Automatic Certified Verification of Cryptographic Programs with CoQCRYPTO LINE

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

CoQCRYPTO LINE is an automatic certified verification tool for cryptographic programs. It is built on OCAML programs extracted from algorithms fully certified in Coq with SS-REFLECT. Similar to other automatic tools, CoQCRYPTO LINE calls external decision procedures during verification. To ensure correctness, all answers from external decision procedures are validated by certified certificate checkers in CoQCRYPTO LINE. We evaluate CoQCRYPTO LINE 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.

CoQCRYPTO LINE is an automatic verification tool for cryptographic programs with pretty good assurance. Like other automatic techniques, CoQCRYPTO LINE reduces verification tasks to various computational problems solved by external decision procedures. Unlike other techniques, proof assistants are used to certify CoQCRYPTO LINE to attain higher assurance. Instead of cryptographic programs, we use proof assistants to certify the correctness of the CoQCRYPTO LINE 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, CoQCRYPTO LINE automatically performs verification tasks with better assurance than other automatic tools.

More precisely, we formalize semantics for the typed CRYPTO LINE 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. CoQCRYPTO LINE is built on OCAML programs extracted from the certified Coq algorithm. Overall, CoQCRYPTO LINE contains ≈ 68k lines of OCAML programs extracted from ≈ 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 COQCRYPTO LINE. 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 COQCRYPTO LINE. 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 COQCRYPTO LINE 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 CRYPTOVERIFY [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]). COQCRYPTO LINE 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. CRYPTO LINE [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. CRYPTO LINE 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 BVCRYPTO LINE [36], which is the first automatic and partly certified verification tool for cryptographic programs. COQCRYPTO LINE nonetheless possesses three essential advantages: (i) BVCRYPTO LINE only supports the unsigned integer representation but COQCRYPTO LINE supports both signed and unsigned representations; (ii) the SMT-based technique in BVCRYPTO LINE is not certified whereas COQCRYPTO LINE certifies both algebraic and SMT-based techniques; (iii) COQCRYPTO LINE is a standalone tool built on extracted OCAML programs but BVCRYPTO LINE 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 COQCRYPTO LINE are briefed in Section 3. An overview of COQCRYPTO LINE is given in Section 4. It is followed by our formal semantics of the typed CRYPTO LINE 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*B; 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.

COQFBV. 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. COQFBV 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. COQFBV 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 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_lte bexp exp0 exp1` evaluates to `exp0` if the `QF_BV` predicate `bexp` is true and `exp1` 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, $\mathbb{X}$ a set of variables and $\mathbb{Z}[\mathbb{X}]$ the set of multivariable polynomials in $\mathbb{X}$ with integral coefficients. Let $f_0, f_1, f_2 \in \mathbb{Z}[\mathbb{X}]$. $f_0(\mathbb{X}) = f_1(\mathbb{X})$ is a polynomial equation; $f_0(\mathbb{X}) \equiv f_1(\mathbb{X}) \bmod f_2(\mathbb{X})$ is a polynomial modular equation. A (modular) equation is a polynomial equation or a polynomial modular equation. A root of a polynomial equation $f_0(\mathbb{X}) = f_1(\mathbb{X})$ is a sequence $\mathbb{Z}$ of integers such that $f_0(\mathbb{Z}) - f_1(\mathbb{Z}) = 0$. A root of a polynomial modular equation $f_0(\mathbb{X}) + f_1(\mathbb{X}) \bmod f_2(\mathbb{X})$ is a sequence $\mathbb{Z}$ of integers such that $f_2(\mathbb{Z})$ divides $f_0(\mathbb{Z}) - f_1(\mathbb{Z})$. A system of (modular) equations is a set of (modular) equations. A root of a system of (modular) equations is a sequence $\mathbb{Z}$ of integers such that $\mathbb{Z}$ is a root of every (modular) equation in the system. Given two systems $\Pi$ and $\Pi'$ of (modular) equations, $\Pi$ entails $\Pi'$ if all roots of $\Pi$ are also roots of $\Pi'$. The root entailment problem is to decide whether $\Pi$ entails $\Pi'$.

## 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 $d_0 + d_1 \times 2^n + ... + d_m \times 2^{n(m-1)}$. The 256-bit integer represented by $m$'s is the prime $p_{256} = 2^{256} - 2^{224} - 2^{192} + 2^9 + 1$ 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 $p_{256}$ 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
  CNF #1: [CERTIFIED] 0.522745 seconds
  CNF #2: [UNSAT] 0.850775 seconds
Results of checking CNF formulas: [OK] 144.669136 seconds
Finding polynomial coefficients
Finished finding polynomial coefficients 0.000012 seconds
Verification result: [OK] 135.297161 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 $p_{256}$ for the curve. We have a similar pre-condition as for `ecp_nistz256_add`.

```coq
and [ m0=0xfffffffffffffffff, m1=0x8000000000000000, m2=0xfffffffffffffffff, m3=0x8000000000000000 ]
and [ m0=0xfffffffffffffffff@64, m1=0x8000000000000000, m3=0x8000000000000000, n0=0x8000000000000000 ]
```

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 then the prime $p_{256}$. Formally, we have the following post-condition:
prove that the output integer is in the proper range.

\[
\text{limbs } 64 \{ \text{c0, c1, c2, c3} \} < \text{limbs } 64 \{ \text{m0, m1, m2, m3} \}
\]

\[
\text{limbs } 64 \{ \text{b0, b1, b2, b3} \} < \text{limbs } 64 \{ \text{m0, m1, m2, m3} \}
\]

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:

\[
\text{assert true } && \text{and } [ \text{carry}=0@1, \text{overflow}=0@1 ];
\]

\[
\text{assume and } [ \text{carry}=0@1, \text{overflow}=0 ]; \text{true};
\]

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 CRYPTOline 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, manually 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. 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.
Full text of the document is provided above. The image contains a flowchart illustrating the process of transforming specifications into SMT problems, with modules and functions labeled CoqCryptoLine parser, SSA2QFBV, SSA2ZSSA, SSA, and Validator. The diagram shows the flow from CoqCryptoLine parser to SSA2QFBV, SSA2ZSSA, SSA, and finally to the SMT solver, with validation steps indicated.

