Automatic Certified Verification of Cryptographic Programs with CoqCryptoLine

Ming-Hsien Tsai
National Applied Research Laboratories

Yu-FU Fu
Georgia Institute of Technology

Jiaxiang Liu and Xiaomu Shi
Shenzhen University

Bow-Yaw Wang and Bo-Yin Yang
Academia Sinica

Abstract

CoqCryptoLine is an automatic certified verification tool for cryptographic programs. It is built on OCaml programs extracted from algorithms fully certified in Coq with SSReflect. Similar to other automatic tools, CoqCryptoLine calls external decision procedures during verification. To ensure correctness, all answers from external decision procedures are validated by certified certificate checkers in CoqCryptoLine. We evaluate CoqCryptoLine on cryptographic programs from Bitcoin, boringSSL, NSS, and OpenSSL. The first certified verification of the reference implementation for number theoretic transform in the post-quantum key exchange mechanism Kyber is also reported.

1 Introduction

Cryptographic programs are crucial to computer security, but they are notoriously difficult to develop. On the one hand, cryptographic programs perform tedious computation over complex algebraic structures. They also need to be extremely efficient for frequent uses on the other. Such are the foremost challenges in developing cryptographic programs. In order to improve qualities of cryptographic programs, novel verification techniques have been developed in various projects [4, 5, 8, 12, 13, 15, 17, 23, 32, 38]. Among them, interactive techniques employ proof assistants and presumably offer better guarantees. They nonetheless require significant human intervention and might not be ideal for daily developments. Automatic techniques on the other hand employ sophisticated decision procedures and thus need little human guidance. However, they might not be very trustful due to possibly unknown errors in complicated decision procedures. An automatic technique with high assurance would be most useful for developing cryptographic programs.

CoqCryptoLine is an automatic verification tool for cryptographic programs with pretty good assurance. Like other automatic techniques, CoqCryptoLine reduces verification tasks to various computational problems solved by external decision procedures. Unlike other techniques, proof assistants are used to certify CoqCryptoLine to attain higher assurance. Instead of cryptographic programs, we use proof assistants to certify the correctness of the CoqCryptoLine verification tool once and for all. Results from external decision procedures are also validated by certificate checkers. To further improve assurance, these certificate checkers are themselves certified by proof assistants. With certified verification algorithms and validated answers from decision procedures, CoqCryptoLine automatically performs verification tasks with better assurance than other automatic tools.

More precisely, we formalize semantics for the typed CryptoLine language and its specification verification problem in [17]. We then specify our verification algorithm and certify its proof of correctness in Coq [11]. Our algorithm transforms the specification verification problem to two computational problems via algebraic and bit-vector reductions. The algebraic reduction is designed for checking non-linear (modular) equations in cryptographic programs through the root entailment problem. The bit-vector reduction is designed for bit-accurate analysis through the SMT problem over the Quantifier-Free Bit-Vector (QF_BV) theory.

For algebraic reduction, we formalize the root entailment problem in Coq and prove the soundness theorem for our reduction. Our soundness theorem for algebraic reduction requires soundness conditions on input cryptographic programs. These conditions in turn demand bit-accurate analysis. They are formally specified in our proof and checked by external SMT QF_BV solvers. For bit-vector reduction, we adopt the formal SMT QF_BV theory in [34] and establish the soundness theorem for our reduction. With the soundness theorems for both reductions, certified techniques for solving the root entailment problem and the SMT problem over the QF_BV theory are employed. CoqCryptoLine is built on OCaml programs extracted from the certified Coq algorithm. Overall, CoqCryptoLine contains $\approx 68k$ lines of OCaml programs extracted from $\approx 24k$ lines of Coq proof scripts.

For evaluation, 52 cryptographic functions in the security libraries Bitcoin [35], boringSSL [19], NSS [25] and

OPENSSL [31] are verified by COQCRYPTOLINE. They are implementations for field and group operations in the elliptic curves secp256k1 (BITCOIN) and Curve25519 (others). These functions have been verified by other automated tools without certificates [17, 24]. Verification results are now certified by COQCRYPTOLINE. We also verify the reference implementation of the number-theoretic transform in the post-quantum key exchange mechanism scheme KYBER from PQCLEAN [33]. To the best of our knowledge, ours is the first verification result on the reference implementation *and* with certificates. We have the following contributions:

1. We propose a methodology for building automatic verification tools with pretty good assurance;
2. We develop the automatic certified verification tool COQCRYPTOLINE for cryptographic programs; and
3. We report the first certified verification on programs in industrial security libraries and the reference implementation of the number-theoretic transform in KYBER.

Related Work. Projects such as HACL* [38], JASMIN [4] and FIAT-CRYPTO [15] apply the correct-by-construction method to construct correct cryptographic programs, whilst EASYCRYPT [5] and CRYPTOVERIF [23] construct machine-checked proofs characterizing probabilistic security properties. Our work on the other hand focuses on verifying programs in existing security libraries. Various cryptography primitives have been formalized and manually verified in proof assistants (for instance, [1–3, 6, 10, 26, 27, 37]). COQCRYPTOLINE in contrast is automatic and thus requires much less human intervention. The first semi-automatic verification on real-world cryptographic assembly programs was proposed in [13]. An SMT solver as well as a proof assistant is used to verify an extensively annotated assembly program. VALE [12, 16] provides a high-level language for specifying assembly programs. Its verification technique is based on SMT solvers and sometimes needs manually added lemmas. CRYPTOLINE [17, 32] is also a tool designed for the specification and verification of cryptographic assembly codes. Its verification algorithm utilizes computer algebra systems in addition to SMT solvers. CRYPTOLINE is also leveraged to verify cryptographic C programs [24]. However, none of the aforementioned automatic techniques is certified. Correctness of these verification tools need to be trusted. The most relevant work is BVCRYPTOLINE [36], which is the first automatic and partly certified verification tool for cryptographic programs. COQCRYPTOLINE nonetheless possesses three essential advantages: (i) BVCRYPTOLINE only supports the unsigned integer representation but COQCRYPTOLINE supports both signed and unsigned representations; (ii) the SMT-based technique in BVCRYPTOLINE is not certified whereas COQCRYPTOLINE certifies both algebraic and SMT-based techniques; (iii) COQCRYPTOLINE is a standalone tool built on extracted OCAML programs but BVCRYPTOLINE is a proof script and hence less efficient. Among automatic certified verification tools, the authors in [22] formalized and certified a Dijkstra-style verification condition generator for a small language in COQ. It was not designed for cryptography verification and no real-world case studies were reported. The verification condition generator of the cryptography verification tool VALE/F* is also certified [16]. VALE/F* however uses the SMT solver Z3 and F* programming language. These external tools must be trusted.

This paper is organized as follows. Section 2 reviews preliminaries. Illustrations of COQCRYPTOLINE are briefed in Section 3. An overview of COQCRYPTOLINE is given in Section 4. It is followed by our formal semantics of the typed CRYPTOLINE language (Section 5). Section 6 highlights our proof of correctness for the verification algorithm. Experimental results are reported in Section 7.

2 Preliminaries

The *coq-nbits* Theory. *coq-nbits* is a formal bit-vector theory in COQ [34]. In the theory, a bit vector is formalized as a Boolean sequence of the type `bits` in the least significant bit first order. It provides the following bit-vector functions: *arithmetic functions* — addition `addB`, subtraction `subB`, half-multiplication `mulB` of the type `bits` → `bits` → `bits`; addition with carry `adcB` and subtraction with borrow `sbbB` of the type `bool` → `bits` → `bits` → `bool` ∗ `bits`; arithmetic right shift function `sarB` of the type `nat` → `bits` → `bits`; *logical functions* — bitwise complement `invB` : `bits` → `bits`; bitwise conjunction `andB` : `bits` → `bits` → `bits`; left shift function `shlB` and logical right shift `shrB` function of the type `nat` → `bits` → `bits`; *arithmetic predicates* — signed and unsigned comparisons including `sltB`, `sleB`, `sgtB`, `sgeB`, `ltB`, `leB`, `gtB` and `geB` of the type `bits` → `bits` → `bool`.

COQQFBV. Given a Boolean formula over Boolean variables, the formula is *satisfiable* if there is an assignment to Boolean variables so that the formula evaluates to true. The *Boolean Satisfiability (*SAT*)* problem is to decide whether a given Boolean formula is satisfiable. *Satisfiability Modulo Theories (*SMT*)* extends Boolean satisfiability with various theories [9]. In the *Quantifier-Free Bit-Vector (*QF_BV*)* theory, QF_BV predicates on QF_BV expressions are admitted. COQQFBV is a certified solver for the SMT QF_BV theory [34]. It formalizes the SMT QF_BV theory using the *coq-nbits* theory. In the formal theory, QF_BV expressions are of the type `QFBV.exp`. QF_BV variables `qfbv_var` and constants `qfbv_const bits` are of the type `QFBV.exp`. COQQFBV moreover provides QF_BV operations such as `qfbv_add` *exp exp*, `qfbv_sub` *exp exp*, `qfbv_mul` *exp exp*; bitwise logical operations `qfbv_not` *exp*, `qfbv_and` *exp exp*; logical shift operations `qfbv_shl` *exp n*, `qfbv_lshr` *exp n*;

the arithmetic right shift operation `qfbv_ashr` *exp* *n*.

QF_BV predicates are of the type `QFBV.bexp` in CoqQFBV. They include: equality `qfbv_eq` *exp exp*, signed and unsigned less than predicates `qfbv_slt` *exp exp* and `qfbv_ult` *exp exp* respectively; logical negation `qfbv_lneg` *bexp*, conjunction `qfbv_conj` *bexp bexp*, implication `qfbv_imp` *bexp bexp*. Finally, the CoqQFBV expression `qfbv_ite` *bexp* exp_0 exp_1 evaluates to exp_0 if the QF_BV predicate *bexp* is true and exp_1 otherwise. A CoqQFBV *query* is a sequence of QF_BV predicates. An assignment to QF_BV variables *satisfies* a predicate if it evaluates the predicate to true; an assignment *satisfies* a CoqQFBV query if it satisfies every predicate in the query. A CoqQFBV query is *satisfiable* if there is an assignment to QF_BV variables satisfying the query. The SMT QF_BV *problem* is to decide whether a given CoqQFBV query is satisfiable.

Polynomial Modular Equations. Let $\mathbb{N}$ be the set of non-negative integers, $\mathbb{Z}$ the set of integers, $\bar{\mathbf{x}}$ a set of variables and $\mathbb{Z}[\bar{\mathbf{x}}]$ the set of multivariate polynomials in $\bar{\mathbf{x}}$ with integral coefficients. Let $f_0, f_1, f_2 \in \mathbb{Z}[\bar{\mathbf{x}}]$. $f_0(\bar{\mathbf{x}}) = f_1(\bar{\mathbf{x}})$ is a polynomial *equation*; $f_0(\bar{\mathbf{x}}) \equiv f_1(\bar{\mathbf{x}}) \bmod f_2(\bar{\mathbf{x}})$ is a polynomial *modular equation*. A *(modular) equation* is a polynomial equation or a polynomial modular equation. A *root* of a polynomial equation $f_0(\bar{\mathbf{x}}) = f_1(\bar{\mathbf{x}})$ is a sequence $\bar{z}$ of integers such that $f_0(\bar{z}) - f_1(\bar{z}) = 0$. A *root* of a polynomial modular equation $f_0(\bar{\mathbf{x}}) \equiv f_1(\bar{\mathbf{x}}) \bmod f_2(\bar{\mathbf{x}})$ is a sequence $\bar{z}$ of integers such that $f_2(\bar{z})$ divides $f_0(\bar{z}) - f_1(\bar{z})$. A *system* of (modular) equations is a set of (modular) equations. A *root* of a system of (modular) equations is a sequence $\bar{z}$ of integers such that $\bar{z}$ is a root of every (modular) equation in the system. Given two systems Π and Π' of (modular) equations, Π *entails* Π' if all roots of Π are also roots of Π'. The *root entailment* problem is to decide whether Π entails Π'.

3 CoqCryptoLine

CoqCryptoLine is an automatic certified verification tool for cryptographic programs. To illustrate how CoqCryptoLine is used, the x86_64 assembly subroutines `ecp_nistz256_add` and `ecp_nistz256_mul_montx` from OpenSSL are verified as examples.

Figure 1 shows the input for CoqCryptoLine. It contains a CryptoLine specification for the assembly subroutine `ecp_nistz256_add`. The original subroutine is marked between the comments `ecp_nistz256_add STARTS` and `ecp_nistz256_add ENDS`, which is obtained automatically from the Python script provided by CryptoLine [32]. The left column contains the parameter declaration, pre-condition, and variable initialization. More precisely, three 256-bit unsigned integers are declared as inputs. Each 256-bit input integer is denoted by four 64-bit unsigned integer variables in the least significant bit first representation. The expression `limbs` *n* `[`d_0`,` d_1`, ...,` d_m`]` is short for d_0 `+` d_1`*2**`*n* `+ ... +`d_m`*2**(`*m*`*`*n*`)`. The 256-bit integer represented by `m`'s is the prime $p256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ from the NIST curve. The 256-bit integers represented by `a`'s and `b`'s are less then the prime. The inputs and constants are then put in the variables for memory cells with the MOV instructions.

The right column contains the post-condition of the subroutine `ecp_nistz256_add`. After the subroutine ends, the 256-bit result is moved to the variables `c`'s. The post-condition specifies two properties about the subroutine. Firstly, the 256-bit integer represented by `c`'s is the sum of the input integers represented by `a`'s and `b`'s modulo the prime $p256$ represented by `m`'s. Secondly, the output integer is less than the prime. Observe that the extended 320-bit sum of the 256-bit integers represented by `a`'s and `b`'s is computed in the modular equation. Since input integers in the specified range may induce overflow when computing 256-bit sums, the modular equation would not hold for 256-bit sums.

