CoqCryptoLine: A Verified Model Checker with Certified Results

Ming-Hsien Tsai[4(✉)], Yu-Fu Fu[2], Jiaxiang Liu[5], Xiaomu Shi[3], Bow-Yaw Wang[1], and Bo-Yin Yang[1]

[1] Academia Sinica, Taipei, Taiwan
{bywang,byyang}@iis.sinica.edu.tw
[2] Georgia Institute of Technology, Atlanta, USA
yufu@gatech.edu
[3] Institute of Software, Chinese Academy of Sciences, Beijing, China
xshi0811@gmail.com
[4] National Institute of Cyber Security, Taipei, Taiwan
mhtsai208@gmail.com
[5] Shenzhen University, Shenzhen, China
jiaxiang0924@gmail.com

Abstract. We present the verified model checker CoqCryptoLine for cryptographic programs with certified verification results. The CoqCryptoLine verification algorithm consists of two reductions. The algebraic reduction transforms into a root entailment problem; and the bit-vector reduction transforms into an SMT QF_BV problem. We specify and verify both reductions formally using Coq with MathComp. The CoqCryptoLine tool is built on the OCaml programs extracted from verified reductions. CoqCryptoLine moreover employs certified techniques for solving the algebraic and logic problems. We evaluate CoqCryptoLine on cryptographic programs from industrial security libraries.

1 Introduction

CoqCryptoLine [1] is a verified model checker with certified verification results. It is designed for verifying complex non-linear integer computations commonly found in cryptographic programs. The verification algorithms of CoqCryptoLine consist of two reductions. The algebraic reduction transforms polynomial equality checking into a root entailment problem in commutative algebra; the bit-vector reduction reduces range properties to satisfiability of queries in the Quantifier-Free Bit-Vector (QF_BV) logic from Satisfiability Modulo Theories (SMT) [6]. Both verification algorithms are formally specified and verified by the proof assistant Coq with MathComp [7,17]. CoqCryptoLine verification programs are extracted from the formal specification and therefore verified by the proof assistant automatically.

The original version of this chapter was revised: The mistakes in authors affiliation information and typographical errors have been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-031-37703-7_22

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 227–240, 2023.
https://doi.org/10.1007/978-3-031-37703-7_11

To minimize errors from external tools, recent developments in certified verification are employed by CoqCryptoLine. The root entailment problem is solved by the computer algebra system (CAS) Singular [19]. CoqCryptoLine asks the external algebraic tool to provide certificates and validates certificates with the formal polynomial theory in Coq. SMT QF_BV queries on the other hand are answered by the verified SMT QF_BV solver CoqQFBV [33]. Answers to SMT QF_BV queries are therefore all certified as well. With formally verified algorithms and certified answers from external tools, CoqCryptoLine gives verification results with much better guarantees than average automatic verification tools.

Reliable verification tools would not be very useful if they could not check real-world programs effectively. In our experiments, CoqCryptoLine verifies 54 real-world cryptographic programs. 52 of them are from well-known security libraries such as Bitcoin [35] and OpenSSL [30]. They are implementations of field and group operations in elliptic curve cryptography. The remaining two are the Number-Theoretic Transform (NTT) programs from the post-quantum cryptosystem Kyber [10]. All field operations are implemented in a few hundred lines and verified in 6 minutes. The most complicated generic group operation in the elliptic curve Curve25519 consists of about 4000 lines and is verified by CoqCryptoLine in 1.5 h.

Related Work. There are numerous model checkers in the community, e.g. [8, 13,21–23]. Nevertheless, few of them are formally verified. To our knowledge, the first verification of a model checker was performed in Coq for the modal μ-calculus [34]. The LTL model checker CAVA [15,27] and the model checker Munta [38,39] for timed automata were developed and verified using Isabelle/HOL [29], which can be considered as verified counterparts of SPIN [21] and Uppaal [23], respectively. CoqCryptoLine instead checks CryptoLine models [16,31] that are for the correctness of cryptographic programs. It can be seen as a verified version of CryptoLine. A large body of work studies the correctness of cryptographic programs, e.g. [2–4,9,12,14,24,26,40], cf. [5] for a survey. They either require human intervention or are unverified, while our work is fully automatic and verified. The most relevant work is bvCryptoLine [37], which is the first automated and partly verified model checker for a very limited subset of CryptoLine. We will compare our work with it comprehensively in Sect. 2.3.