The text describes the process of transforming specifications to the SMT QF_BV problem for cryptographic assembly programs. It details the role of CoqCryptoLine in this process, including the transformation of specifications, the generation of SMT QF_BV queries, and the verification of post-conditions.

The diagram is labeled as Figure 2: Overview of CoqCryptoLine.
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 \( a_0 + a_1z + a_2z^2 + \ldots + a_9z^9 \) (or equivalently limbs 64 \( [a_0, a_1, a_2, \ldots, a_9] \)) mentioned in Section 3. The type of algebraic expressions is \( \text{eexp} \). An algebraic expression is inductively defined to be a variable \( \text{Evar} v \), an integer constant \( \text{Econst} n \), a unary algebraic expression \( \text{Euop} e \), or a binary algebraic expression \( \text{Ebop} e1 e2 \) where \( v \) is a variable, \( n \) is an integer of \( \text{COQ} \)'s type \( \text{Z} \), \( e \) is a unary algebraic operator, \( e1 \) and \( e2 \) are algebraic expressions. The unary algebraic operator \( \text{Eneg} \) (negation), and the binary algebraic operators \( \text{Eadd} \) (addition), \( \text{Esub} \) (subtraction), and \( \text{Emul} \) (multiplication) are supported.

An algebraic expression \( e \) is evaluated in a store \( s \) under a type environment \( \text{te} \) to an integer \( \text{eval\_eexp} e \text{ te } s \). 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 \( \text{bv2z} \) which converts a bit-vector to an integer by the \( \text{coq\text{-}nbits} \) 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

\[
\text{bv2z} \; (t \; : \; \text{typ}) \; (bs \; : \; \text{bits}) \; : \; \text{Z} \; := \\
\begin{cases} 
\text{match } t \text{ with } \\
| \text{Uint } _{\text{unsigned}} => & \text{to\_zpos} \; bs \\
| \text{Tsint } _{\text{two's complement}} => & \text{to\_z} \; bs \\
\end{cases}
\]

Algebraic operators \( \text{Eneg} \), \( \text{Eadd} \), \( \text{Esub} \), and \( \text{Emul} \) 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 \( \text{QF\_BV} \) expressions in \( \text{CoQ\text{-}QFBV} \). More specifically, a range expression is a variable \( \text{Vvar} v \), a bit-vector constant \( \text{Rconst} w \), a unary range expression \( \text{Ruop} w \), a binary range expression \( \text{Rbinop} w \), a zero extension \( \text{Rext} w i \), or a signed extension \( \text{RsExt} w i \) 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 \( e1 \) and \( e2 \) are range expressions. Two unary range operators \( \text{Rneg} \) (negation) and \( \text{Rnrb} \) (binary inversion) are supported. The supported binary range operators include \( \text{Radd} \) (addition), \( \text{Rsub} \) (subtraction), \( \text{Rmul} \) (multiplication), \( \text{Rmod} \) (unsigned remainder), \( \text{Rsrem} \) (signed remainder with sign follows dividend), \( \text{Rsmod} \) (signed remainder with sign follows divisor), bitwise AND (\( \text{Rand} \)), bitwise OR (\( \text{Rorb} \)), and bitwise XOR (\( \text{Rxorb} \)). The type of range expressions is \( \text{rexp} \).

A range expression \( e \) is evaluated in a store \( s \) to a bit-vector \( \text{eval\_rexp} e \text{ s} \). The definition of \( \text{eval\_rexp} \) follows the semantics of \( \text{QF\_BV} \) expressions defined in CoQ\text{-}QFBV. For example, \( \text{Radd} \) is evaluated in the same way as \( \text{qfbv\text{-}add} \) in CoQ\text{-}QFBV. 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 \( \text{ebexp} \). An algebraic predicate is inductively defined to be an atomic algebraic predicate or a conjunction (\( \text{Eand} \)) of algebraic predicates. An atomic algebraic predicate is \( \text{Etrue} \) or a (modular) equation over algebraic expressions of type \( \text{eexp} \). Given algebraic expressions \( e1 \), \( e2 \), and \( m \), \( \text{Eq} e1 e2 \) is the equality of \( e1 \) and \( e2 \) while \( \text{Eqmod} e1 e2 m \) is the congruence of \( e1 \) and \( e2 \) modulo \( m \). For example, the congruence \( X \equiv 1 \pmod{2} \) can be defined as the algebraic predicate \( \text{Eqmod} \; (\text{Evar} X) \; (\text{Econst} 1) \; (\text{Econst} 2) \) assuming that \( X \) is a variable.