Using eight threads, CoqCryptoLine verifies all inputs satisfying the pre-condition must result in outputs satisfying the post-condition in 136 seconds with the transcript below:

```
$ run_coqcryptoline ecp_nistz256_add.cl
Parsing Cryptoline file:            [OK]            0.000588 seconds
Checking CNF formulas (3):
        CNF #0:                     [UNSAT]         0.429629 seconds
                                    [CERTIFIED]     0.602632 seconds
        CNF #1:                     [UNSAT]         0.722745 seconds
                                    [CERTIFIED]     0.850775 seconds
        CNF #2:                     [UNSAT]        32.176368 seconds
                                    [CERTIFIED]    82.114442 seconds
Results of checking CNF formulas:   [OK]          114.669136 seconds
Finding polynomial coefficients
Finished finding polynomial coefficients            0.000012 seconds
Verification result:                [OK]          135.276716 seconds
```

The annotation is almost minimal in Figure 1. In order to verify cryptographic programs, input assumptions and output requirements need be specified by verifiers manually. Variables for memory cells are initialized straightforwardly. No further human intervention is needed in this case.

The assembly subroutine `ecp_nistz256_mul_montx` is similar. It takes two 256-bit unsigned integers represented by 64-bit variables `a`'s and `b`'s. Recall the variables `m`'s denote the 256-bit prime $p256$ for the curve. We have a similar pre-condition as for `ecp_nistz256_add`.

```
and [ m0=0xffffffffffffffff, m1=0x00000000ffffffff,
      m2=0x0000000000000000, m3=0xffffffff00000001 ]
&&
and [ m0=0xffffffffffffffff@64, m1=0x00000000ffffffff@64,
      m2=0x0000000000000000@64, m3=0xffffffff00000001@64,
      limbs 64 [a0,a1,a2,a3] <u limbs 64 [m0,m1,m2,m3],
      limbs 64 [b0,b1,b2,b3] <u limbs 64 [m0,m1,m2,m3] ]
```

The first part of the pre-condition is for the algebraic reduction; the second part is for the bit-vector reduction.

The output 256-bit integer represented in the variables `c`'s has two requirements. Firstly, the output integer multiplied by 2^{256} is equal to the product of the input integers modulo the prime. Secondly, the output integer is less than the prime $p256$. Formally, we have the following post-condition:

```
proc main
(uint64 a0, uint64 a1, uint64 a2, uint64 a3,
 uint64 b0, uint64 b1, uint64 b2, uint64 b3,
 uint64 m0, uint64 m1, uint64 m2, uint64 m3) =
{ true
  &&
  and [ m0=0xffffffffffffffff@64,
        m1=0x00000000ffffffff@64,
        m2=0x0000000000000000@64,
        m3=0xffffffff00000001@64,
        limbs 64 [a0, a1, a2, a3] <u
            limbs 64 [m0, m1, m2, m3],
        limbs 64 [b0, b1, b2, b3] <u
            limbs 64 [m0, m1, m2, m3] ] }

mov L0x7fffffffd9c0 a0; mov L0x7fffffffd9c8 a1;
mov L0x7fffffffd9d0 a2; mov L0x7fffffffd9d8 a3;
mov L0x7fffffffd9e0 b0; mov L0x7fffffffd9e8 b1;
mov L0x7fffffffd9f0 b2; mov L0x7fffffffd9f8 b3;

mov L0x55555557c000 0xffffffffffffffff@uint64;
mov L0x55555557c008 0x00000000ffffffff@uint64;
mov L0x55555557c010 0x0000000000000000@uint64;
mov L0x55555557c018 0xffffffff00000001@uint64;

(* ecp_nistz256_add STARTS *)
mov r8 L0x7fffffffd9c0;
mov r13 0@uint64;
mov r9 L0x7fffffffd9c8;
mov r10 L0x7fffffffd9d0;
mov r11 L0x7fffffffd9d8;
adds carry r8 r8 L0x7fffffffd9e0;
adcs carry r9 r9 L0x7fffffffd9e8 carry;
mov rax r8;
adcs carry r10 r10 L0x7fffffffd9f0 carry;
adcs carry r11 r11 L0x7fffffffd9f8 carry;
mov rdx r9;
adc r13 r13 0@uint64 carry;
subb carry r8 r8 L0x55555557c000;
mov rcx r10;
sbbs carry r9 r9 L0x55555557c008 carry;
sbbs carry r10 r10 L0x55555557c010 carry;
mov r12 r11;
sbbs carry r11 r11 L0x55555557c018 carry;
sbbs carry r13 r13 0@uint64 carry;
cmov r8 carry rax r8;
cmov r9 carry rdx r9;
mov L0x7fffffffda00 r8;
cmov r10 carry rcx r10;
mov L0x7fffffffda08 r9;
cmov r11 carry r12 r11;
mov L0x7fffffffda10 r10;
mov L0x7fffffffda18 r11;
(* ecp_nistz256_add ENDS *)

mov c0 L0x7fffffffda00; mov c1 L0x7fffffffda08;
mov c2 L0x7fffffffda10; mov c3 L0x7fffffffda18;

{ true
  &&
  and [ eqmod limbs 64 [c0, c1, c2, c3, 0@64]
              limbs 64 [a0, a1, a2, a3, 0@64] +
                  limbs 64 [b0, b1, b2, b3, 0@64]
              limbs 64 [m0, m1, m2, m3, 0@64],
        limbs 64 [c0, c1, c2, c3] <u
            limbs 64 [m0, m1, m2, m3] ] }
```

Figure 1: CRYPTOLINE Specification for `ecp_nistz256_add`

```
eqmod limbs 64 [0, 0, 0, 0, c0, c1, c2, c3]
      limbs 64 [a0, a1, a2, a3] * limbs 64 [b0, b1, b2, b3]
      limbs 64 [m0, m1, m2, m3]
&&
limbs 64 [c0, c1, c2, c3] <u limbs 64 [m0, m1, m2, m3]
```

Here, we employ the algebraic reduction to verify the non-linear modular equality. The bit-vector reduction is used to verify that the output integer is in the proper range.

For `ecp_nistz256_mul_montx`, more annotations are needed however. These annotations are additional hints for COQCRYPTOLINE to verify the post-condition. For instance, consider adding two 256-bit integers represented by 64-bit variables. A chain of four 64-bit additions is performed and carries are propagated. At the end of the addition chain, the last carry is almost certainly zero or the 256-bit sum is incorrect. In `ecp_nistz256_mul_montx`, two addition chains are running interleavingly. One uses the carry flag for carries; the other uses the overflow flag. To tell COQCRYPTOLINE about the last carries, the following annotation is added at the end of two interleaving addition chains:

```
assert true && and [ carry=0@1, overflow=0@1 ];
assume and [ carry=0, overflow=0 ] && true;
```

The ASSERT instruction verifies both carry and overflow flags are zeroes through the bit-vector reduction. The ASSUME instruction then passes the information to the algebraic reduction. Effectively, COQCRYPTOLINE checks both flags must be zeroes for all inputs satisfying the pre-condition. The facts are then used as lemmas to verify the post-condition with the algebraic reduction.

The full specification for `ecp_nistz256_mul_montx` is listed in Appendix A. Out of 230 lines, 50 lines of annotations are added manually. Among the 50 lines of annotations, about 20 of them are for variable declaration, pre-condition, variable initialization, and post-condition. As in `ecp_nistz256_add`, they are added rather straightforwardly. The remaining 30 lines of annotations give more hints to COQCRYPTOLINE. With all 50 lines of annotations, COQCRYPTOLINE verifies the post-condition in 189 seconds with eight threads.

These examples illustrate the typical verification flow. In order to verify a cryptographic program, verifiers first construct a CRYPTOLINE specification. The pre-condition for program inputs, the post-condition for outputs, and variable initialization need be specified manually. Additional annotations may be added as hints. Notice that all hints only tell COQCRYPTOLINE *what* properties should hold. They do not explain *why* properties should hold. Proofs of annotated hints and the post-condition are found by COQCRYPTOLINE automatically. Consequently, manual annotations are minimized and verification efforts are reduced significantly.

4 Technology Overview

Figure 2 outlines the components in COQCRYPTOLINE. In the figure, dashed components represent external tools. Rectangular boxes are certified components and rounded boxes are uncertified. We use the proof assistant COQ with SSREFLECT to certify components in COQCRYPTOLINE [11, 18]. Note that all our proof efforts are transparent to verifiers. No COQ proof is needed from verifiers during verification of cryptographic programs with COQCRYPTOLINE (Section 3).

A CRYPTOLINE specification contains a CRYPTOLINE program with pre- and post-conditions. A CRYPTOLINE specification is valid if every program execution starting from a program state (called *store*) satisfying the pre-condition ends in a store satisfying the post-condition. From a CRYPTOLINE specification text, the COQCRYPTOLINE parser translates the text into an abstract syntax tree defined in the COQ module

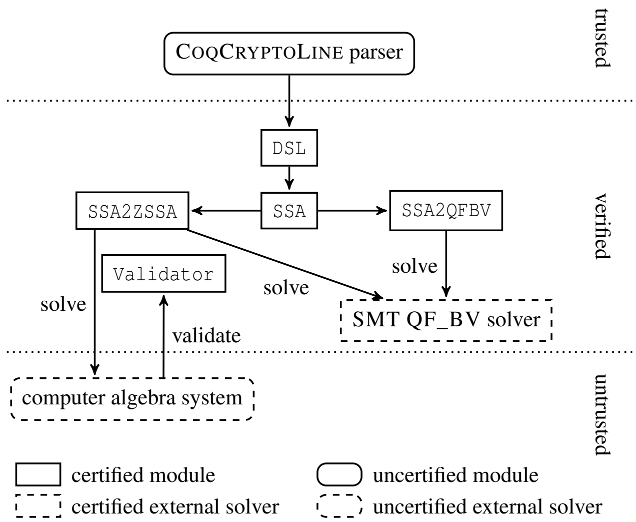

Figure 2: Overview of CoqCryptoLine

DSL. The module gives formal semantics for the typed CryptoLine language in [17]. Validity of CryptoLine specifications is also formalized (Section 5). Similar to most program verification tools, CoqCryptoLine transforms CryptoLine specifications to the static single assignment (SSA) form. The SSA module gives our transformation algorithm. It moreover shows that validity of CryptoLine specifications is preserved by the SSA transformation.

CoqCryptoLine then reduces specifications in SSA form via two reductions. The SSA2ZSSA module contains our algebraic reduction to the root entailment problem (Section 6.1). Concretely, a system of (modular) equations is constructed from the given program so that program executions correspond to roots of the system of (modular) equations. To verify post-conditions, it suffices to check if roots for executions are also roots of (modular) equations in the post-condition. However, program executions can deviate from roots of (modular) equations when over- or under-flow occurs. CoqCryptoLine generate soundness conditions to ensure executions correspond to roots of (modular) equations. The verification problem is thus reduced to the root entailment problem provided that soundness conditions hold.

The SSA2QFBV module gives our bit-vector reduction to the SMT QF_BV problem. It constructs a CoqQFBV query to check validity of the given CryptoLine specification (Section 6.2). Concretely, a CoqQFBV query is built such that all program executions correspond to satisfying assignments to the query and vice versa. To verify post-conditions, it suffices to check if satisfying assignments for the query also satisfy the post-conditions. The verification problem is thus reduced to the SMT QF_BV problem. Additional CoqQFBV queries are constructed to check soundness conditions for the algebraic reduction. We formally prove the equivalence between soundness conditions and corresponding queries.

With the two certified reduction algorithms, it remains to solve the root entailment problem and the SMT QF_BV problem with external solvers. CoqCryptoLine improves the techniques in [21, 36] to validate answers from an external computer algebra system (CAS). To solve instances of the SMT QF_BV problem, CoqCryptoLine employs the certified SMT QF_BV solver CoqQFBV [34]. In all cases, instances of the root entailment problem and the SMT QF_BV problem are solved with certificates. To further improve assurance, CoqCryptoLine employs certified certificate checkers to validate answers to the root entailment and the SMT QF_BV problem problem.

The CoqCryptoLine tool is built on OCaml programs extracted from certified algorithms in Coq with SSReflect [11, 18]. We moreover integrate the OCaml programs from the certified SMT QF_BV solver CoqQFBV [34]. Our trusted computing base consists of (1) CoqCryptoLine parser, (2) text interface with external SAT solvers (from CoqQFBV), (3) the proof assistant Isabelle [29] (from CoqQFBV) and (4) the Coq proof assistant. Particularly, sophisticated decision procedures in external CASs and SAT solvers used in CoqQFBV are not trusted.

5 Typed CryptoLine

CryptoLine is a domain specific language for modeling cryptographic assembly programs [32, 36]. Modern cryptography relies on complicated computation over large finite fields or groups. To improve efficiency, such computation is often implemented by assembly programs in industrial security libraries like OpenSSL [31]. Moreover, signed and unsigned numbers with different over- and under-flow bounds often coexist. A type system is added to CryptoLine to distinguish variables in different representations. We formalize the typed CryptoLine and its semantics in [17] into the DSL module using the proof assistant Coq with SSReflect and *coq-nbits*. Certain Coq notations and definitions may be unfolded in our presentation.

