry, r3] \leftarrow \mathsf{Split}(r3, 64);$
$127 : mulr31 \leftarrow mulr31 + mulrdx + carry;$
$128 : [tmp, r0] \leftarrow \mathsf{Split}(r0, 51);$
$129 : mulr01 \leftarrow mulr01 \times \mathsf{Pow}(2, 13) + tmp;$
$130 : [tmp, r1] \leftarrow \mathsf{Split}(r1, 51);$
$131 : mulr11 \leftarrow mulr11 \times \mathsf{Pow}(2, 13) + tmp;$
$132 : r1 \leftarrow r1 + mulr01;$
$133 : [tmp, r2] \leftarrow \mathsf{Split}(r2, 51);$
$134 : mulr21 \leftarrow mulr21 \times \mathsf{Pow}(2, 13) + tmp;$
$135 : r2 \leftarrow r2 + mulr11;$
$136 : [tmp, r3] \leftarrow \mathsf{Split}(r3, 51);$
$137 : mulr31 \leftarrow mulr31 \times \mathsf{Pow}(2, 13) + tmp;$
$138 : r3 \leftarrow r3 + mulr21;$
$139 : [tmp, r4] \leftarrow \mathsf{Split}(r4, 51);$
$140 : mulr41 \leftarrow mulr41 \times \mathsf{Pow}(2, 13) + tmp;$
$141 : r4 \leftarrow r4 + mulr31;$
$142 : mulr41 \leftarrow mulr41 \times 19;$
$143 : r0 \leftarrow r0 + mulr41;$
$144 : mult \leftarrow r0;$
$145 : [mult, tmp] \leftarrow \mathsf{Split}(mult, 51);$
$146 : mult \leftarrow mult + r1;$
$147 : r1 \leftarrow mult;$
$148 : [mult, tmp2] \leftarrow \mathsf{Split}(mult, 51);$
$149 : r0 \leftarrow tmp;$
$150 : mult \leftarrow mult + r2;$
$151 : r2 \leftarrow mult;$
$152 : [mult, tmp] \leftarrow \mathsf{Split}(mult, 51);$
$153 : r1 \leftarrow tmp2;$
$154 : mult \leftarrow mult + r3;$
$155 : r3 \leftarrow mult;$
$156 : [mult, tmp2] \leftarrow \mathsf{Split}(mult, 51);$
$157 : r2 \leftarrow tmp;$
$158 : mult \leftarrow mult + r4;$
$159 : r4 \leftarrow mult;$
$160 : [mult, tmp] \leftarrow \mathsf{Split}(mult, 51);$
$161 : r3 \leftarrow tmp2;$
$162 : mult \leftarrow mult \times 19;$
$163 : r0 \leftarrow r0 + mult;$
$164 : r4 \leftarrow tmp;$
$165 : z0 \leftarrow r0;$
$166 : z1 \leftarrow r1;$
$167 : z2 \leftarrow r2;$
$168 : z3 \leftarrow r3;$
$169 : z4 \leftarrow r4;$

Let mul denote the above program. Its algebraic specification is

$$(\!|\top|\!)\ \mathsf{mul}\ (\!|\begin{array}{c} radix51(x4, x3, x2, x1, x0) \times radix51(y4, y3, y2, y1, y0) \\ \equiv radix51(z4, z3, z2, z1, z0) \bmod 2^{255} - 19 \end{array}|\!).$$