Given a store \( s \) and a type environment \( \text{te} \), the semantics of an algebraic predicate \( e \) in \( s \) under \( \text{te} \) is defined as the proposition \( \text{eval\_ebexp} e \text{ te } s \) which holds if and only if all atomic algebraic predicates in \( e \) hold in \( s \) under \( \text{te} \). The atomic algebraic predicate \( \text{Etrue} \) always holds. \( \text{Eq} e1 e2 \) holds if \( \text{eval\_ebexp} e1 \text{ te } s = \text{eval\_ebexp} e2 \text{ te } s \) where \( = \) is the equality in CoQ. \( \text{Eqmod} e1 e2 m \) holds if modulo \( \text{eval\_ebexp} e1 \text{ te } s \) \( \text{eval\_ebexp} e2 \text{ te } s \) \( \text{eval\_ebexp} m \text{ te } s \). 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 \( \text{QF\_BV} \) predicates in CoQ\text{-}QFBV. More specifically, a range predicate is an atomic range predicate or an arbitrary Boolean expression (\( \text{Rneg} \) for Boolean \( \text{NOT} \), \( \text{Rand} \) for Boolean \( \text{AND} \), \( \text{Rorb} \) for Boolean \( \text{OR} \), and \( \text{Rxorb} \) for Boolean \( \text{XOR} \)).
for Boolean AND, and \( \text{Ror} \) for Boolean OR) over range predicates. An atomic range predicate is \( \text{Rtrue} \), an equality \( \text{Req} w \) \( e1 e2 \), or a comparison \( \text{Rcmp} w \text{rcop} e1 e2 \) where \( \text{rcop} \) is a comparison operator, \( e1 \) and \( e2 \) are range expressions, and \( w \) is the width of the arguments. A comparison operator can be \( \text{Rult} \) (unsigned less-than) or \( \text{Rsll} \) (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 CRYPTO-LINE text, is represented as the range predicate \( \text{Rcmp} w \text{Rult} (\text{Rvar} X) (\text{Rvar} Y) \), assuming that both \( X \) and \( Y \) have width \( w \). The type of range predicates is \( \text{rbexp} \).

A range predicate \( e \) is evaluated in a store \( s \) to a Boolean \( \text{eval_rbexp} e s \). The definition of \( \text{eval_rbexp} \) follows the semantics of QF_BV predicates defined in CoQFQBV. For example, \( \text{Rult} \) is evaluated in the same way as \( \text{qfbv_ult} \) in CoQFQBV.

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

### 5.3 Instructions and Programs

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

An instruction of type \( \text{instr} \) assigns destination variables with values of source atoms.

**Inductive instr : Type :=**

- \( \text{Imov} : \text{var} \to \text{atom} \to \text{instr} \)
  - \( \text{Iadd} : \ldots \) \( \text{Iadds} : \ldots \) \( \text{Iaddc} : \ldots \) \( \text{Iadc} : \ldots \)
  - \( \text{Isub} : \ldots \) \( \text{Isubb} : \ldots \) \( \text{Isbb} : \ldots \) \( \text{Isbbs} : \ldots \)
  - \( \text{Iul} : \ldots \) \( \text{Imull} : \ldots \) \( \text{Imulj} : \ldots \)
  - \( \text{Iahl} : \ldots \) \( \text{Icshl} : \ldots \) \( \text{Ijoin} : \ldots \) \( \text{Isplit} : \ldots \)
  - \( \text{Inot} : \ldots \) \( \text{Iand} : \ldots \) \( \text{Ior} : \ldots \) \( \text{IXor} : \ldots \)
  - \( \text{Icmov} : \ldots \) \( \text{Inondet} : \ldots \) \( \text{Icast} : \ldots \)
  - \( \ldots \)
  - \( \text{Iassum} : \text{bexp} \to \text{instr} \).

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

Executions of CRYPTO-LINE programs are formalized by relational semantics. Informally, our semantics of CRYPTO-LINE 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 \( \text{eval_instr} \) \( i \to s \to t \) denotes that the successor store \( t \) can be reached by executing \( i \) at \( s \). For example, \( \text{eval_instr} \) \( (\text{Imov} v a) s \to t \) holds if \( s.\text{Upd} v (\text{eval_atom} a) s \to t \) holds, that is, \( t \) is updated from \( s \) by mapping \( v \) to the value of \( a \) in \( s \). For the addition instruction \( \text{Iadd} \), \( \text{eval_instr} \) \( (\text{Iadd} v a1 a2) s \to t \) holds if \( s.\text{Upd} v (\text{addB} (\text{eval_atom} a1) s) (\text{eval_atom} a2) s \to t \) 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 \( \text{Icast} \) 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 \( \text{Icast} v \) \( ty \) \( a \to s \) assigns \( bs \) represented in \( ty \) to \( v \). Let \( \text{size} bs \) be the width of \( bs \) and \( w \) be the width of \( ty \). Depending on the relation between \( \text{size} bs \) 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 \( \text{ucastB} bs w \).

**Definition ucastB (bs : bits) (w : nat) : bits :=**

- \( \text{if} w < \text{size} bs \to \text{bs} \)
- \( \text{else if} \ w < \text{size} bs \to \text{low} w bs \)
- \( \text{else zext} (w - \text{size} bs) bs \)

where \( \text{low} w bs \) is the lower \( w \) bits of \( bs \). Otherwise the casted value is \( \text{scastB} bs w \) where \( \text{scastB} \) is defined same as \( \text{ucastB} \) except that \( \text{zext} \) is used for extension instead of \( \text{zext} \). See Appendix B for more details of the semantics of the instructions inTyped CRYPTO-LINE.