5.1 Types, Variables, and Stores

Types in the CryptoLine type system, or *CL types* for short, are inductively defined as typ in Coq.

```
Inductive typ : Set :=
  Tuint : nat → typ | Tsint : nat → typ.
```

Let w be a natural number of Coq's type nat. Tuint w and Tsint w are the CL types of bit-vectors of width w in the unsigned and two's complement signed representations respectively.

Variables in CryptoLine are typed under a *type environment*, which is a finite mapping from variables to CL types. The type of variables is var and the type of type environments is `env`. Type environments evolve during program executions as a variable may be assigned values of different CL types in different program locations. Variables in CRYPTOLINE are evaluated in a *store*, which is a finite mapping from variables to bit-vectors. The type of stores is `S.t`. Let `v` be a variable, `bs` be a bit-vector, and `s` and `t` are stores. `S.acc v s` denotes the value of `v` in `s`. The COQ's proposition `S.Upd v bs s t` denotes that `t` is updated from `s` by mapping `v` to `bs`. Note that there may be inconsistency between the width of the CL type of a variable in a type environment and the width of the bit-vector value of the variable in a store. This inconsistency is prevented by conformity to be introduced later in Section 5.4.

5.2 Expressions and Predicates

CRYPTOLINE has two forms of expressions, of which one is used to describe multivariate polynomials over integers and the other is used to describe operations over bit-vectors. Both forms of expressions are evaluated in a store.

Algebraic expressions in CRYPTOLINE are used to describe multivariate polynomials over integers such as the polynomial `a0 + a1*2**64 + a2*2**128 + a3*2**192` (or equivalently `limbs 64 [a0, a1, a2, a3]`) mentioned in Section 3. The type of algebraic expressions is `eexp`. An algebraic expression is inductively defined to be a variable `Evar v`, an integer constant `Econst n`, a unary algebraic expression `Eunop euop e`, or a binary algebraic expression `Ebinop ebop e1 e2` where `v` is a variable, `n` is an integer of COQ's type `Z`, `euop` is a unary algebraic operator, `ebop` is a binary algebraic operator, and `e`, `e1`, and `e2` are algebraic expressions. The unary algebraic operator `Eneg` (negation), and the binary algebraic operators `Eadd` (addition), `Esub` (subtraction), and `Emul` (multiplication) are supported.

An algebraic expression `e` is evaluated in a store `s` under a type environment `te` to an integer `eval_eexp e te s`. Since a store maps a variable to a bit-vector, the bit-vector has to be converted to an appropriate integer in the evaluation. This is done by the function `bv2z` which converts a bit-vector to an integer by the *coq-nbits* functions `to_Zpos` (using unsigned representation) and `to_Z` (using two's complement representation) depending on the CL type of the variable in the type environment.

```
Definition bv2z (t : typ) (bs : bits) : Z :=
  match t with
  | Tuint _ => to_Zpos bs
  | Tsint _ => to_Z bs
  end.
```

Algebraic operators `Eneg`, `Eadd`, `Esub`, and `Emul` are evaluated using the COQ notations `-`, `+`, `-`, and `*` respectively for unary negation, addition, subtraction, and multiplication over integers.

Range expressions in CRYPTOLINE are used to describe operations over bit-vectors and are designed as a subset of QF_BV expressions in COQQFBV. More specifically, a range expression is a variable `Rvar v`, a bit-vector constant `Rconst w bs`, a unary range expression `Runop w ruop e`, a binary range expression `Rbinop w rbop e1 e2`, a zero extension `Ruext w e i`, or a signed extension `Rsext w e i` where `v` is a variable, `bs` is a bit-vector, `ruop` is a unary range operator, `rbop` is a binary range operator, `i` is a natural number for the number of bits to be extended, `w` is a natural number for the bit width of the arguments, and `e`, `e1`, and `e2` are range expressions. Two unary range operators `Rnegb` (negation) and `Rnotb` (bitwise inversion) are supported. The supported binary range operators include `Radd` (addition), `Rsub` (subtraction), `Rmul` (multiplication), `Rumod` (unsigned remainder), `Rsrem` (signed remainder with sign follows dividend), `Rsmod` (signed remainder with sign follows divisor), bitwise AND (`Randb`), bitwise OR (`Rorb`), and bitwise XOR (`Rxorb`). The type of range expressions is `rexp`.

A range expression `e` is evaluated in a store `s` to a bit-vector `eval_rexp e s`. The definition of `eval_rexp` follows the semantics of QF_BV expressions defined in COQQFBV. For example, `Radd` is evaluated in the same way as `qfbv_add` in COQQFBV. Note that type environments are not needed in the evaluation of range expressions.

Same as expressions, CRYPTOLINE has two forms of predicates, of which one is used to describe integer properties and the other is used to describe bit-accurate properties.

Algebraic predicates in CRYPTOLINE are used to describe integer properties. The type of algebraic predicates is `ebexp`. An algebraic predicate is inductively defined to be an atomic algebraic predicate or a conjunction (`Eand`) of algebraic predicates. An *atomic algebraic predicate* is `Etrue` or a (modular) equation over algebraic expressions of type `eexp`. Given algebraic expressions `e1`, `e2`, and `m`, `Eeq e1 e2` is the equality of `e1` and `e2` while `Eeqmod e1 e2 m` is the congruence of `e1` and `e2` modulo `m`. For example, the congruence $X \equiv 1 \bmod 2$ can be defined as the algebraic predicate `Eeqmod (Evar X) (Econst 1) (Econst 2)` assuming that X is a variable.

Given a store `s` and a type environment `te`, the semantics of an algebraic predicate `e` in `s` under `te` is defined as the proposition `eval_ebexp e te s` which holds if and only if all atomic algebraic predicates in `e` hold in `s` under `te`. The atomic algebraic predicate `Etrue` always holds. `Eeq e1 e2` holds if `eval_eexp e1 te s = eval_eexp e2 te s` where `=` is the equality in COQ. `Eeqmod e1 e2 m` holds if `modulo (eval_eexp e1 te s) (eval_eexp e2 te s) (eval_eexp m te s)`. For integers `x`, `y`, and `p`, `modulo x y p` holds if and only if there exists some integer `k` such that `x - y = k * p`.

Range predicates in CRYPTOLINE on the other hand are used to describe bit-accurate properties and are designed as a subset of QF_BV predicates in COQQFBV [1]. More specifically, a range predicate is an atomic range predicate or an arbitrary Boolean expression (`Rneg` for Boolean NOT, `Rand`

[1] It is possible to define range predicates as QF_BV predicates although the subset is sufficient for us to verify many cryptographic programs.

for Boolean AND, and Ror for Boolean OR) over range predicates. An *atomic range predicate* is Rtrue, an equality Req w e1 e2, or a comparison Rcmp w rcop e1 e2 where rcop is a comparison operator, e1 and e2 are range expressions, and w is the width of the arguments. A comparison operator can be Rult (unsigned less-than) or Rslt (signed less-than). For example, testing whether an unsigned variable X is less than an unsigned variable Y, written as X <u Y in the CRYPTOLINE text, is represented as the range predicate Rcmp w Rult (Rvar X) (Rvar Y), assuming that both X and Y have width w. The type of range predicates is rbexp.

A range predicate e is evaluated in a store s to a Boolean eval_rbexp e s. The definition of eval_rbexp follows the semantics of QF_BV predicates defined in COQQFBV. For example, Rult is evaluated in the same way as qfbv_ult in COQQFBV.

A *predicate* in CRYPTOLINE is composed of an algebraic predicate and a range predicate. The type of predicates is bexp. The algebraic predicate and the range predicate of a predicate e are obtained by eqn_bexp e and rng_bexp e respectively. The evaluation of a predicate e in a store s under a type environment te is defined as the proposition eval_bexp e te s, which is the conjunction of eval_ebexp (eqn_bexp e) te s and eval_rbexp (rng_bexp e) s.

5.3 Instructions and Programs

An atom is either Avar v or Aconst ty bs where v is a variable, bs is a bit-vector, ty is the intended type of the bit-vector. The function eval_atom evaluates an atom in a store. Given a store s, a variable v, a bit-vector bs, and a type ty, eval_atom (Avar v) s and eval_atom (Aconst ty bs) s are defined as S.acc v s and bs respectively.

An instruction of type instr assigns *destination variables* with values of *source atoms*.

```
Inductive instr : Type :=
  Imov : var → atom → instr
| Iadd : ... | Iadds : ... | Iadc : ... | Iadcs : ...
| Isub : ... | Isubb : ... | Isbb : ... | Isbbs : ...
| Imul : ... | Imull : ... | Imulj : ...
| Ishl : ... | Icshl : ... | Ijoin : ... | Isplit : ...
| Inot : ... | Iand : ... | Ior : ... | Ixor : ...
| Icmov : ... | Inondet : ... | Icast : ...
...
| Iassume : bexp → instr.
```

Imov v a assigns the value of the source atom a to the destination variable v. Arithmetic instructions such as addition (Iadd and Iadds), addition with carry (Iadc and Iadcs), subtraction (Isub and Isubb), subtraction with borrow (Isbb and Isbbs), half-multiplication (Imul), and full multiplication (Imull and Imulj) are supported. Additional flags (such as carry and borrow flags) are set in Iadds, Iadcs, Isubb, and Isbbs. Bitwise operations (Inot, Iand, Ior, and Ixor), conditional moves (Icmov), shifting operations (Ishl and Icshl), and splitting operations (Isplit) are also allowed. The Ijoin v a1 a2 instruction concatenates values of source atoms a1 and a2 and puts the concatenation in the destination variable v. The Icast v t a instruction casts the value of the source atom a into the designated type t. The non-deterministic instruction Inondet v t assigns the destination variable v an arbitrary value in the designated type t. For verification purposes, COQCRYPTOLINE allows programmers assumptions about executions. The Iassume e instruction ensures that the designated predicate e holds in all executions. A program is a sequence of instructions.

Executions of CRYPTOLINE programs are formalized by relational semantics. Informally, our semantics of CRYPTOLINE programs specifies how stores are changed by instructions in a program. Consider a type environment te, stores s, t and an instruction i. The inductive proposition eval_instr te i s t denotes that the *successor* store t can be reached by executing i at s. For example, eval_instr te (Imov v a) s t holds if S.Upd v (eval_atom a s) s t holds, that is, t is updated from s by mapping v to the value of a in s. For the addition instruction Iadd, eval_instr te (Iadd v a1 a2) s t holds if S.Upd v (addB (eval_atom a1 s) (eval_atom a2 s)) s t holds, that is, t is updated from s by mapping v to the bit-vector sum of a1 and a2 in s.

The executions of Icast instructions are more complicated. Let v be a variable, ty be a CL type, a be an atom, s be a store, and bs be the evaluation of a in s. The execution of Icast v ty a at s assigns bs represented in ty to v. Let size bs be the width of bs and w be the width of ty. Depending on the relation between size bs and w, the casted value may be bs, its truncation, or its extension. If the CL type of a is unsigned, the casted value assigned to v is ucastB bs w:

```
Definition ucastB (bs : bits) (w : nat) : bits :=
  if w == size bs then bs
  else if w < size bs then low w bs
       else zext (w - size bs) bs.
```

where low w bs is the lower w bits of bs. Otherwise the casted value is scastB bs w where scastB is defined same as ucastB except that sext is used for extension instead of zext. See Appendix B for more details of the semantics of the instructions in typed CRYPTOLINE.

Given a type environment te, a program p, and stores s and t, the proposition eval_program te p s t denotes that t can be reached by executing p from the s under te. If p is an empty sequence, denoted by [::], the store is unchanged, i.e., eval_program te [::] s s holds. If p is an instruction i followed by a program p′, denoted by i::p′, eval_program te (i::p′) s t holds if there is some store u such that both eval_instr te i s u and eval_program te′ p′ u t hold where te′ is instr_succ_typenv i te. The type environment updated from te after executing the instruction i is formalized as the term instr_succ_typenv i te. Similarly, type environment updated from te after executing the program p is formalized as the term program_succ_typenv p te.

Compared to BVCRYPTOLINE, COQCRYPTOLINE offers several new instructions such as `Inondet`, `Icmov`, `Inot`, `Iand`, `Ior`, `Ixor`, `Imulj`, `Icast`, `Ijoin`, and `Iassume`. Bitwise operations `Iand` and `Ior` are often used for bit masking, which is commonly used in security libraries. The `Iassume` instruction allows interchangeability between algebraic properties and range properties. That is, for an algebraic property hard to be proved by the CAS, we may prove its corresponding range property by the SMT QF_BV solver and then assume that the algebraic property holds, and vice versa. For example, as mentioned in Section 3, we prove `carry = 0@1` in the range side using an SMT QF_BV solver and then assume `carry = 0` in the algebraic side so that the external CAS knows `carry = 0` when solving algebraic predicates. Such interchangeability is not available in BVCRYPTOLINE.

5.4 Specifications

A specification `s` is formalized as a COQ record `spec` with four fields, of which `sinputs s` is the initial type environment, `spre s` is the pre-condition, `sprog s` is the program, and `spost s` is the post-condition. Both the pre-condition and the post-condition are predicates.

```
Record spec : Type :=
  { sinputs : env; spre : bexp;
    sprog : program; spost : bexp }.
```

To focus on the algebraic part and the range part of a specification, we introduce another two forms of specifications.

```
Record espec :=
  { esinputs : env; espre : bexp;
    esprog : program; espost : ebexp }.
Record rspec :=
  { rsinputs : env; rspre : rbexp;
    rsprog : program; rspost : rbexp }.
```

The functions `espec_of_spec` and `rspec_of_spec` convert a specification to an algebraic specification of type `espec` and a range specification of type `rspec` respectively simply by dropping either algebraic predicates or range predicates in the pre- and post-conditions.