2 CoqCryptoLine

CoqCryptoLine is an automatic verification tool that takes a CryptoLine specification as input and returns certified results indicating the validity of the specification. We briefly describe the CryptoLine language [16] followed by the modules, features, and optimizations of CoqCryptoLine in this section.

2.1 CryptoLine Language

A CryptoLine specification contains a CryptoLine program with pre- and post-conditions, where the CryptoLine program usually models some

cryptographic program [16,31]. Both the pre- and post-conditions consist of an algebraic part, which is formulated as a conjunction of (modular) equations, and a range part as an SMT QF_BV predicate. A CryptoLine specification is valid if every program execution starting from a program state satisfying the pre-condition ends in a state satisfying the post-condition.

CryptoLine is designed for modeling cryptographic assembly programs. Besides the assignment (MOV) and conditional assignment (CMOV) statements, CryptoLine provides arithmetic statements such as addition (ADD), addition with carry (ADC), subtraction (SUB), subtraction with borrow (SBB), half multiplication (MUL) and full multiplication (MULL). Most of them have versions that model the carry/borrow flags explicitly (like ADDS, ADCS, SUBS, SBBS). It also allows bitwise statements, for instance, bitwise AND (AND), OR (OR) and left-shift (SHL). To deal with multi-word arithmetic, CryptoLine further includes multi-word constructs, for example, those that split (SPLIT) or join (JOIN) words, as well as multi-word shifts (CSHL). CryptoLine is strongly typed, admitting both signed and unsigned interpretations for bit-vector variables and constants. The CAST statement converts types explicitly. Finally, CryptoLine also supports special statements (ASSERT and ASSUME) for verification purposes.

2.2 The Architecture of CoqCryptoLine

CoqCryptoLine reduces the verification problem of a CryptoLine specification to instances of root entailment problems and SMT problems over the QF_BV logic. These instances are then solved by respective certified techniques. Moreover, the components in CoqCryptoLine are also specified and verified by the proof assistant Coq with MathComp [7,17]. Figure 1 gives an overview of CoqCryptoLine. In the figure, dashed components represent external tools. Rectangular boxes are verified components and rounded boxes are unverified. Note that all our proof efforts using Coq are transparent to users. No Coq proof is required from users during verification of cryptographic programs with CoqCryptoLine. Details can be found in [36].

Starting from a CryptoLine specification text, the CoqCryptoLine parser translates the text into an abstract syntax tree defined in the Coq module `DSL`. The module gives formal semantics for the typed CryptoLine language [16]. The validity of CryptoLine specifications is also formalized. 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 the verification problem via two Coq modules.

The `SSA2ZSSA` module contains our algebraic reduction to the root entailment problem. Concretely, a system of (modular) equations is constructed from the given program so that program executions correspond to the roots of the system of (modular) equations. To verify algebraic post-conditions, it suffices to check if the 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 will generate