Given a type environment \( te \), a program \( p \), and stores \( s \) and \( t \), the proposition \( \text{eval_program} \text{te} p s t \) 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., \( \text{eval_program} te [:] s \to s \) holds. If \( p \) is an instruction \( i \) followed by a program \( p' \), denoted by \( i: p' \), \( \text{eval_program} te (i: p') s \to t \) holds if there is some store \( u \) such that both \( \text{eval_instr} te i s u \) and \( \text{eval_program} te' p' u \to t \) hold where \( te' \) is \( \text{instr_succ_typenv} i \text{te} \). The type environment updated from \( te \) after executing the instruction \( i \) is formalized as the term \( \text{instr_succ_typenv} i \text{te} \). Similarly, type environment updated from \( te \) after executing the program \( p \) is formalized as the term \( \text{program_succ_typenv} \).
Compared to BVCRYPTOLINE, COQCryptol ine offers several new instructions such as Inondet, Icmov, Imul, Iand, lor, lxor, Imulj, Icast, Ijoin, and Iassume. Bitwise operations Iand and lor are often used for bit masking, which is commonly used in security libraries. The Iassume instruction allows interchangeability between algebraic specifications 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 = 0⃗1 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 $\mathsf{s}$ is formalized as a COQ record $\mathsf{spec}$ with four fields, of which $\mathsf{sinputs}$ $\mathsf{s}$ is the initial type environment, $\mathsf{spre}$ $\mathsf{s}$ is the pre-condition, $\mathsf{sprog}$ $\mathsf{s}$ is the program, and $\mathsf{spost}$ $\mathsf{s}$ is the post-condition. Both the pre-condition and the post-condition are predicates.

Record $\mathsf{spec} : \text{Type} :=$
{ $\mathsf{sinputs} : \text{env}$; $\mathsf{spre} : \text{bexp}$;
 $\mathsf{sprog} : \text{program}$; $\mathsf{spost} : \text{bexp}$. }.

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

Record $\mathsf{espec} :=$
{ $\mathsf{esinputs} : \text{env}$; $\mathsf{espre} : \text{bexp}$;
 $\mathsf{esprog} : \text{program}$; $\mathsf{espost} : \text{bexp}$. }.

Record $\mathsf{rspec} :=$
{ $\mathsf{rsinputs} : \text{env}$; $\mathsf{rspre} : \text{rbexp}$;
 $\mathsf{rsprog} : \text{program}$; $\mathsf{rspost} : \text{rbexp}$. }

The functions $\mathsf{espec}\_\text{of}\_\text{spec}$ and $\mathsf{rspec}\_\text{of}\_\text{spec}$ convert a specification to an algebraic specification of type $\mathsf{espec}$ and a range specification of type $\mathsf{rspec}$ respectively by dropping either algebraic predicates or range predicates in the pre- and post-conditions.

A store $\mathsf{s}$ is conformed to a type environment $\mathsf{te}$, defined as $\mathsf{S}\_\text{conform}\ \mathsf{s}\ \mathsf{te}$ in Coq, if and only if for every variable $\mathsf{v}$ in $\mathsf{te}$, the type of $\mathsf{v}$ in $\mathsf{te}$ and the bit-vector value of $\mathsf{v}$ in $\mathsf{s}$ have the same width. The validity of a specification $\mathsf{s}$, defined as the proposition $\mathsf{valid}\_\mathsf{spec}\ \mathsf{s}$, holds if and only if the execution of $\mathsf{sprog}$ $\mathsf{s}$ from any store $\mathsf{s1}$ conformed to $\mathsf{sinputs}$ $\mathsf{s}$ and satisfying $\mathsf{spre}$ $\mathsf{s}$ terminates in a store $\mathsf{s2}$ where $\mathsf{spost}$ $\mathsf{s}$ holds.

The validity of an algebraic specification and the validity of a range specification are defined similarly as $\mathsf{valid}\_\mathsf{espec}$ and $\mathsf{valid}\_\mathsf{rspec}$ respectively. We have the lemma $\mathsf{valid}\_\mathsf{spec}\_\mathsf{split}$ for splitting the validity of a specification into the validity of its algebraic part and the validity of its range part.

Lemma $\mathsf{valid}\_\mathsf{spec}\_\mathsf{split}$ ($\mathsf{s} : \text{spec}$) :
$\mathsf{valid}\_\mathsf{espec} (\mathsf{espec}\_\text{of}\_\text{spec} \mathsf{s}) \rightarrow$
$\mathsf{valid}\_\mathsf{rspec} (\mathsf{rspec}\_\text{of}\_\text{spec} \mathsf{s}) \rightarrow \mathsf{valid}\_\mathsf{spec} \mathsf{s}$.

6 Certified Verification

Given a typed CRYPTOLINE specification text, the COQCryptol ine parser translates the text into a term of type $\mathsf{spec}$, or more specifically $\mathsf{DSL}\_\mathsf{spec}$ ($\mathsf{spec}$ in the DSL module). A specification of type $\mathsf{DSL}\_\mathsf{spec}$ is verified by the function $\mathsf{verify}\_\mathsf{dsl}$:

Definition $\mathsf{verify}\_\mathsf{dsl}$ ($\mathsf{o} : \text{options}$) ($\mathsf{s} : \mathsf{DSL}\_\mathsf{spec}$) ::= $
\mathsf{verify}\_\mathsf{ssa} \mathsf{o} (\mathsf{SSA}\_\mathsf{ssa}\_\mathsf{spec} \mathsf{s})$.

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

A specification in SSA is verified by the function $\mathsf{verify}\_\mathsf{ssa}$ where the algebraic reduction (to the root entailment problem) and the bit-vector reduction (to the SMT QF_BV problem) are applied.

Definition $\mathsf{verify}\_\mathsf{ssa}$ ($\mathsf{o} : \text{options}$) ($\mathsf{s} : \mathsf{SSA}\_\mathsf{spec}$) ::= $
\mathsf{verify}\_\mathsf{rspec}\_\mathsf{algsnd} \mathsf{o} \mathsf{s}$ ($\mathsf{verify}\_\mathsf{espec} \mathsf{o} \mathsf{s}$).