A store `s` is *conformed* to a type environment `te`, defined as `S.conform s te` in COQ, if and only if for every variable `v` in `te`, the type of `v` in `te` and the bit-vector value of `v` in `s` have the same width. The validity of a specification `s`, defined as the proposition `valid_spec s`, holds if and only if the execution of `sprog s` from any store `s1` conformed to `sinputs s` and satisfying `spre s` terminates in a store `s2` where `spost s` holds.

```
Definition valid_spec (s : spec) : Prop :=
  ∀ s1 s2 : S.t,
    S.conform s1 (sinputs s) →
    eval_bexp (spre s) (sinputs s) s1 →
    eval_program (sinputs s) (sprog s) s1 s2 →
    eval_bexp (spost s)
      (program_succ_typenv (sprog s) (sinputs s)) s2.
```

The validity of an algebraic specification and the validity of a range specification are defined similarly as `valid_espec` and `valid_rspec` respectively. We have the lemma `valid_spec_split` for splitting the validity of a specification into the validity of its algebraic part and the validity of its range part.

```
Lemma valid_spec_split (s : spec) :
  valid_espec (espec_of_spec s) →
  valid_rspec (rspec_of_spec s) → valid_spec s.
```

6 Certified Verification

Given a typed CRYPTOLINE specification text, the COQCRYPTOLINE parser translates the text into a term of type `spec`, or more specifically `DSL.spec` (`spec` in the `DSL` module). A specification of type `DSL.spec` is verified by the function `verify_dsl`:

```
Definition verify_dsl (o : options) (s : DSL.spec) :=
  verify_ssa o (SSA.ssa_spec s).
```

where `SSA.ssa_spec` is the SSA transformation. The SSA form of the specification is then verified by the function `verify_ssa`. The type of specifications in SSA is `SSA.spec` (`spec` in the `SSA` module). The two modules `DSL` and `SSA` are basically the same except that they have different types of variables. Thus all the syntax and semantics defined in `DSL` (Section 5) are also available in `SSA`. We may omit `DSL.` and `SSA.` when it is clear in the context.

A specification in SSA is verified by the function `verify_ssa` where the algebraic reduction (to the root entailment problem) and the bit-vector reduction (to the SMT QF_BV problem) are applied.

```
Definition verify_ssa (o : options) (s : SSA.spec) :=
  (verify_rspec_algsnd s) && (verify_espec o s)
```

The algebraic reduction and the solving of root entailment problems are performed in the function `verify_espec`. The bit-vector reduction and the solving of SMT QF_BV queries together with the soundness conditions are performed in the function `verify_rspec_algsnd`. We detail `verify_espec` and `verify_rspec_algsnd` in the following subsections. While we present our verification algorithms defined in COQ, the algorithms are extracted to OCAML code by COQ for execution.

6.1 Algebraic Specification Verification

The function `verify_espec` applies the algebraic reduction to a specification in SSA through `algred_espec` and then solves the resulting root entailment problems through `verify_rep` or its parallel version `verify_rep_list` both with answers verified by a validator.

```
Definition verify_espec (o : options) (s : SSA.spec) :=
  (let rp : rep := (algred_espec o
```

```
                 (new_svar_spec s) (espec_of_spec s)) in
  if sequential_solving o
  then (verify_rep o rp)
  else (verify_rep_list o rp)).
```

A root entailment problem is formalized as a COQ's record `rep` where a system of (modular) equations is represented as an algebraic predicate.

```
Record rep : Type :=
  { premise : SSA.ebexp; conseq : SSA.ebexp }.
```

Given a root entailment problem `rp`, we want to decide whether `valid_rep rp` holds, that is, whether the premise `premise rp` entails the consequence `conseq rp`, formalized as `entails (premise rp) (conseq rp)`.

```
Definition entails (f g : ebexp) : Prop :=
  ∀ s, eval_zbexp f s → eval_zbexp g s.
Definition valid_rep (rp : rep) : Prop :=
  entails (premise rp) (conseq rp).
```

In a root entailment problem, algebraic expressions and algebraic predicates are evaluated through `eval_zexp` and `eval_zbexp` over integer stores, which are mappings from variables to integer values. We formalize integer stores as the type `ZS.t` in COQ. The evaluation functions `eval_zexp` and `eval_zbexp` are the same as `eval_eexp` and `eval_ebexp` respectively except that the conversion function `bv2z` is not needed. `bv2z` is not used in the algebraic reduction because the value of a variable in an integer store is already an integer.

The algebraic reduction `algred_espec` translates an algebraic specification `s` to a root entailment problem.

```
Definition algred_espec o avn (s : SSA.espec) : rep :=
  let (_, eprogs) :=
    algred_program (esinputs s) avn init_g (esprog s) in
  {| premise := eand (eqn_bexp (espre s)) (eands eprogs);
     conseq := espost s |}.
```

During the reduction, a system of (modular) equations is represented as a sequence of algebraic predicates temporarily. Intuitively, a system of (modular) equations `eprogs` is constructed from the program `esprog s` so that program executions correspond to roots of the system of (modular) equations. We then check if `eprogs` conjuncted with the precondition `eqn_bexp (espre s)` entails the post-condition `espost s` of `s`. Here `eand` (and `eands`) is used to construct a conjunction (`Eand`) from two algebraic predicates (and a list of algebraic predicates respectively).

The function `algred_program` reduces a program instruction by instruction through `algred_instr` where an atom is translated to an algebraic expression by `algred_atom`.

```
Definition algred_atom (a : SSA.atom) : SSA.eexp :=
  match a with
  | Avar v => Evar v
  | Aconst ty bs => Econst (bv2z ty bs)
  end.
Definition algred_instr te avn g (i : SSA.instr) : (N * seq
      SSA.ebexp) :=
  match i with
  | Iadd v a1 a2 =>
    let za1 := algred_atom a1 in
    let za2 := algred_atom a2 in
    (g, [:: Eeq (Evar v) (Ebinop Eadd za1 za2)])
  | Icast v ty a => algred_cast avn g v ty a (atyp a te)
  ...
  end.
```

Consider for example the instruction `Iadd v a1 a2` where `v` is a variable and `a1` and `a2` are two atoms. The execution of the instruction assigns `v` the bit-vector sum of the values of `a1` and `a2` computed by `addB`. We translate this execution to the equation `Eeq (Evar v) (Ebinop Eadd za1 za2)` where `za1` is `algred_atom a1` and `za2` is `algred_atom a2`. However, the execution does not correspond to roots of the equation when the bit-vector sum overflows. For example, consider two constant atoms both of type `Tuint 4`. Assume they have bit-vector values $(1111)_2$ and $(1000)_2$ (with least significant bit first) respectively. The two constant atoms have unsigned integer values 15 and 1 respectively. The bit-vector addition results in $(0000)_2$, which has an unsigned integer value 0. Obviously $15+1 \neq 0$. Thus to make our algebraic reduction sound, over- and under-flows must be avoided. We say that a specification `s` is *algebraically sound*, defined as the proposition `ssa_spec_algsnd s`, if and only if there is neither over- nor under-flow during the execution of the program in the specification. As checking over- and under-flows requires bit-accurate analysis, the establishment of `ssa_spec_algsnd s` is carried out in our range reduction in Section 6.2.

Let `v` be a variable, `ty` be a CL type, `a` be an atom, `te` be a type environment, and `aty` be the CL type of `a` under `te`. Consider for another example the algebraic reduction of the instruction `Icast v ty a` under `te`. Assume the target CL type `ty` is `Tuint wv` and `aty` is `Tuint wa` where `wv` and `wa` are two natural numbers. The execution of the instruction is translated to the equation `algred_cast avn g v ty a aty` where `avn` and `g` are used to generate fresh variables.

```
Definition algred_cast avn g v ty a aty :=
  match ty, aty with
  | Tuint wv, Tuint wa =>
    if wv ≥ wa then (g, [:: Eeq (evar v) (algred_atom a)])
    else let discarded := (avn, g) in
          let g' := N.succ g in
          (g', [:: algred_split discarded v (algred_atom a)
                wv])
  ...
  end.
```

If the width `wv` is greater than or equal to `wa` (`wv >= wa`), then the value of `a` can be represented in the CL type `ty` perfectly and thus the equation `Eeq (evar v) (algred_atom a)` must hold. Otherwise, only a part of the value can be represented in `ty`. In the latter case, there must be some value, denoted by the fresh variable `discarded`, such that the polynomial equation $\texttt{a} + \texttt{discarded} \times 2^{\texttt{wv}} = \texttt{v}$ holds. This polynomial equation is represented as the algebraic predicate `algred_split discarded v (algred_atom a) wv`. For example, consider casting a constant atom $(1101)_2$ of type `Tuint 4` to a target type `Tuint 2`. The casted value is the lower 2 bits $(11)_2$ of the atom (see ucastB in Section 5). While the atom has the unsigned integer value 11, the casted value is 3 in the unsigned representation. We have the equation $11 + (-2) \times 2^2 = 3$, that is, the integer value of discard is -2 (which is the negation of the unsigned value of the higher 2 bits $(01)_2$ of the atom).

The correctness of our algebraic reduction is guaranteed by the following soundness lemma:

```
Lemma algred_espec_sound (o : options) (s : SSA.spec) :
  well_formed_ssa_spec s → ssa_spec_algsnd s →
  valid_rep (algred_espec o
    (new_svar_spec s) (espec_of_spec s)) →
  valid_espec (espec_of_spec s).
```

where well_formed_ssa_spec checks if a specification is a well-formed specification in SSA. The lemma algred_espec_sound states that if a well-formed specification s in SSA is algebraically sound (ssa_spec_algsnd s) and the root entailment problem reduced from the specification holds, i.e. valid_rep (algred_espec o (new_svar_spec s) (espec_of_spec s)), then the algebraic specification espec_of_spec s is valid. Well-formedness ensures that the source atoms in an instruction have compatible CL types. See [17] for more details of well-formedness.

To prove this lemma, we have to construct an integer store (of type ZS.t) from the terminating store (of type S.t) of the program execution so that the premise of the root entailment problem holds in the integer store. Such an integer store is constructed by converting the bit-vector values of variables in the store to an integer value through bv2z. However this is not enough because there may be fresh variables created for Icast instructions in the premise but neither in the specification nor in the store. Extra proof effort is made to set the integer values of the fresh variables properly. Our algebraic reduction is sound but incomplete because program executions correspond to roots of the constructed system of (modular) equations but not vice versa. It remains to show how to solve a root entailment problem.

A root entailment problem of type rep is solved by an external CAS in verify_rep (or its parallel version verify_rep_list).

```
Definition verify_imp (ip : imp) : bool :=
  let '(_, _, ps, m, q) := zpexprs_of_imp ip in
  let (cs, c) := ext_solve_imp ps q m in
  validate_imp_answer ps m q cs c.
Definition verify_rep o (rp : rep) : bool :=
  if rewrite_assignments o
  then all verify_imp (imps_of_rep_simplified o rp)
  else all verify_imp (imps_of_rep rp).
```

The function verify_rep converts a root entailment problem to ideal membership problems through imps_of_rep (or imps_of_rep_simplified with rewriting) based on the approach in [21, 36], invokes the external CAS to solve all ideal membership problems through ext_solve_imp, and then verifies the answers from the CAS through the validator validate_imp_answer. If the answers from the CAS are successfully verified by validate_imp_answer, the root entailment problem holds. The correctness of verify_rep and its parallel version verify_rep_list is provided by the following lemmas.

```
Lemma verify_rep_sound o (rp : rep) :
  verify_rep o rp → valid_rep rp.
Lemma verify_rep_list_sound o (rp : rep) :
  verify_rep_list o rp → valid_rep rp.
```

6.2 Range Specification Verification

The range reduction converts bit-accurate verification problems to SMT QF_BV queries. In verify_ssa, two bit-accurate verification problems are reduced from a specification in SSA and solved through verify_rspec_algsnd. One bit-accurate verification problem is the validity of the range part of the specification and the other is the algebraic soundness of the specification. While the former problem is reduced through rngred_rspec, the latter problem is reduced through rngred_algsnd.

```
Definition rngred_rspec_algsnd (s : SSA.spec) : seq QFBV.
    bexp :=
  (rngred_rspec s) ++ (rngred_algsnd s).
Definition verify_rspec_algsnd (s : SSA.spec) : bool :=
  let fE := program_succ_typenv (sprog s) (sinputs s) in
  let es := simplify_bexp (rngred_rspec_algsnd s) in
  let '(_, _, _, cnfs) :=
    bb_hbexps_cache fE (map QFBVHash.hash_bexp es) in
  ext_all_unsat cnfs.
```

To get the benefit of parallel computation, instead of constructing a large SMT QF_BV query for both bit-accurate verification problems, our range reduction constructs an SMT QF_BV query for each of the atomic range predicates to be verified and soundness conditions of instructions. The SMT QF_BV queries are then solved by the certified SMT QF_BV solver CoqQFBV, which bit blasts a query into a satisfiability problem, invokes a SAT solver to solve the satisfiability problem, and then verifies the satisfiable assignments or proof of unsatisfiability returned by the SAT solver. We observe that solving SMT QF_BV queries one by one parallelly using CoqQFBV is still very slow due to bit blasting multiple times the same QF_BV expressions representing the program execution of the specification. To prevent this bottleneck, we bit blast all SMT QF_BV queries in bb_hbexps_cache and store the results in a cache. For expressions and predicates that have been bit blasted, we simply find the results from the cache. During the reduction, SMT QF_BV queries are simplified in simplify_bexp. It remains to show how the range reduction is applied in rngred_rspec and rngred_algsnd.