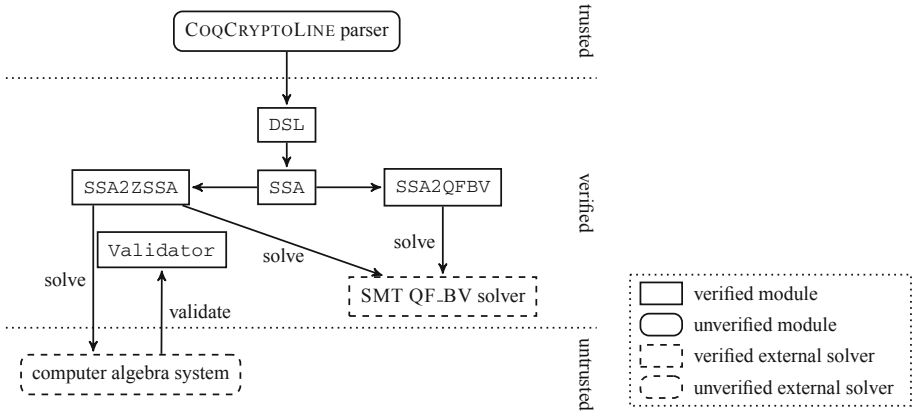


Fig. 1. Overview of CoqCryptoLine

soundness conditions to ensure the executions conform to our (modular) equations. The algebraic 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 an SMT query to check the validity of the given CryptoLine range specification. Concretely, an SMT QF_BV query is built such that all program executions correspond to satisfying assignments to the query and vice versa. To verify the range post-conditions, it suffices to check if satisfying assignments for the query also satisfy the post-conditions. The range verification problem is thus reduced to the SMT QF_BV problem. On the other hand, additional SMT 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 formally verified reduction algorithms, it remains to solve the root entailment problems and the SMT QF_BV problems with external solvers. CoqCryptoLine invokes an external computer algebra system (CAS) to solve the root entailment problems, and improves the techniques in [20,37] to validate the (untrusted) returned answers. Currently, the CAS Singular [19] is supported. To solve the SMT QF_BV problems, CoqCryptoLine employs the certified SMT QF_BV solver CoqQFBV [33]. In all cases, instances of the two kinds of problems are solved with certificates. And CoqCryptoLine employs verified certificate checkers to validate the answers to further improve assurance.

Note that the algebraic reduction in `SSA2ZSSA` is sound but not complete due to the abstraction of bit-accurate semantics into (modular) polynomial equations over integers. Thus a failure in solving the root entailment problem by CAS does not mean that the algebraic post-conditions are violated. On the other hand, the bit-vector reduction in `SSA2QFBV` is both sound and complete.

The CoqCryptoLine tool is built on OCaml programs extracted from verified algorithms in Coq with MathComp. We moreover integrate the OCaml

programs from the certified SMT QF_BV solver CoqQFBV. 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 the SAT solver certificate validator Grat used by CoqQFBV) and (4) the Coq proof assistant. Particularly, sophisticated decision procedures in external CASs and SAT solvers used in CoqQFBV need not be trusted.

2.3 Features and Optimizations

CoqCryptoLine comes with the following features and optimizations implemented in its modules.

Type System. CoqCryptoLine fully supports the type system of the CryptoLine language. The type system is used to model bit-vectors of arbitrary bit-widths with unsigned or signed interpretation. Such a type system allows CoqCryptoLine to model more industrial examples translated from C programs via GCC [16] or LLVM [24] compared to bvCryptoLine [37], which only allows unsigned bit-vectors, all of the same bit-width.

Mixed Theories. With the assert and assume statements supported by CoqCryptoLine, it is possible to make an assertion on the range side (or on the algebraic side) and then make an equivalent assumption on the algebraic side (or resp. on the range side). With this feature, a predicate can be asserted on one side where the predicate is easier to prove, and then assumed on the other side to ease the verification of other predicates. The equivalence between the asserted predicate and the assumed predicate is currently not verified by CoqCryptoLine, though it is achievable. Both assert and assume statements are not available in bvCryptoLine.

Multi-threading. All extracted OCaml code from the verified algorithms in Coq runs sequentially. To speed up, SMT QF_BV problems, as well as root entailment problems, are solved parallelly.

Efficient Root Entailment Problem Solving. CoqCryptoLine can be used as a solver for root entailment problems with certificates validated by a verified validator. A root entailment problem is reduced to an ideal membership problem, which is then solved by computing Gröbner basis [20]. To solve a root entailment problem with a certificate, we need to find a witness of polynomials $c_0, \ldots, c_n$ such that

$$q = \Sigma_{i=0}^{n} c_i p_i \tag{1}$$

where q and p_i's are given polynomials. To compute the witness, bvCryptoLine relies on `gbarith` [32], where new variables are introduced. CoqCryptoLine utilizes the `lift` command in Singular instead without adding fresh variables. We show in the evaluation section that using `lift` is more efficient than using `gbarith`. The witness found is further validated by CoqCryptoLine, which

relies on the polynomial normalization procedure `norm_subst` in Coq to check if Eq. 1 holds. bvCryptoLine on the other hand uses the `ring` tactic in Coq, where extra type checking is performed. Elimination of ideal generators through variable substitution is an efficient approach to simplify an ideal membership problem [37]. The elimination procedure implemented in CoqCryptoLine can identify much more variable substitution patterns than those found by bvCryptoLine.