The algebraic reduction and the solving of root entailment problems are performed in the function $\mathsf{verify}\_\mathsf{espec}$. The bit-vector reduction and the solving of SMT QF_BV queries together with the soundness conditions are performed in the function $\mathsf{verify}\_\mathsf{rspec}\_\mathsf{algsnd}$. We detail $\mathsf{verify}\_\mathsf{espec}$ and $\mathsf{verify}\_\mathsf{rspec}\_\mathsf{algsnd}$ 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 $\mathsf{verify}\_\mathsf{espec}$ applies the algebraic reduction to a specification in SSA through $\mathsf{algred}\_\mathsf{espec}$ and then solves the resulting root entailment problems through $\mathsf{verify}\_\mathsf{rep}$ or its parallel version $\mathsf{verify}\_\mathsf{rep}\_\mathsf{list}$ both with answers verified by a validator.
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 :=
\forall s, eval_ebexp f s \rightarrow eval_ebexp 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_ebexp and eval_zbexp over integer stores, which are mappings from variables to integer values. We formalize integer stores as the type \(\mathbb{Z}\). The evaluation functions eval_ebexp 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 \texttt{algred}\_espec translates an algebraic specification \texttt{spec} to a root entailment problem.

Definition algred\_espec \texttt{spec} :=
let \texttt{spec} :=
algred\_program \texttt{spec} \texttt{init}\_g \texttt{esprog} \texttt{spec} \texttt{in}
| premise := eand \texttt{eqn}\_bexp \texttt{spec} \texttt{espre} \texttt{spec} \texttt{eands eprogs};
| conseq := espost \texttt{spec} |.

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

The function \texttt{algred}\_program reduces a program instruction by instruction through \texttt{algred}\_instr where an atom is translated to an algebraic expression by \texttt{algred}\_atom.

Definition algred\_atom \texttt{a} :=
match \texttt{a} with
\mid \texttt{Avar v} \Rightarrow \texttt{Evar v}
\mid \texttt{Aconst ty bs} \Rightarrow \texttt{Econst (bv2z \texttt{ty bs)}}
end.

Definition algred\_instr \texttt{te} \texttt{av n g} :=
match \texttt{g} with
| \texttt{Iadd v a1 a2} ⇒
  \texttt{algred}\_atom \texttt{a1 in}
  \texttt{algred}\_atom \texttt{a2 in}
  \texttt{Icast v ty a ⇒ algred\_cast avn g v ty a (atyp a te)}
end.

Consider for example the instruction \texttt{Iadd v a1 a2} where \texttt{v} is a variable and \texttt{a1} and \texttt{a2} are two atoms. The execution of the instruction assigns \texttt{v} the bit-vector sum of the values of \texttt{a1} and \texttt{a2} computed by \texttt{addB}. We translate this execution to the equation \texttt{Eeq (Evar v) (Ebinop Eadd za1 za2)} where \texttt{za1} is \texttt{algred}\_atom \texttt{a1} and \texttt{za2} is \texttt{algred}\_atom \texttt{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 \texttt{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 \texttt{s} is \textit{algebraically sound}, defined as the proposition \texttt{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 \texttt{ssa\_spec\_algsnd s} is carried out in our range reduction in Section 6.2.

Let \texttt{v} be a variable, \texttt{ty} be a CL type, \texttt{a} be an atom, \texttt{tb} be a type environment, and \texttt{at} be the CL type of a under \texttt{te}. Consider for another example the algebraic reduction of the instruction \texttt{Icast v ty a under te}. Assume the target CL type \texttt{ty} is \texttt{Tuint wv} and \texttt{at} is \texttt{Tuint wa} where \texttt{wv} and \texttt{wa} are two natural numbers. The execution of the instruction is translated to the equation \texttt{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 \texttt{ty, aty with}
| \texttt{Tuint wv, Tuint wa} ⇒
  \texttt{Tuint wv and at} \texttt{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}.

If the width \texttt{wv} is greater than or equal to \texttt{wa} (\texttt{wv} \(\geq\) \texttt{wa}), then the value of \texttt{a} can be represented in the CL type \texttt{ty} perfectly and thus the equation \texttt{Eeq (Evar v) (algred\_atom a)} must hold. Otherwise, only a part of the value can be represented in \texttt{ty}. In the latter case, there must be some value, denoted by the fresh variable \texttt{discarded}, such that the polynomial equation \texttt{a - discarded \times 2^w = v} holds. This polynomial equation is represented as the algebraic predicate \texttt{algred\_split discarded v (algred\_atom a) wv}. For example, consider casting a constant atom \((1101)_2\) of type \texttt{Tuint 4} to a target type \texttt{Tuint 2}. The casted value is
the lower 2 bits \((11)_2\) of the atom (see \texttt{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 \texttt{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:

\textbf{Lemma} \texttt{algred_espec_sound \(o : \text{options}\) \(s : \text{SSA.spec}\) : 
\begin{align*}
\text{well_formed_ssa_spec} s & \rightarrow \text{valid_rep} (\text{algred_espec} o \ (\text{new_svar_spec} s) \ (\text{espec_of_spec} s)) \\
& \rightarrow \text{valid_espec} (\text{espec_of_spec} s).
\end{align*}

where \texttt{well_formed_ssa_spec} checks if a specification is a well-formed specification in SSA. The lemma \texttt{algred_espec_sound} states that if a well-formed specification \(s\) in SSA is algebraically sound \((\text{ssqspec_algsnd} s)\) and the root entailment problem reduced from the specification holds, i.e. \texttt{valid_rep} \((\text{algred_espec} o \ (\text{new_svar_spec} s) \ (\text{espec_of_spec} s))\), then the algebraic specification \texttt{espec_of_spec} 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 \texttt{ZsG} from the terminating store (of type \texttt{SsG}) 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 \texttt{bv2z}. However this is not enough because there may be fresh variables created for \texttt{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 \texttt{rep} is solved by an external CAS in \texttt{verify_rep} (or its parallel version \texttt{verify_rep_list}).