The function rngred_rspec first converts the input specification s to a range specification rspec_of_spec s of type rspec and then reduces the validity of rspec_of_spec s to SMT QF_BV queries through functions bexp_rbexp and bexp_program. Intuitively, for an atomic range predicate e in the post-condition of a range specification s, rngred_rspec constructs a QF_BV predicate qpre equivalent to rspre s through bexp_rbexp, QF_BV predicates qprog equivalent to the execution of rsprog s through bexp_program, a QF_BV predicate qpost equivalent to e through bexp_rbexp, and then an SMT QF_BV query checking if qpost is implied by the conjunction of qpre and qprog. Since the range predicates in typed CRYPTOLINE are a subset of the QF_BV predicates in COQQFBV, the conversion from rbexp to QFBV.bexp is straightforward. The function bexp_program converts a program to QF_BV predicates instruction by instruction through bexp_instr. The function bexp_instr basically follows the semantics of eval_instr but changes bit-vector operations to appropriate QF_BV expressions and makes equalities instead of assignments. For example, consider the instruction Iadd v a1 a2 where v is a variable and a1 and a2 are two atoms. The execution of the instructions assigns v the bit-vector sum of the values of a1 and a2 computed by the *coq-nbits* function addB. The QF_BV predicate constructed by bexp_instr for the instruction is then qfbv_eq (qfbv_var v) (qfbv_add (qfbv_atom a1) (qfbv_atom a2)). Given a variable v, a CL type ty, and a bit-vector n, qfbv_atom maps Avar v and Aconst _ n to qfbv_var v and qfbv_const n respectively.

The function rngred_algsnd reduces the soundness conditions of a specification to SMT QF_BV queries. Each instruction in the specification has its soundness condition computed by bexp_instr_algsnd and represented as a QF_BV predicate. For each instruction in the specification, we construct an SMT QF_BV query checking if the soundness condition of the instruction is implied by qpre and qprog, where qpre and qprog are constructed through bexp_rbexp and bexp_program respectively in the same way as aforementioned.

```
Definition bexp_atom_uaddB_algsnd a1 a2 : QFBV.bexp :=
  qfbv_lneg (qfbv_uaddo (qfbv_atom a1) (qfbv_atom a2)).
Definition bexp_atom_saddB_algsnd a1 a2 : QFBV.bexp :=
  qfbv_lneg (qfbv_saddo (qfbv_atom a1) (qfbv_atom a2)).
Definition bexp_atom_addB_algsnd E a1 a2 : QFBV.bexp :=
  let 'aty := atyp a1 E in
  if Typ.is_unsigned aty then bexp_atom_uaddB_algsnd a1 a2
  else bexp_atom_saddB_algsnd a1 a2.
...
Definition bexp_instr_algsnd E (i : instr) : QFBV.bexp :=
  match i with
  | Iadd _ a1 a2 => bexp_atom_addB_algsnd E a1 a2
  ...
  end.
```

For addition, subtraction, multiplication, and shifting operations that have potential over- and under-flow issues, we extend COQQFBV with over- and under-flow QF_BV predicates and their bit blasting rules with correctness proof certified by COQ. Consider for example the instruction Iadd v a1 a2 where the variable v, and the atoms a1 and a2 are of the same unsigned CL type. The instruction Iadd v a1 a2 is algebraically sound if the SMT QF_BV predicate qfbv_lneg (qfbv_uaddo (qfbv_atom a1) (qfbv_atom a2)) holds where qfbv_lneg constructs a logical negation in SMT QF_BV. The function qfbv_uaddo constructs our extended predicate for unsigned addition overflow, which states that the carry of the unsigned addition equals one.

Our range reduction is sound and complete by the following lemmas.

```
Lemma verify_rspec_algsnd_sound (s : SSA.spec) :
  well_formed_ssa_spec s → verify_rspec_algsnd s →
  valid_rspec (rspec_of_spec s) ∧ ssa_spec_algsnd s.
Lemma verify_rspec_algsnd_complete (s : SSA.spec) :
  well_formed_ssa_spec s →
  valid_rspec (rspec_of_spec s) → ssa_spec_algsnd s →
  verify_rspec_algsnd s.
```

The soundness lemma verify_rspec_algsnd_sound states that if verify_rspec_algsnd s is true for a well-formed SSA specification s, then the range specification of s is valid and s is algebraically sound. The completeness lemma verify_rspec_algsnd_complete guarantees that a counterexample found by an SMT QF_BV solver is indeed a violation of the specification. Note that BVCRYPTOLINE does not provide any completeness.

6.3 Correctness

We build the correctness of our top-level verification function verify_dsl. By valid_spec_split and algred_espec_sound, we prove the following lemma.

```
Theorem algred_spec_sound (o : options) (s : SSA.spec) :
  well_formed_ssa_spec s → ssa_spec_algsnd s →
  valid_rspec (rspec_of_spec s) →
  valid_rep (algred_espec o
    (new_svar_spec s) (espec_of_spec s)) →
  valid_spec s.
```

The lemma algred_spec_sound states that to verify a well-formed specification in SSA, it is sufficient to verify its algebraic soundness, validity of the range part, and the validity of the algebraic part. By algred_spec_sound, verify_rspec_algsnd_sound, verify_rep_sound, and verify_rep_list_sound, we have the following soundness theorem, guaranteeing the validity of a well-formed specification in SSA if it is successfully verified by verify_ssa.

```
Theorem verify_ssa_sound (o : options) (s : SSA.spec) :
  well_formed_ssa_spec s → verify_ssa o s →
  SSA.valid_spec s.
```

Additionally, our SSA transformation SSA.ssa_spec preserves validity and well-formedness.

```
Theorem ssa_spec_sound (s : DSL.spec) :
  SSA.valid_spec (SSA.ssa_spec s) → DSL.valid_spec s.
Theorem ssa_spec_well_formed (s : DSL.spec) :
  DSL.well_formed_spec s →
  well_formed_ssa_spec (SSA.ssa_spec s).
```

Finally, by `verify_ssa_sound`, `ssa_spec_sound`, and `ssa_spec_well_formed`, we prove the soundness of the verification function `verify_dsl`.

```
Theorem verify_dsl_sound (o : options) (s : DSL.spec) :
  DSL.well_formed_spec s → verify_dsl o s →
  DSL.valid_spec s.
```

The main theorem `verify_dsl_sound` guarantees that the input specification is valid if it is well-formed and verified by `verify_dsl`.

7 Evaluation

We evaluate COQCRYPTOLINE on benchmarks from four industrial security libraries BITCOIN [35], BORINGSSL [15,19], NSS [25] and OPENSSL [31]. A case study on the post-quantum key encapsulation mechanism scheme KYBER is also evaluated. We compare COQCRYPTOLINE against the uncertified verification tool CRYPTOLINE [17]. Both tools use the computer algebra system SINGULAR for the algebraic technique [20], but CRYPTOLINE does not certify answers. For the SMT-based technique, COQCRYPTOLINE invokes the certified SMT QF_BV solver COQQFBV [34]. CRYPTOLINE however uses the efficient but uncertified SMT solver BOOLECTOR [28]. BVCRYPTOLINE is not in our evaluation because 43 out of the 52 benchmarks are not supported by BVCRYPTOLINE. Evaluation is performed on an Ubuntu 20.04 machine with a 3.20GHz CPU and 1TB RAM. For each benchmark, the external solvers SINGULAR and COQQFBV run concurrently with 20 threads while other parts run sequentially.

In this evaluation, 52 C implementations of field and group operations for elliptic curves secp256k1 (BITCOIN) and Curve25519 (BORINGSSL, NSS, OPENSSL) are verified. Moreover, the C program for KYBER Number-Theoretic Transform (NTT) in PQCLEAN [33] is verified.

Field and Group Operations in Security Libraries The 52 programs for various field and group operations in secp256k1 and Curve25519 were reported in [17]. For those written in assembly, we obtain CRYPTOLINE programs by automatic extraction (Section 3). For others written in C, we did not verify their C source codes. Rather, we extract CRYPTOLINE programs from compiler intermediate representations after machine-independent optimizations automatically [24]. Subsequently, these CRYPTOLINE programs reflect actual computation more accurately than original C source codes. We verify whether every function correctly implements the corresponding field or group operation and outputs results in expected bounds. Annotations for these programs are mostly straightforward. COQCRYPTOLINE verifies almost all programs with certificates. Some group operations (x25519_scalar_mult_generic, point_add_and_double, and x25519_scalar_mult) are verified but not fully certified. Each of them has three algebraic post-conditions; COQCRYPTOLINE verifies all three algebraic post-conditions but only certifies two of them. Stack overflow exception is raised when our certificate checker validates answers from SINGULAR.

Table 1: Experimental Results

Fcn	L_{CL}	T_{CCL}	T_{CL}	*Fcn*	L_{CL}	T_{CCL}	T_{CL}
bitcoin/asm/secp256k1_fe_*							
mul_inner					158	76.6	3.6
sqr_inner					138	37.1	2.2
bitcoin/field/secp256k1_fe_*							
add	8	0.1	0.2	mul_inner	123	87.3	3.0
cmov	45	3.1	0.5	mul_int	9	2.4	0.2
negate	13	1.0	0.3	sqr_inner	111	57.7	2.0
from_storage					27	0.4	0.3
normalize					41	169.1	65.5
normalize_var					47	132.3	61.6
normalize_weak					14	81.2	16.2
secp256k1_fe_normalizes_to_zero					47	190.0	62.3
bitcoin/group/							
secp256k1_ge_neg					29	0.5	0.5
secp256k1_ge_from_storage					51	1.0	0.5
secp256k1_gej_double_var.part.14					871	2435.4	34.3
bitcoin/scalar/secp256k1_scalar_*							
add	111	3.8	1.4	mul_512	296	132.0	5.3
eq	30	0.7	0.1	negate	40	32.6	0.6
mul	808	456.0	13.9	reduce	103	2.5	1.0
sqr	820	353.2	12.4	sqr_512	308	52.7	5.4
secp256k1_scalar_reduce_512					515	124.7	5.6
boringssl/fiat_curve25519/fe_*							
add	8	0.1	0.2	mul_impl	106	68.1	3.1
sub	13	0.2	0.3	sqr_impl	88	30.7	1.7
fe_mul121666					50	1.7	0.7
x25519_scalar_mult_generic					965	3317.2	341.0
boringssl/fiat_curve25519_x86/fe_*							
add	13	0.2	0.5	mul_impl	348	208.8	6.6
sub	23	0.4	0.8	sqr_impl	272	86.9	4.9
fe_mul121666					87	2.8	1.3
x25519_scalar_mult_generic					3008	14164.5	1106.7
nss/Hacl_Curve25519_51/							
fadd0	8	0.2	0.3	fsub0	13	0.3	0.4
fmul0	120	175.2	34.3	fmul1	60	18.3	0.8
fsqr0	91	95.1	4.2	fsqr20	179	258.7	6.1
fmul20					221	590.7	24.0
point_add_and_double					1057	7507.4	1302.5
point_double					524	1936.8	53.3
openssl/curve25519/fe51_*							
add	8	0.1	0.3	sub	18	0.2	0.2
mul	104	107.8	2.5	sq	86	51.8	1.4
fe51_mul121666					51	1.7	0.7
x25519_scalar_mult					948	5669.9	614.8
PQClean/kyber/NTT							
PQCLEAN_KYBER512_CLEAN_ntt					5421	122931.6	741.5

KYBER and NTT (Number Theoretic Transform) multiplications Crystals-KYBER [7], a round 3 finalist key establishment method (KEM) of the NIST postquantum standardization process [30] is a lattice-based cryptosystem. In such KEMs, aside from symmetric-key primitives (e.g. SHA-3), the critical steps are modular polynomial multiplications, which are conducted using the NTT. This is an analogue of fast-Fourier transform (FFT) for finite fields.

NTTs are based on the isomorphism from $\mathbb{F}_q[X]/(X^{2n}-c^2)$ to $\mathbb{F}_q[X]/(X^n-c)\times\mathbb{F}_q[X]/(X^n+c)$, or $f(X)+X^n g(X)\mapsto (f(X)+cg(X), f(X)-cg(X))$ for $\deg f, g < n$. This splits into many $(f_i, g_i)\mapsto(f_i+cg_i, f_i-cg_i)$ maps — Cooley-Tukey butterflies [14]. This repeats with a square root for c in $\mathbb{F}_q$.

The usual textbook NTT concludes at linear $X-\omega$ factors. And multiplication between two images of the NTT map is just pairwise multiplication in $\mathbb{F}_q \cong \mathbb{F}_q[X]/(X-\omega)$. In KYBER, the ring is $\mathbb{Z}_{3329}[X]/(X^{256}+1)$ where -1 has a 128-th but no 256-th principal root. So via 7 layers of in-place CT butterflies [14], the KYBER NTT maps a polynomial of degree 255, to 128 linear polynomials (each modulo a different $X^2-\zeta_j$, where ζ_j's are the principal 256-th roots of unity):

$$\mathbb{Z}_q[X]/(X^{256}+1) \overset{NTT}{\leftrightarrow} \mathbb{Z}_q[X]/(X^2-\zeta_0)\times\cdots\times\mathbb{Z}_q[X]/(X^2-\zeta_{127})$$

This is called an "incomplete NTT". One can compute a modular polynomial product via two such incomplete NTTs, a pairwise product of linear polynomials modulo various $X^2-\omega$, and then applying an inverse incomplete NTT.

In KYBER, polynomial coefficients are elements from the residue system modulo $q = 3329$. Addition and multiplication in KYBER NTT are therefore modular arithmetic over the residue system. Multiplying by a constant c in KYBER is usually "signed Montgomery": $ac \equiv (ac' - (ac'' \bmod R)q)/R \pmod q$, with R a power of 2 (usu. 2^{16}), $c' = cR \bmod q$, and $c'' = c'(q^{-1} \bmod R) \bmod R$. The division is exact because $ac''q \equiv ac' \pmod R$, for a result between $\pm q$ if $|a| < q/2$.