Multi-moduli. Modular equations with multi-moduli are common in post-quantum cryptography. For example, the post-quantum cryptosystem Kyber uses the polynomial ring $\mathbb{Z}_{3329}[X]/\langle X^{256}+1\rangle$ containing two moduli 3329 and $X^{256}+1$. To support multi-moduli in CoqCryptoLine, in the proof of our algebraic reduction, we have to find integers $c_0, \ldots, c_n$ such that $e_1 - e_2 = \Sigma_{i=0}^{n} c_i m_i$ given the proof of $e_1 = e_2 \pmod{m_0, \ldots, m_n}$ where e_1, e_2, and m_i's are integers. Instead of implementing a complicated procedure to find the exact c_i's, we simply invoke the `xchoose` function provided by MathComp to find c_i's based on the proof of $e_1 = e_2 \pmod{m_0, \ldots, m_n}$. Multi-moduli is not supported by bvCryptoLine.

Tight Integration with CoqQFBV. CoqCryptoLine verifies every atomic range predicate separately using the certified SMT QF_BV solver CoqQFBV. Constructing a text file as the input to CoqQFBV for every atomic range predicate is not a good idea because the bit-blasting procedure in CoqQFBV is performed several times for the identical program. CoqCryptoLine thus is tightly integrated with CoqQFBV to speed up bit-blasting of the same program using the cache provided by CoqQFBV. bvCryptoLine uses the SMT solver Boolector to prove range predicates without certificates.

Slicing. During the reductions from the verification problem of a CryptoLine specification to instances of root entailment problems and SMT QF_BV problems, a verified static slicing is performed in CoqCryptoLine to produce smaller problems. Unlike the work in [11], which sets all assume statements as additional slicing criteria, the slicing in CoqCryptoLine is capable of pruning unrelated predicates in assume statements. The slicing procedure implemented in CoqCryptoLine is much more complicated than the one in bvCryptoLine due to the presence of assume statements. This feature is provided as command-line option because it makes the verification incomplete. With slicing, the time in verifying industrial examples is reduced dramatically.

3 Walkthrough

We illustrate how CoqCryptoLine is used in this section. The x86_64 assembly subroutine `ecp_nistz256_mul_montx` from OpenSSL [30] shown in Fig. 2 is verified as an example.

An input for CoqCryptoLine contains a CryptoLine specification for the assembly subroutine. The original subroutine is marked between the comments

`PROGNAME STARTS` and `PROGNAME ENDS`, which is obtained automatically from the Python script provided by CryptoLine [31].

Prior to the "START" comment are the parameter declaration, pre-condition, and variable initialization. After the "END" comment is the post-condition of the subroutine. After the subroutine ends, the result is moved to the output variables.

The assembly subroutine `ecp_nistz256_mul_montx` takes two 256-bit unsigned integers a and b and the modulus m as inputs. The 256-bit integer m is the prime $p256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ from the NIST curve. The 256-bit integers a and b (less than the prime) are the multiplicands. Each 256-bit input integer $d \in \{a, b, m\}$ is denoted by four 64-bit unsigned integer variables d_i (for $0 \leq i < 4$) in little-endian representation. The expression `limbs n [`d_0`, `d_1`, ..., `d_i`]` is short for ``$d_0$ + $d_1$*2**n + ... +$d_i$*2**($i$*n)``[1]. The inputs and constants are then put in the variables for memory cells with the MOV statements. There are two parts to a pre-condition. The first part is for the algebraic reduction; the second part is for the bit-vector reduction:

```
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 output 256-bit integer represented by the four variables c_i (for $0 \leq i < 4$) has two requirements. Firstly, the output integer times 2^{256} equals the product of the input integers modulo $p256$. Secondly, the output integer is less than $p256$. Formally, we have this post-condition:

```
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, and the bit-vector reduction to verify the proper range of the output integer.

However, verifying `ecp_nistz256_mul_montx` takes extra annotations to hint CoqCryptoLine how to verify the post-condition. E.g., in adding two 256-bit integers represented by 64-bit variables, a chain of four 64-bit additions is performed and carries are propagated. The last carry as the chain ends must be zero or the 256-bit sum is incorrect. In `ecp_nistz256_mul_montx` two interleaved addition chains use the carry and the overflow flags for carries respectively, so we annotate as follows at the end of two interleaving addition chains to tell CoqCryptoLine about the final carries:

[1] ** is the exponentiation operator in CryptoLine.

```
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;

...

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;
split ddc rcx r9 32;
shl rcx rcx 32;
adcs overflow r13 r13 rbp overflow;
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;

...

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] }
```

Fig. 2. CryptoLine Model for `ecp_nistz256_mul_montx`

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

The ASSERT statement verifies that both the carry and overflow flags are zeroes through the bit-vector reduction. The ASSUME statement then passes this information to the algebraic reduction. Effectively, CoqCryptoLine checks that both flags are zero for all inputs satisfying the pre-condition, then uses those facts as lemmas to verify the post-condition with the algebraic reduction.