\textbf{Definition} \texttt{verify_imp \(ip : \text{imp}\) : bool \(:=\)}
\begin{align*}
\text{let } ('L, \_, ps, m, q) & := \text{speexprs_of_imp} \, ip \, \text{in} \\
\text{let } ('cs, c) & := \text{ext_solve_imp} \, ps \, m \, q \, \text{in} \\
\text{validate_imp_answer} \, ps \, m \, q \, cs \, c.
\end{align*}

\textbf{Definition} \texttt{verify_imp \(o : \text{options}\) \(rp : \text{rep}\) : bool \(:=\)}
\begin{align*}
\text{let } '(_, _, _, cs, c) := \text{ext_solve_imp} \, ps \, m \, q \, \text{in} \\
\text{then all verify_imp} \, \text{imps_of_rep_simplified} \, o \, \text{rp} \\
\text{else all verify_imp} \, \text{imps_of_rep} \, \text{rp}.
\end{align*}

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

\textbf{Lemma} \texttt{verify_rep_sound \(o : \text{options}\) \(rp : \text{rep}\) : \texttt{verify_rep} \(o \, rp \rightarrow \text{valid_rep} \, rp\).}

\textbf{Lemma} \texttt{verify_rep_list_sound \(o : \text{options}\) \(rp : \text{rep}\) : \texttt{verify_rep_list} \(o \, rp \rightarrow \text{valid_rep} \, rp\).}

### 6.2 Range Specification Verification

The range reduction converts bit-accurate verification problems to SMT QF_BV queries. In \texttt{verify_ssa}, two bit-accurate verification problems are reduced from a specification in SSA and solved through \texttt{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 \texttt{rngred_rspec}, the latter problem is reduced through \texttt{rngred_algsnd}.

\textbf{Definition} \texttt{rngred_rspec_algsnd \((s : \text{SSA.spec}) : \text{seq} \text{QFBV}\) : \texttt{bexp ::= \text{rngred_rspec} \((s : \text{SSA.spec}) \, \text{bexp} \, := \text{rngred_rspec} \, (s) \, + \, \text{rngred_algsnd} \, (s)\).}

\textbf{Definition} \texttt{verify_rspec_algsnd \((s : \text{SSA.spec}) : \text{bool} := \)}
\begin{align*}
\text{let } \text{es} & := \text{bexp_rbexp} \, \text{rngred_rspec} \, \text{algred_algsnd} \, \text{es} \, \text{in} \\
\text{let } \text{bb_hbexps_cache} \, \text{fE} & := \text{map} \, \text{QFBVHash.hash_bexp} \, \text{es} \, \text{in} \\
\text{ext_all_unsat} \, \text{cnfs}.
\end{align*}

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 CoQFBV, 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 CoQFBV 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 \texttt{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 \texttt{simplify_bexp}. It remains to show how the range reduction is applied in \texttt{rngred_rspec} and \texttt{rngred_algsnd}.

The function \texttt{rngred_rspec} first converts the input specification \(s\) to a range specification \texttt{rspec_of_spec} \(s\) of type \texttt{rspec} and then reduces the validity of \texttt{rspec_of_spec} \(s\) to SMT QF_BV queries through functions \texttt{bexp_rbexp}.
and bexp_program. Intuitively, for an atomic range predicate e in the post-condition of a range specification s, rngred_rspre 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 CRTPOLINE 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 the value of the variables 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 al a2 : QFBV.bexp := qfbv_lneg (qfbv_uaddo (qfbv_atom al) (qfbv_atom a2)). |
| Definition bexp_atom_uaddB_algsnd al a2 : QFBV.bexp := qfbv_lneg (qfbv_uaddo (qfbv_atom al) (qfbv_atom a2)). |
| Definition bexp_atom_uaddB_algsnd al a2 : QFBV.bexp := qfbv_lneg (qfbv_uaddo (qfbv_atom al) (qfbv_atom a2)). |
| Definition bexp_instr_algsnd E (i : instr) : QFBV.bexp := match i with | Iadd _ a1 a2 => bexp_atom_uaddB_algsnd E al 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 al) (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_rspre_algsnd_sound (s : SSA.spec) : well_formed_ssa_spec s → verify_rspre_algsnd s → valid_rspec (rspec_of_spec s) ∧ ssa_spec_algsnd s.

Lemma verify_rspre_algsnd_complete (s : SSA.spec) : well_formed_ssa_spec s → valid_rspec (rspec_of_spec s) → ssa_spec_algsnd s → verify_rspre_algsnd s.

The soundness lemma verify_rspre_algsnd_sound states that if verify_rspre_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_rspre_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 → verify_ssa o 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_rspre_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.

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 CoCryptOLINE 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 CoCryptOLINE 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, CoCryptOLINE invokes the certified SMT QF_BV solver CoQFBV [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 CoQFBV 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. CoCryptOLINE 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; CoCryptOLINE 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>
<thead>
<tr>
<th>Function</th>
<th>n</th>
<th>m</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>secp256k1_scalar_mult</td>
<td>106</td>
<td>68.1</td>
<td>3.1</td>
<td>515</td>
</tr>
<tr>
<td>secp256k1_scalar_reduce_512</td>
<td>103</td>
<td>2.5</td>
<td>1.0</td>
<td>515</td>
</tr>
<tr>
<td>secp256k1_scalar_neg</td>
<td>296</td>
<td>132.0</td>
<td>5.3</td>
<td>111</td>
</tr>
<tr>
<td>secp256k1_scalar_sum</td>
<td>808</td>
<td>456.0</td>
<td>13.3</td>
<td>103</td>
</tr>
<tr>
<td>secp256k1_cube_neg</td>
<td>308</td>
<td>52.7</td>
<td>5.4</td>
<td>820</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>1111</td>
<td>38.0</td>
<td>1.4</td>
<td>40</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>1305</td>
<td>60.0</td>
<td>0.7</td>
<td>50</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>225</td>
<td>86.9</td>
<td>4.9</td>
<td>23</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>225</td>
<td>86.9</td>
<td>4.9</td>
<td>23</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>225</td>
<td>86.9</td>
<td>4.9</td>
<td>23</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>225</td>
<td>86.9</td>
<td>4.9</td>
<td>23</td>
</tr>
<tr>
<td>secp256k1_cube_sum</td>
<td>225</td>
<td>86.9</td>
<td>4.9</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 1: Experimental Results