We verify whether the reference C implementation of KYBER NTT correctly computes 128 linear polynomials for any input polynomial of degree 255 with coefficients in the residue system modulo q. Let $F = \sum_{k=0}^{255} f_k X^k$ denote the input to the KYBER NTT, each coefficient $f_k < q$ represented as a 16-bit signed integer in an array of size 256. Let $\sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k}X^k$ be the j-th polynomial obtained at the end of layer i with $0 \le i \le 6$ and $0 \le j < 2^{i+1}$. Using signed Montgomery multiplications, we need no mod-q reductions during 7 layers of CT butterflies. We verify that each layer correctly implements CT butterflies by specifying algebraic and range post-conditions.

The following algebraic post-conditions are verified at the end of layer 0, with $0 \le k < 128$:

$$\begin{aligned} g_{0,0,k} &\equiv f_k + \zeta_{0,0} f_{k+128} \mod q, \\ \text{and } g_{0,1,k} &\equiv f_k - \zeta_{0,0} f_{k+128} \mod q. \end{aligned}$$

And at the end of layers 1 to 6, we specify the post-conditions

$$\begin{aligned} g_{i,2j,k} &\equiv g_{i-1,j,k} + \zeta_{i,j} g_{i-1,j,k+256/2^{i+1}} \mod q, \\ \text{and } g_{i,2j+1,k} &\equiv g_{i-1,j,k} - \zeta_{i,j} g_{i-1,j,k+256/2^{i+1}} \mod q, \end{aligned}$$

with $0 \le j < 2^i$ and $0 \le k < 256/2^{i+1}$. $\zeta_{i,j}$'s are the factors used by CT butterflies at layer i. The range pre-condition $-q \le f_k < q$ is specified for layer 0. At the end of layer i, the range post-conditions we verified are

$$-(3+i)q \le g_{i,j,k} < (3+i)q,$$

for $0 \le j < 2^{i+1}$ and $0 \le k < 256/2^{i+1}$. And the range post-conditions of layer $i-1$ are required as the range pre-conditions for layer i with $1 \le i \le 6$.

Results Table 1 shows evaluation results. The column L_{CL} denotes the number of CRYPTOLINE instructions. T_{CCL} and T_{CL} are respectively the running time of COQCRYPTOLINE and CRYPTOLINE in seconds. Most functions are verified by COQCRYPTOLINE within 10 minutes. The hardest one, KYBER NTT, is also verified. CRYPTOLINE verifies KYBER NTT in 741.5 seconds (≈13 minutes) but COQCRYPTOLINE needs 122931.6 seconds (≈1.5 days). In all cases, external solvers take much more time than COQCRYPTOLINE OCAML program does. Between external solvers, the verification of algebraic properties in KYBER NTT calls SINGULAR 1792 times and takes 96.23% of the time; the verification of range properties invokes COQQFBV 8064 times and spends 3.70% of the time. In contrast, the other 52 functions call SINGULAR 60 times in total and takes only 0.13% of the time; they invoke COQQFBV 7074 times using 99.78% of the time.

8 Conclusion

Adopting recent developments in certified verification, we build the certified automated verification tool COQCRYPTOLINE for cryptographic programs. We certify our proof of correctness for the COQCRYPTOLINE verification algorithm fully in COQ with SSREFLECT. For efficiency, COQCRYPTOLINE employs external tools and validates their answers with certificates. We evaluate COQCRYPTOLINE on benchmarks from industrial security libraries (BITCOIN, BORINGSSL, NSS, OPENSSL) and a post-quantum cryptography standard finalist (KYBER). Certified verification needs to validate its results and hence can be less efficient. In our experiments, COQCRYPTOLINE verifies most cryptographic programs with certificates in reasonable time (10 minutes). The most complicated benchmark on KYBER number-theoretic transform needs 1.5 days and may still be faster than certified manual proofs. To our knowledge, this is the first certified verification on operations of the elliptic curve secp256k1 used in BITCOIN, and the reference implementation of KYBER number-theoretic transform. Certified verification with COQCRYPTOLINE is perhaps useful for real cryptographic programs.

References

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

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

[3] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The

case of BBS. *Science of Computer Programming*, 77(10–11):1058–1074, 2012.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823. ACM, 2017.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Gregoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. A machine-checked proof of security for aws key management service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 63–78, 2019.

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

[7] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [30], 2020. https://pq-crystals.org/kyber/.

[8] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. Easypqc: Verifying post-quantum cryptography. Cryptology ePrint Archive, Report 2021/1253, 2021. https://ia.cr/2021/1253.

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

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

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[12] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, pages 917–934. USENIX Association, 2017.

[13] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 software. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 299–309. ACM, 2014.

[14] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[15] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE, 2019.

[16] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, January 2019.

[17] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1606. ACM, 2019.

[18] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[19] Google. Boringssl, 2021. https://boringssl.googlesource.com/boringssl/.

[20] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, 2002.

[21] John Harrison. Automating elementary number-theoretic proofs using gröbner bases. In Frank Pfenning, editor, *International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2007.

[22] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *4th International Conference on Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2012.

[23] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the wireguard virtual private network protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 231–246, 2019.

[24] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic in cryptographic c programs. In Julia Lawall and Darko Marinov, editors, *IEEE/ACM International Conference on Automated Software Engineering*, pages 552–564. IEEE, 2019.

[25] Mozilla. Network security services, 2021. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[26] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2013.

[27] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2007.

[28] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[29] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[31] OpenSSL. OpenSSL library. https://github.com/openssl/openssl, 2021.

[32] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives. In Sven Schewe and Lijun Zhang, editors, *International Conference on Concurrency Theory*, LIPIcs, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[33] PQClean. The PQClean project. https://github.com/PQClean/PQClean, 2021.

[34] Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. CoqQFBV: A scalable certified SMT quantifier-free bit-vector solver. In Rustan Leino and Alexandra Silva, editors, *International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2021.

[35] The Bitcoin Developers. Bitcoin source code, 2021. https://github.com/bitcoin/bitcoin.

[36] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1973–1987. ACM, 2017.

[37] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2020. ACM, 2017.

[38] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.

A `ecp_nistz256_mul_montx`

Figure 3 shows the complete CRYPTOLINE specification for the assembly subroutine `ecp_nistz256_mul_montx` in OPENSSL.

B Typed CRYPTOLINE

CRYPTOLINE is a domain specific language for modeling cryptographic assembly programs [17,32,36].

B.1 Syntax

Figure 4 gives the formal CRYPTOLINE syntax in COQCRYPTOLINE. A type is represented by `Tuint w` or `Tsint w` for a natural number `w` in COQCRYPTOLINE An *atom* is of the form `Avar var` or `Aconst t bits` where `var` is a variable, `t` is a type, and `bits` a bit-vector. `Imov v a` assigns the value of the source atom `a` to the destination variable `v`. The conditional move instruction `Icmov v c a1 a2` assigns the destination variable `v` the value of either source atoms `a1` and `a2` by the flag `c`. Arithmetic instructions such as addition (`Iadd` and `Iadds`), addition with carry (`Iadc` and `Iadcs`), subtraction (`Isub` and `Isubb`), subtraction with borrow (`Isbb` and `Isbbs`), subtraction with carry (`Isbc` and `Isbcs`), half-multiplication (`Imul`), and full multiplication (`Imull` and `Imulj`) are supported. Additional flags (such as carry and borrow flags) are set in `Iadds`, `Iadcs`, `Isubb`, `Isbbs`, and `Isbcs`. While `Imull vh vl a1 a2` splits the result of a full multiplication of `a1` and `a2` into the high bits `vh` and the low bits `vl`,

```
proc main
(uint64 a0, uint64 a1, uint64 a2, uint64 a3,
 uint64 b0, uint64 b1, uint64 b2, uint64 b3,
 uint64 m0, uint64 m1, uint64 m2, uint64 m3) =
{ and [ m0 = 0xffffffffffffffff,
        m1 = 0x00000000ffffffff,
        m2 = 0x0000000000000000,
        m3 = 0xffffffff00000001 ]
&&
  and [ m0 = 0xffffffffffffffff@64,
        m1 = 0x00000000ffffffff@64,
        m2 = 0x0000000000000000@64,
        m3 = 0xffffffff00000001@64,
        limbs 64 [a0, a1, a2, a3] <u
            limbs 64 [m0, m1, m2, m3],
        limbs 64 [b0, b1, b2, b3] <u
            limbs 64 [m0, m1, m2, m3] ] }

mov L0x7fffffffd9b0 a0; mov L0x7fffffffd9b8 a1;
mov L0x7fffffffd9c0 a2; mov L0x7fffffffd9c8 a3;
mov L0x7fffffffd9d0 b0; mov L0x7fffffffd9d8 b1;
mov L0x7fffffffd9e0 b2; mov L0x7fffffffd9e8 b3;

mov L0x55555557c000 0xffffffffffffffff@uint64;
mov L0x55555557c008 0x00000000ffffffff@uint64;
mov L0x55555557c010 0x0000000000000000@uint64;
mov L0x55555557c018 0xffffffff00000001@uint64;

(* ecp_nistz256_mul_montx STARTS *)
mov rdx L0x7fffffffd9d0;
mov r9 L0x7fffffffd9b0;
mov r10 L0x7fffffffd9b8;
mov r11 L0x7fffffffd9c0;
mov r12 L0x7fffffffd9c8;
mull r9 r8 rdx r9;
mull r10 rcx rdx r10;
mov r14 0x20@uint64;
mov r13 0@uint64;
clear carry;
clear overflow;
mull r11 rbp rdx r11;
mov r15 L0x55555557c018;
adcs carry r9 r9 rcx carry;
mull r12 rcx rdx r12;
mov rdx r8;
adcs carry r10 r10 rbp carry;
assert r14=32 && true;
split ddc rbp r8 32;
shl rbp rbp 32;
adcs carry r11 r11 rcx carry;
assert r14=32 && true;
split rcx dc r8 32;

assert true && rcx=ddc;
assume rcx=ddc && true;

adc r12 r12 0x0@uint64 carry;
adds carry r9 r9 rbp;
adcs carry r10 r10 rcx carry;
mull rbp rcx rdx r15;
mov rdx L0x7fffffffd9d8;
adcs carry r11 r11 rcx carry;
adcs carry r12 r12 rbp carry;
adc r13 r13 0x0@uint64 carry;

mov r8 0@uint64;
clear carry;
clear overflow;
mull rbp rcx rdx L0x7fffffffd9b0;
adcs carry r9 r9 rcx carry;
adcs overflow r10 r10 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9b8;
adcs carry r10 r10 rcx carry;
adcs overflow r11 r11 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c0;
adcs carry r11 r11 rcx carry;
adcs overflow r12 r12 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c8;
mov rdx r9;
adcs carry r12 r12 rcx carry;
assert r14=32 && true;
split ddc rcx r9 32;
shl rcx rcx 32;
adcs overflow r13 r13 rbp overflow;
assert r14=32 && true;
split rbp dc r9 32;

assert true && rbp=ddc;
assume rbp=ddc && true;

adcs carry r13 r13 r8 carry;
adcs overflow r8 r8 r8 overflow;

assert true && and [carry=0@1,overflow=0@1];
assume and [carry=0,overflow=0] && true;

adc r8 r8 0x0@uint64 carry;
adds carry r10 r10 rcx;
adcs carry r11 r11 rbp carry;
mull rbp rcx rdx r15;
mov rdx L0x7fffffffd9e0;
adcs carry r12 r12 rcx carry;
adcs carry r13 r13 rbp carry;
adc r8 r8 0x0@uint64 carry;

mov r9 0@uint64;
clear carry;
clear overflow;
mull rbp rcx rdx L0x7fffffffd9b0;
adcs carry r10 r10 rcx carry;
adcs overflow r11 r11 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9b8;
adcs carry r11 r11 rcx carry;
adcs overflow r12 r12 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c0;
adcs carry r12 r12 rcx carry;
adcs overflow r13 r13 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c8;
mov rdx r10;
adcs carry r13 r13 rcx carry;
assert r14=32 && true;
split ddc rcx r10 32;
shl rcx rcx 32;
adcs overflow r8 r8 rbp overflow;
assert r14=32 && true;
split rbp dc r10 32;

assert true && rbp=ddc;
assume rbp=ddc && true;

adcs carry r8 r8 r9 carry;
adcs overflow r9 r9 r9 overflow;

assert true && and [carry=0@1,overflow=0@1];
assume and [carry=0,overflow=0] && true;

adc r9 r9 0x0@uint64 carry;
adds carry r11 r11 rcx;
adcs carry r12 r12 rbp carry;
mull rbp rcx rdx r15;
mov rdx L0x7fffffffd9e8;
adcs carry r13 r13 rcx carry;
adcs carry r8 r8 rbp carry;
adc r9 r9 0x0@uint64 carry;
mov r10 0@uint64;
clear carry;
clear overflow;
mull rbp rcx rdx L0x7fffffffd9b0;
adcs carry r11 r11 rcx carry;
adcs overflow r12 r12 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9b8;
adcs carry r12 r12 rcx carry;
adcs overflow r13 r13 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c0;
adcs carry r13 r13 rcx carry;
adcs overflow r8 r8 rbp overflow;
mull rbp rcx rdx L0x7fffffffd9c8;
mov rdx r11;
adcs carry r8 r8 rcx carry;
assert r14=32 && true;
split ddc rcx r11 32;
shl rcx rcx 32;
adcs overflow r9 r9 rbp overflow;
assert r14=32 && true;
split rbp dc r11 32;

assert true && rbp=ddc;
assume rbp=ddc && true;

adcs carry r9 r9 r10 carry;
adcs overflow r10 r10 r10 overflow;

assert true && and [carry=0@1,overflow=0@1];
assume and [carry=0,overflow=0] && true;

adc r10 r10 0x0@uint64 carry;
adds carry r12 r12 rcx;
adcs carry r13 r13 rbp carry;
mull rbp rcx rdx r15;
mov rbx r12;
mov r14 L0x55555557c008;
adcs carry r8 r8 rcx carry;
mov rdx r13;
adcs carry r9 r9 rbp carry;
adc r10 r10 0x0@uint64 carry;

nondet r12o@uint64; nondet r13o@uint64;
nondet r8o@uint64;  nondet  r9o@uint64;
nondet r10o@uint64;
assume and [ r12o=r12, r13o=r13, r8o=r8,
             r9o=r9, r10o=r10 ]
    && and [ r12o=r12, r13o=r13, r8o=r8,
             r9o=r9, r10o=r10 ];

mov eax 0@uint64;
clear carry;
clear overflow;
mov rcx r8;
sbbs carry r12 r12 0xffffffffffffffff@uint64 carry;
sbbs carry r13 r13 r14 carry;
sbbs carry r8 r8 0x0@uint64 carry;
mov rbp r9;
sbbs carry r9 r9 r15 carry;
sbbs carry r10 r10 0x0@uint64 carry;
cmov r12 carry rbx r12;
cmov r13 carry rdx r13;
mov L0x7fffffffd9f0 r12;
cmov r8 carry rcx r8;
mov L0x7fffffffd9f8 r13;
cmov r9 carry rbp r9;

assert true &&
       eqmod limbs 64 [r12, r13, r8, r9, 0@64]
             limbs 64 [r12o, r13o, r8o, r9o, r10o]
             limbs 64 [m0, m1, m2, m3, 0@64];
assume eqmod limbs 64 [r12, r13, r8, r9, 0]
             limbs 64 [r12o, r13o, r8o, r9o, r10o]
             limbs 64 [m0, m1, m2, m3, 0] && true;

mov L0x7fffffffda00 r8;
mov L0x7fffffffda08 r9;
(* ecp_nistz256_mul_montx ENDS *)

mov c0 L0x7fffffffd9f0;
mov c1 L0x7fffffffd9f8;
mov c2 L0x7fffffffda00;
mov c3 L0x7fffffffda08;

{ eqmod limbs 64 [0, 0, 0, 0, c0, c1, c2, c3]
        limbs 64 [a0, a1, a2, a3] *
        limbs 64 [b0, b1, b2, b3]
        limbs 64 [m0, m1, m2, m3]
&&
  limbs 64 [c0, c1, c2, c3] <u
      limbs 64 [m0, m1, m2, m3] }
```

Figure 3: CRYPTOLINE Model for `ecp_nistz256_mul_montx`

```
Inductive typ : Set :=
| Tuint : nat → typ | Tsint : nat → typ.
Inductive atom : Type :=
| Avar : var → atom | Aconst : typ → bits → atom.
Inductive instr : Type :=
| Imov : var → atom → instr
| Icmov : var → atom → atom → atom → instr
| Iadd : var → atom → atom → instr
| Iadds : var → var → atom → atom → instr
| Iadc : var → atom → atom → atom → instr
| Iadcs : var → var → atom → atom → atom → instr
| Isub : var → atom → atom → instr
| Isubc : var → var → atom → atom → instr
| Isubb : var → var → atom → atom → instr
| Isbb : var → atom → atom → atom → instr
| Isbbs : var → var → atom → atom → atom → instr
| Isbc : var → atom → atom → atom → instr
| Isbcs : var → var → atom → atom → atom → instr
| Imul : var → atom → atom → instr
| Imull : var → var → atom → atom → instr
| Imulj : var → atom → atom → instr
| Inot : var → typ → atom → instr
| Iand : var → typ → atom → atom → instr
| Ior : var → typ → atom → atom → instr
| Ixor : var → typ → atom → atom → instr
| Ishl : var → atom → nat → instr
| Isplit : var → var → atom → nat → instr
| Ijoin : var → atom → atom → instr
| Icshl : var → var → atom → atom → nat → instr
| Icast : var → typ → atom → instr
| Ivpc : var → typ → atom → instr
| Inondet : var → typ → instr
| Inop : instr | Iassume : bexp → instr.
```

Figure 4: CRYPTOLINE Types, Instructions and Programs

```
Inductive eunop : Set := Eneg.
Inductive ebinop : Set := Eadd | Esub | Emul.
Inductive eexp : Type :=
| Evar : var → eexp | Econst : Z → eexp
| Eunop : eunop → eexp → eexp
| Ebinop : ebinop → eexp → eexp → eexp.
Inductive ebexp : Type :=
| Etrue | Eeq : eexp → eexp → ebexp
| Eeqmod : eexp → eexp → eexp → ebexp
| Eand : ebexp → ebexp → ebexp.

Inductive runop : Set := Rnegb | Rnotb.
Inductive rbinop : Set :=
| Radd | Rsub | Rmul | Rumod | Rsrem | Rsmod
| Randb | Rorb | Rxorb.
Inductive rcmpop : Set :=
| Rult | Rule | Rugt | Ruge | Rslt | Rsle | Rsgt | Rsge.
Inductive rexp : Type :=
| Rvar : var → rexp | Rconst : nat → bits → rexp
| Runop : nat → runop → rexp → rexp
| Rbinop : nat → rbinop → rexp → rexp → rexp
| Ruext : nat → rexp → nat → rexp
| Rsext : nat → rexp → nat → rexp.
Inductive rbexp : Type :=
| Rtrue | Req : nat → rexp → rexp → rbexp
| Rcmp : nat → rcmpop → rexp → rexp → rbexp
| Rneg : rbexp → rbexp | Rand : rbexp → rbexp → rbexp
| Ror : rbexp → rbexp → rbexp.

Definition bexp : Type := ebexp * rbexp
```

Figure 5: CRYPTOLINE Algebraic and Range Predicates

`Imulj v a1 a2` stores the result in one variable `v` of a wider width. Moreover, bitwise logical operations `Inot`, `Iand`, `Ior`, and `Ixor` are allowed. The `Ishl v a n` instruction shifts the bit vector value `a` to the left by `n`. The `Isplit vh vl a n` instruction decomposes the value of the source atom `a` at position `n`, stores the resulting high bits in `vh`, and stores the resulting low bits in `vl`. The `Ijoin v a1 a2` instruction concatenates values of source atoms `a1` and `a2` and puts the concatenation in the destination variable `v`. The concatenate-shift-left instruction `Icshl vh vl a1 a2 n` concatenates values of source atoms `a1` and `a2`, shifts the concatenated value to the left by `n`, and decomposes the result into two destination variables `vh` and `vl`. The `Icast v t a` instruction casts the value of the source atom `a` into the designated type `t`. The value-preserving cast `Ivpc v t a` also casts the value of `a` to `t` but requires the value is preserved after casting. The non-deterministic instruction `Inondet v t` assigns the destination variable `v` an arbitrary value in the designated type `t`. For verification purposes, COQCRYPTOLINE allows programmers assumptions about executions. The `Iassume e` instruction ensures that the designated predicate `e` hold in all executions. `Inop` is the null instruction. A `program` is a sequence of instructions.

The formal syntax of a predicate `bexp` in `Iassume` is shown in Figure 5. An algebraic expression `eexp` in algebraic predicates is a variable (`Evar var`), a constant (`Econst z`), the negation of an algebraic expression (`Eneg`), the sum (`Eadd`), difference (`Esub`), or product (`mul`) of two algebraic expressions. Note that the constant `z` in an algebraic expression has type `Z`, which is the type of unbounded integers in COQ. Atomic algebraic predicates include equality (`Eeq e1 e2`) and modular equality (`Eeqmod e1 e2 m`) over algebraic expressions `e1`, `e2`, and `m`. An atomic algebraic `ebexp` is an atomic algebraic predicate or a conjunction (`Eand`) of algebraic predicates.

A range expression `rexp` is a variable (`Rvar`), a constant (`Rconst n bits`), the arithmetic negation (`Rneg`), bitwise inversion (`Rnot`), addition (`Radd`), subtraction (`Rsub`), multiplication (`Rmul`), unsigned remainder (`Rumod`), signed remainder (`Rsrem` and `Rsmod`), bitwise AND (`Rand`), bitwise OR (`Ror`), or bitwise XOR (`Rxor`) over range expressions. Constants in range expressions are bit-vectors of bounded lengths. Atomic range predicates are equality (`Req`), signed or unsigned comparisons (`Rcmp`) over range expressions `rexp`. A range predicate `rbexp` is an arbitrary Boolean expression (`Rneg`, `Rand`, and `Ror`) over atomic range predicates.

B.2 Semantics

Figure 6-8 show the formal semantics of instructions and predicates defined in CoqCryptoLine. Recall that `eval_atom a s` evaluates the atom `a` on the store `s`. Let `v`, `v'` be variables, `bits`, `bits'` bit-vectors, and `s`, `t` stores. The proposition `S.Upd v bits s t` denotes that the store `t` is obtained by updating the value of the variable `v` with the bit vector `bits` in the store `s`; `S.Upd2 v bits v' bits' s t` denotes that the store `t` is obtained by updating the values of the variables `v` and `v'` with the bit-vectors `bits` and `bits'` in the store `s`.

Because of `Inondet`, our formal semantics is relational. The predicate `eval_instr te i s t` denotes that the store `t` can be reached from the store `s` after executing the instruction `i` in the type environment `te`. Concretely, `eval_instr te (Imov v a) s t` holds if `S.Upd v (eval_atom a s) s t` holds. That is, `t` is obtained by updating the variable `v` with the value of the atom `a` in the `s`. There are two cases for the `Icmov v c a1 a2` instruction. If `eval_atom c s` is true and `v` is updated with `eval_atom a1 s` in `t`, then `eval_instr te (Icmov v c a1 a2) s t` holds (`EIcmovT`). If `eval_atom c s` is false, `v` needs to be updated with `eval_atom a2 s` in `t` (`EIcmovF`). `eval_instr te Inop s s` always holds.

The instruction `Iadd v a1 a2` uses the bit-vector function `addB` from *coq-nbits* to update `v` with the sum of the `eval_atom a1 s` and `eval_atom a2 s`. `Iadds c v a1 a2` moreover sets the bit variable `c` to the carry of the sum. The *coq-nbits* function `carry_addB` computes the carry. The instruction `Iadc v a1 a2 y` uses the bit-vector function `adcB` to compute the sum of `eval_atom a1 s` and `eval_atom a2 s` with carry `eval_atom y s`. The `adcB` function returns a tuple `(c, s)` where *c* is the carry and *s* is the sum. The semantics for `Iadcs c v a1 a2` is similar. The semantics for various subtraction instructions use the bit-vector functions `subB` and `sbbB` as well.

For `Imul v a1 a2`, the function `mulB` computes the half-product of `eval_atom a1 s` and `eval_atom a2 s`. For unsigned full multiplication `Imull vh vl a1 a2`, `eval_atom a1 s` and `eval_atom a2 s` are extended by zeros `zext`. The high bits of the extended product are computed by the *coq-nbits* function `high` and stored in `vh`. The low bits are in `vl` are computed by `low` and stored in `vl`. The signed full-multiplication uses the sign-extension function `sext` instead. `Imulj v a1 a2` updates `v` with the full product.

The instruction `Ishl v a i` uses the bit-vector function `shlB` to shift `eval_atom a s` to the left by *i* bits and stores the shifted result in `v`. `Icshl vh vl a1 a2 i` concatenates `eval_atom a1 s` and `eval_atom a2 s` and shifts the concatenation to the left by *i* bits. The variable `vh` is updated with the high bits of the shifted concatenation. The low bits of the shifted concatenation is shifted to the right by *i* bits and stored in `vl`.

The `Inondet v ty` updates the variable `v` with the bit-vector `n` of the same size as the type `ty`. `Ijoin v ah al` updates `v` with the concatenation of `eval_atom ah s` and `eval_atom al s`. The unsigned `Isplit vh vl a n` instruction shifts `eval_atom a s` to the right by `n` bits (`shrB`) and stores the shifted result in `vh`. The variable `vl` is updated with the low `n` bits of `eval_atom a s`. The signed `Isplit vh vl a n` uses the arithmetic right-shift function `sarB` to compute the value of `vh` instead. The bitwise logical instructions `Inot`, `Iand`, `Ior`, and `Ixor` use the *coq-nbits* functions `invB`, `andB`, `orB`, and `xorB` respectively. Both `Icast v ty a` and `vpc v ty a` use the auxiliary `tcast` function. `eval_instsr (Iassume e) s s` holds if `eval_bexp e te s` is true.

For the semantics of algebraic predicates, `bv2z t bits` converts the bit-vector `bits` to `Z` by the type `t` and `acc2z te v s` returns the integer value of the variable `v` in the store `s` under the type environment `te`. The semantics of algebraic expressions is defined by corresponding integer functions in Coq (`eval_eexp`). For algebraic predicates, `Etrue` evaluates to `True`. `Eeq e1 e2` checks if the algebraic expressions `e1` and `e2` evaluate to the same integer. `Eeqmod e1 e2 p` checks if the difference of `eval_eexp e1 te s` and `eval_eexp e2 te s` is divided by `eval_eexp p te s`.