**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 $F_q[X]/(X^n - c)$ to $F_q[X]/(X^n - c) \times F_q[X]/(X^n + c)$, or $f(X) + X^n g(X)$ $\rightarrow (f(X) + cg(X), f(X) - cg(X))$ for deg $f, g < n$. This splits into many $(f_i, g_i) \rightarrow (f_i + cg_i, f_i - cg_i)$ maps — Cooley-Tukey
and multiplication between two images of the NTT map is just pairwise multiplication in $\mathbb{F}_q \cong \mathbb{F}_q[X]/(X^2 - \omega)$. In KYBER, the ring is $\mathbb{Z}_{252}[X]/(X^{256} + 1)$ where $-1$ has a 128th but no 256th 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 $\zeta_j$'s are the principal 256th roots of unity):

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

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”: $ac \equiv (ac' - (ac'') \mod R)R \mod q$, with $R$ a power of 2 (usu. $2^{16}$), $c' = cR \mod q$, and $c'' = c'(q-1) \mod R \mod q$. The division is exact because $ac''q \equiv ac' \mod R$, 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 $F = \sum_{k=0}^{255} f_kX^k$ 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 $\sum_{k=0}^{255} g_{i,k}X^k$ be the $j$-th polynomial obtained at the end of layer $i$ with $0 \leq i \leq 6$ and $0 \leq j < 2^{i+1}$. 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 $0 \leq k < 128$:

$$g_{0,0,k} \equiv f_k + \zeta_0 g_{0,0,k+128} \mod q,$$
and
$$g_{0,1,k} \equiv f_k - \zeta_0 g_{0,0,k+128} \mod q.$$

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

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

$$(3+i)q \leq g_{i,j,k} \leq (3+i)q,$$
for $0 \leq j < 2^{i+1}$ and $0 \leq k < 256/2^{i+1}$. And the range post-conditions of layer $i-1$ are required as the range pre-conditions for layer $i$ with $1 \leq i \leq 6$.

### References


A ecp_nistz256_mul_montx

Figure 3 shows the complete CRYPTO LINE specification for the assembly subroutine ecp_nistz256_mul_montx in OpenSSL.

B Typed CRYPTO LINE

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

B.1 Syntax

Figure 4 gives the formal CRYPTO LINE syntax in COQCRIPTO LINE. A type is represented by Tuint w or Tint w for a natural number w in COQCRIPTO LINE 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 Iadcx), 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, Iadcx, 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 = 0x0000000000000000,
m2 = 0x0000000000000000,
m3 = 0xffffffff00000001 ] }

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

and [ a0, a1, a2, a3, b0, b1, b2, b3 ] <u limbs 64 [m0, m1, m2, m3] } }

move L0x7fffffff@64 a0; move L0x7fffffff@64 a1; move L0x7fffffff@64 a2; move L0x7fffffff@64 a3;
mov rdx r9; mov rdx r11; mov rdx r13; mov rdx r12; mov rdx r8; mov rdx r10; mov rdx r14; mov rdx r15;mov c1 L0x7fffffffa0; mov c0 L0x7fffffffa0;mov r8 L0x7fffffffda0;mov r9 L0x7fffffffc0;mov r10 L0x7fffffffd0;mov r11 L0x7fffffffe0;mov r12 L0x7fffffff0;mov r13 L0x7fffffff0;mov r14 L0x7fffffff4;mov r15 L0x7fffffff8;

model L0x7fffffffda0 0xffffffff00000001@uint64;
model L0x7fffffff00 0x0000000000000000@uint64;
model L0x7fffffff80 0x00000000ffffffff@uint64;
model L0x7fffffff90 0xffffffffffffffff@uint64;

(** ecp_nists256_mul_montx STARTS **)
mov rdx L0x7fffffffda0;mov rdx L0x7fffffffda0;mov rdx L0x7fffffffda0;mov rdx L0x7fffffffda0;

** CRYPTO LINE Model for ecp_nists256_mul_montx **

Figure 3: CRYPTO LINE Model for ecp_nists256_mul_montx
The value-preserving cast \( \text{Ivpc} \) \( v \) \( t \) \a instruction casts the value of the source atom \( a \) into the designated type \( t \). The non-deterministic instruction \( \text{Inondet} \) \( v \) \( t \) assigns the destination variable \( v \) an arbitrary value in the designated type \( t \). For verification purposes, \text{COQCRYPTO}LINE allows programmers assumptions about executions. The \text{Iassume} \( e \) instruction ensures that the designated predicate \( e \) hold in all executions. \text{Inop} is the null instruction. A program is a sequence of instructions.