The semantics of range expressions use the corresponding *coq-nbits* functions in `eval_rexp`. For arithmetic range expressions, the bit-vector functions `negB`, `addB`, `subB`, `mulB`, `uremB`, `sremB`, and `smodB` are used for `Rnegb`, `Radd`, `Rsub`, `Rmul`, `Rumod`, `Rsrem`, and `Rsmod` respectively. For bitwise range expressions, `invB`, `andB`, `orB`, and `xorB` are used for `Rnotb`, `Randb`, `Rorb`, `Rxorb` respectively. Range predicates also use the corresponding predicates (`eval_rbexp`). Finally, a predicate evaluates to true if both of its algebraic and range predicates evaluate to true (`eval_bexp`).

```
Definition eval_atom (a : atom) (s : S.t) : bits :=
  match a with | Avar v => S.acc v s | Aconst _ n => n end.
Inductive eval_instr (te : TE.env)
          : instr → state → state → Prop :=
| EImov v a s t : S.Upd v (eval_atom a s) s t
    → eval_instr te (Imov v a) s t
| EIcmovT v c a1 a2 s t : to_bool (eval_atom c s) →
    S.Upd v (eval_atom a1 s) s t
    → eval_instr te (Icmov v c a1 a2) s t
| EIcmovF v c a1 a2 s t : ¬ to_bool(eval_atom c s) →
    S.Upd v (eval_atom a2 s) s t
    → eval_instr te (Icmov v c a1 a2) s t
| EInop s : eval_instr te Inop s s
| EIadd v a1 a2 s t :
    S.Upd v (addB (eval_atom a1 s)(eval_atom a2 s)) s t
    → eval_instr te (Iadd v a1 a2) s t
| EIadds c v a1 a2 s t :
    S.Upd2 v (addB (eval_atom a1 s) (eval_atom a2 s))
            c (1-bits of bool
                   (carry_addB (eval_atom a1 s)
                                 (eval_atom a2 s))) s t
    → eval_instr te (Iadds c v a1 a2) s t
| EIadc v a1 a2 y s t :
    S.Upd v (adcB (to_bool (eval_atom y s))
                    (eval_atom a1 s)
                    (eval_atom a2 s)).2 s t
    → eval_instr te (Iadc v a1 a2 y) s t
| EIadcs c v a1 a2 y s t :
    S.Upd2 v (adcB (to_bool (eval_atom y s))
                     (eval_atom a1 s)
                     (eval_atom a2 s)).2
            c (1-bits of bool
                   ((adcB (to_bool (eval_atom y s))
                           (eval_atom a1 s)
                           (eval_atom a2 s)).1)) s t
    → eval_instr te (Iadcs c v a1 a2 y) s t
| EIsub v a1 a2 s t :
    S.Upd v (subB (eval_atom a1 s)(eval_atom a2 s)) s t
    → eval_instr te (Isub v a1 a2) s t
| EIsubc c v a1 a2 s t :
    S.Upd2 v ((adcB true (eval_atom a1 s)
                      (invB (eval_atom a2 s))).2)
            c (1-bits of bool
                   ((adcB true (eval_atom a1 s)
                           (invB (eval_atom a2 s))).1)) s t
    → eval_instr te (Isubc c v a1 a2) s t
| EIsubb b v a1 a2 s t :
    S.Upd2 v (subB (eval_atom a1 s) (eval_atom a2 s))
           b (1-bits of bool
                  (borrow_subB (eval_atom a1 s)
                                 (eval_atom a2 s))) s t
    → eval_instr te (Isubb b v a1 a2) s t
| EIsbc v a1 a2 y s t :
    S.Upd v (adcB (to_bool (eval_atom y s))
                    (eval_atom a1 s)
                    (invB (eval_atom a2 s))).2 s t
    → eval_instr te (Isbc v a1 a2 y) s t
| EIsbcs c v a1 a2 y s t :
    S.Upd2 v (adcB (to_bool (eval_atom y s))
                     (eval_atom a1 s)
                     (invB (eval_atom a2 s))).2
            c (1-bits of bool
                   ((adcB (to_bool (eval_atom y s))
                           (eval_atom a1 s)
                           (invB (eval_atom a2 s))).1)) s t
    → eval_instr te (Isbcs c v a1 a2 y) s t
```

```
| EIsbb v a1 a2 y s t :
    S.Upd v (sbbB (to_bool (eval_atom y s))
                    (eval_atom a1 s)
                    (eval_atom a2 s)).2 s t
    → eval_instr te (Isbb v a1 a2 y) s t
| EIsbbs b v a1 a2 y s t :
    S.Upd2 v (sbbB (to_bool (eval_atom y s))
                (eval_atom a1 s) (eval_atom a2 s)).2
           b (1-bits of bool
                  ((sbbB (to_bool (eval_atom y s))
                          (eval_atom a1 s)
                          (eval_atom a2 s)).1)) s t
    → eval_instr te (Isbbs b v a1 a2 y) s t
| EImul v a1 a2 s t :
    S.Upd v (mulB (eval_atom a1 s)(eval_atom a2 s)) s t
    → eval_instr te (Imul v a1 a2) s t
| EImullU vh vl a1 a2 s t : is_unsigned (atyp a1 te) →
    S.Upd2 vl (low (size (eval_atom a2 s))
                  (mulB (zext (size (eval_atom a1 s))
                               (eval_atom a1 s))
                         (zext (size (eval_atom a1 s))
                               (eval_atom a2 s))))
           vh (high (size (eval_atom a1 s))
                  (mulB (zext (size (eval_atom a1 s))
                               (eval_atom a1 s))
                         (zext (size (eval_atom a1 s))
                               (eval_atom a2 s)))) s t
    → eval_instr te (Imull vh vl a1 a2) s t
| EImullS vh vl a1 a2 s t : is_signed (atyp a1 te) →
    S.Upd2 vl (low (size (eval_atom a2 s))
                  (mulB (sext (size (eval_atom a1 s))
                               (eval_atom a1 s))
                         (sext (size (eval_atom a1 s))
                               (eval_atom a2 s))))
           vh (high (size (eval_atom a1 s))
                  (mulB (sext (size (eval_atom a1 s))
                               (eval_atom a1 s))
                         (sext (size (eval_atom a1 s))
                               (eval_atom a2 s)))) s t
    → eval_instr te (Imull vh vl a1 a2) s t
| EImuljU v a1 a2 s t : is_unsigned (atyp a1 te) →
    S.Upd v (mulB (zext (size (eval_atom a1 s))
                        (eval_atom a1 s))
                  (zext (size(eval_atom a1 s))
                        (eval_atom a2 s))) s t
    → eval_instr te (Imulj v a1 a2) s t
| EImuljS v a1 a2 s t : is_signed (atyp a1 te) →
    S.Upd v (mulB (sext (size (eval_atom a1 s))
                        (eval_atom a1 s))
                  (sext (size(eval_atom a1 s))
                        (eval_atom a2 s))) s t
    → eval_instr te (Imulj v a1 a2) s t
| EIshl v a i s t : S.Upd v (shlB i (eval_atom a s)) s t
    → eval_instr te (Ishl v a i) s t
| EIcshl vh vl a1 a2 i s t :
    S.Upd2 vl (shrB i (low (size (eval_atom a2 s))
                          (shlB i (cat (eval_atom a2 s)
                                       (eval_atom a1 s)))))
           vh (high (size (eval_atom a1 s))
                    (shlB i (cat (eval_atom a2 s)
                                 (eval_atom a1 s)))) s t
    → eval_instr te (Icshl vh vl a1 a2 i) s t
| EInondet v ty s t n : size n = sizeof_typ ty →
    S.Upd v n s t → eval_instr te (Inondet v ty) s t
| EInot v ty a s t : S.Upd v (invB (eval_atom a s)) s t
    → eval_instr te (Inot v ty a) s t
```

Figure 6: Semantics of CRYPTOLINE Instructions and Predicates

```
| EIjoin v ah al s t :
    S.Upd v (cat (eval_atom al s) (eval_atom ah s)) s t
    → eval_instr te (Ijoin v ah al) s t
| EIsplitU vh vl a n s t : is_unsigned (atyp a te) →
    S.Upd2 vl (shrB ((size (eval_atom a s)) - n)
                    (shlB ((size (eval_atom a s)) - n)
                          (eval_atom a s)))
           vh (shrB n (eval_atom a s)) s t
    → eval_instr te (Isplit vh vl a n) s t
| EIsplitS vh vl a n s t : is_signed (atyp a te) →
    S.Upd2 vl (shrB ((size (eval_atom a s)) - n)
                    (shlB ((size (eval_atom a s)) - n)
                          (eval_atom a s)))
           vh (sarB n (eval_atom a s)) s t
    → eval_instr te (Isplit vh vl a n) s t
| EIand v ty a1 a2 s t :
    S.Upd v (andB (eval_atom a1 s)(eval_atom a2 s)) s t
    → eval_instr te (Iand v ty a1 a2) s t
| EIor v ty a1 a2 s t :
    S.Upd v (orB (eval_atom a1 s) (eval_atom a2 s)) s t
    → eval_instr te (Ior v ty a1 a2) s t
| EIxor v ty a1 a2 s t :
    S.Upd v (xorB (eval_atom a1 s)(eval_atom a2 s)) s t
    → eval_instr te (Ixor v ty a1 a2) s t
| EIcast v ty a s t :
    S.Upd v (tcast (eval_atom a s) (atyp a te) ty) s t
    → eval_instr te (Icast v ty a) s t
| EIvpc v ty a s t :
    S.Upd v (tcast (eval_atom a s) (atyp a te) ty) s t
    → eval_instr te (Ivpc v ty a) s t
| EIassume e s : eval_bexp e te s
    → eval_instr te (Iassume e) s s.
```

Figure 7: Semantics of CRYPTOLINE Instructions (continued)

```
Definition bv2z (t : typ) (bs : bits) : Z :=
  match t with | Tuint _ => to_Zpos bs
               | Tsint _ => to_Z bs end.
Definition acc2z (E : TE.env) (v : V.t) (s : S.t) : Z :=
  bv2z (TE.vtyp v E) (S.acc v s).
Definition eval_eunop (op : eunop) (v : Z) : Z :=
  match op with | Eneg => - v end.
Definition eval_ebinop (op : ebinop) (v1 v2 : Z) : Z :=
  match op with | Eadd => v1 + v2 | Esub => v1 - v2
                | Emul => v1 * v2 end.
Fixpoint eval_eexp (e : eexp)(te : TE.env)(s : S.t) : Z :=
  match e with
  | Evar v => acc2z te v s | Econst n => n
  | Eunop op e => eval_eunop op (eval_eexp e te s)
  | Ebinop op e1 e2 =>
    eval_ebinop op (eval_eexp e1 te s) (eval_eexp e2 te s)
  end.
Definition modulo (a b p : Z) := ∃ k : Z, a - b = k * p.
Fixpoint eval_ebexp (e : ebexp) (te : TE.env) (s : S.t)
         : Prop :=
  match e with
  | Etrue => True
  | Eeq e1 e2 => eval_eexp e1 te s = eval_eexp e2 te s
  | Eeqmod e1 e2 p =>
    modulo (eval_eexp e1 te s) (eval_eexp e2 te s)
           (eval_eexp p te s)
  | Eand e1 e2 => eval_ebexp e1 te s ∧ eval_ebexp e2 te s
  end.
Definition eval_runop (op : runop) (v : bits) : bits :=
  match op with | Rnegb => negB v | Rnotb => invB v end.
Definition eval_rbinop (op : rbinop) (v1 v2 : bits):bits :=
  match op with
  | Radd => addB v1 v2 | Rsub => subB v1 v2
  | Rmul => mulB v1 v2 | Rumod => uremB v1 v2
  | Rsrem => sremB v1 v2 | Rsmod => smodB v1 v2
  | Randb => andB v1 v2 | Rorb => orB v1 v2
  | Rxorb => xorB v1 v2
  end.
Definition eval_rcmpop (op : rcmpop) (v1 v2 : bits):bool :=
  match op with
  | Rult => ltB v1 v2 | Rule => leB v1 v2
  | Rugt => gtB v1 v2 | Ruge => geB v1 v2
  | Rslt => sltB v1 v2 | Rsle => sleB v1 v2
  | Rsgt => sgtB v1 v2 | Rsge => sgeB v1 v2
  end.
Fixpoint eval_rexp (e : rexp) (s : S.t) : bits :=
  match e with
  | Rvar v => S.acc v s | Rconst w n => n
  | Runop _ op e => eval_runop op (eval_rexp e s)
  | Rbinop _ op e1 e2 =>
    eval_rbinop op (eval_rexp e1 s) (eval_rexp e2 s)
  | Ruext _ e i => zext i (eval_rexp e s)
  | Rsext _ e i => sext i (eval_rexp e s)
  end.
Fixpoint eval_rbexp (e : rbexp) (s : S.t) : bool :=
  match e with
  | Rtrue => true
  | Req _ e1 e2 => eval_rexp e1 s == eval_rexp e2 s
  | Rcmp _ op e1 e2 =>
    eval_rcmpop op (eval_rexp e1 s) (eval_rexp e2 s)
  | Rneg e => ¬ (eval_rbexp e s)
  | Rand e1 e2 => (eval_rbexp e1 s) && (eval_rbexp e2 s)
  | Ror e1 e2 => (eval_rbexp e1 s) || (eval_rbexp e2 s)
  end.
Definition eval_bexp (e : bexp) (te : TE.env) (s : S.t)
           : Prop :=
  eval_ebexp (eqn_bexp e) te s ∧ eval_rbexp (rng_bexp e) s.
```

Figure 8: Semantics of CRYPTOLINE Predicates