The formal syntax of a predicate \( \text{bexp} \) in \text{Iassume} is shown in Figure 5. An algebraic expression \( \text{eexp} \) in algebraic predicates is a variable (\( \text{Evar} \) \( v \)), a constant (\( \text{Econst} \) \( z \)), the negation of an algebraic expression (\( \text{Eneg} \)), the sum (\( \text{Eadd} \)), difference (\( \text{Esub} \)), or product (\( \text{Emul} \)) of two algebraic expressions. Note that the constant \( z \) in an algebraic expression has type \( \text{Z} \), which is the type of unbounded integers in \text{COQ}. Atomic algebraic predicates include equality (\( \text{Eeq} \) \( e_1 \) \( e_2 \)) and modular equality (\( \text{Emod} \) \( e_1 \) \( e_2 \) \( m \)) over algebraic expressions \( e_1 \), \( e_2 \), and \( m \). An atomic algebraic \( \text{bexp} \) is an atomic algebraic predicate or a conjunction (\( \text{Rand} \)) of algebraic predicates.

A range expression \( \text{rexp} \) is a variable (\( \text{Evar} \) \( v \)), a constant (\( \text{Econst} \) \( n \)), the arithmetic negation (\( \text{Rneg} \)), bitwise inversion (\( \text{Rnot} \)), addition (\( \text{Radd} \)), subtraction (\( \text{Rsub} \)), multiplication (\( \text{Rmul} \)), unsigned remainder (\( \text{Rmod} \)), signed remainder (\( \text{Rsrem} \) and \( \text{Rsmod} \)), bitwise AND (\( \text{Rand} \)), bitwise OR (\( \text{Ror} \)), or bitwise XOR (\( \text{Rxor} \)) over range expressions. Constants in range expressions are bit-vectors of bounded lengths. Atomic range predicates are equality (\( \text{Req} \)), signed or unsigned comparisons (\( \text{Rcmp} \)) over range expressions \( \text{rexp} \).

A range predicate \( \text{rbexp} \) is an arbitrary Boolean expression (\( \text{Rneg} \), \( \text{Rand} \), and \( \text{Ror} \)) 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 (Imov 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 Iadc 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 high-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 sarrB 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_instrr (Iasssume e) s s holds if eval_eexp 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 other predicates is defined by corresponding integer functions in COQ (eval_eexp). For algebraic predicates, Etrue evaluates to True. Eq e1 e2 checks if the algebraic expressions e1 and e2 evaluate to the same integer. Eqmod e1 e2 p e2 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_eexp. 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) [19]
: instr → state → 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 : to_bool (eval_atom c s) → S.Upd2 v (addB (eval_atom a1 s) (eval_atom a2 s)) c (1-bits of bool) 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 : to_bool (eval_atom y s) → S.Upd2 v (adcB (to_bool (eval_atom y s)) (eval_atom a1 s) (eval_atom a2 s)).2 c (1-bits of bool) 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 : to_bool (eval_atom c s) → S.Upd2 v ((adcB true (eval_atom a1 s) (invB (eval_atom a2 s))).2) c (1-bits of bool) s t → eval_instr te (Isubc c v a1 a2) s t 
| EIsubb b v a1 a2 y s t : S.Upd v (subB (eval_atom a1 s) (eval_atom a2 s)).2 b (1-bits of bool) s t → eval_instr te (Isubb b v a1 a2 y) 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 
| 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 
| EIshl v a i s t : S.Upd v (sbbB (to_bool (eval_atom y s)) (eval_atom a1 s) (eval_atom a2 s)).2 i s t → eval_instr te (Ishl v a i) 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 
| EIsubb b v a1 a2 y s t : S.Upd v (subB (eval_atom a1 s) (eval_atom a2 s)).2 b (1-bits of bool) s t → eval_instr te (Isubb b v a1 a2 y) s t 

Figure 6: Semantics of CRYPTO LINE Instructions and Predicates
Figure 7: Semantics of CRYPTO LINE Instructions (continued)

Definition b2vz (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 :=
b2vz (TE.typ v E) (S.acc v s).

Definition eval_eunop (op : eunop) (v : Z) : Z := eval_instr te (eval_eunop op v te) s t.

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) (eval_eexp e2 te) s
end.

Definition modulo (a b p : Z) := \exists 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
| Radd => addB v1 v2 | Rsub => subB v1 v2
| Rmul => mulB v1 v2 | Rmod => uremB v1 v2
| Rrem => sremB v1 v2 | Rsext => sext v1 v2
| Rand => sgrB v1 v2 | Rror => orB v1 v2
end.

Definition eval_rcmpop (op : rcmpop) (v1 v2 : bits) : bool :=
match op with
| Rult => ltB v1 v2 | Rule => leB v1 v2
| Rslt => sltB v1 v2 | Rsle => sleB 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.

Definition eval_rbexp (e : rbexp) (s : S.t) : bool :=
match e with
| Rvar v => S.acc v s
| Rconst w n => n
| Rbit _ op e => eval_runop op (eval_rexp e s)
| Rcmp _ op e1 e2 =>
  eval_rcmpop op (eval_rexp e1 s) (eval_rexp e2 s)
| Rneg e => ¬(eval_rexp e s)
| Ror e1 e2 => (eval_rbexp e1 s) ∨ (eval_rbexp e2 s)
end.

Fixpoint eval_rerxp (e : rerxp) (s : S.t) : bool :=
match e with
| Rtrue => true
| Rreq e1 e2 => eval_rexp e1 s ∨ eval_rexp e2 s
| Rcmpeq e1 e2 =>
  eval_rcmpop op (eval_rexp e1 s) (eval_rexp e2 s)
| Rneg e => ¬(eval_rerxp e s)
| Rbit _ op e1 e2 =>
  eval_rbinop op (eval_rexp e1 s) (eval_rexp e2 s)
| Ror e1 e2 => (eval_rbexp e1 s) ∨ (eval_rbexp e2 s)
end.

Figure 8: Semantics of CRYPTO LINE Predicates