The full specification for `ecp_nistz256_mul_montx` has 230 lines, including 50 lines of manual annotations. 20 are straightforward annotations for variable declaration and initialization. The remaining 30 lines of annotations are hints to CoqCryptoLine, which then verifies the post-condition in 30 s with 24 threads.

The illustration of the typical verification flow shows how a user constructs a CryptoLine specification. The pre-condition for program inputs, the post-condition for outputs, and variable initialization must be specified manually. Additional annotations may be added as hints. Notice that hints only tell CoqCryptoLine *what*, not *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 Evaluation

We evaluate CoqCryptoLine on 52 benchmarks from four industrial security libraries Bitcoin [35], boringSSL [14,18], nss [25], and OpenSSL [30]. The C reference and optimized avx2 implementations of the Number-Theoretic Transform (NTT) from the post-quantum key encapsulation mechanism Kyber [10] are also evaluated. Among the total 54 benchmarks, 43 benchmarks contain features not supported by bvCryptoLine such as signed variables. All experiments are performed on an Ubuntu 22.04.1 machine with a 3.20GHz Intel Xeon Gold 6134M CPU and 1TB RAM.

Benchmarks from security libraries are various field and group operations from elliptic curve cryptography (ECC). In ECC, rational points on curves are represented by elements in large finite fields. In Bitcoin, the finite field is the residue system modulo the prime $p256k1 = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. For other security libraries (boringSSL, nss, and OpenSSL), we verify the operations in Curve25519 using the residue system modulo the prime $p25519 = 2^{255} - 19$ as the underlying field. Rational points on elliptic curves form a group. The group operation in turn is implemented by a number of field operations.

In lattice-based post-quantum cryptosystems, polynomial rings are used. Specifically, the polynomial ring $\mathbb{Z}_{3329}[X]/\langle X^{256} + 1\rangle$ is used in Kyber. To speed up multiplication in the polynomial ring, Kyber requires the multiplication to be implemented by NTT. NTT is a discrete Fast Fourier Transform over finite fields. Instead of complex roots of unity, NTT uses the principal roots of unity in fields. Mathematically, the Kyber NTT computes the following ring isomorphism

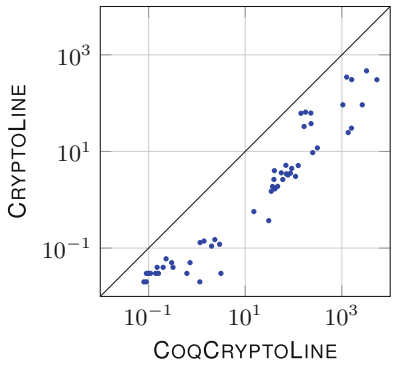
$$\mathbb{Z}_{3329}[X]/\langle X^{256} + 1\rangle \cong \mathbb{Z}_{3329}[X]/\langle X^2 - \zeta_0\rangle \times \cdots \times \mathbb{Z}_{3329}[X]/\langle X^2 - \zeta_{127}\rangle$$

where ζ_i's are the principal roots of unity.

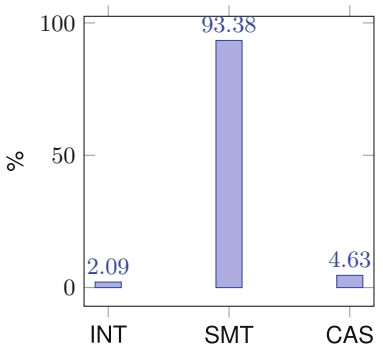
We first compare CoqCryptoLine with all optimizations described in this paper against the unverified model checker CryptoLine [16]. Both tools invoke the computer algebra system Singular [19], but CryptoLine neither lets Singular produce certificates nor certifies answers from Singular. CoqCryptoLine moreover uses the certified SMT QF_BV solver CoqQFBV [33]; CryptoLine uses the uncertified but very efficient Boolector [28].

For the ECC experiments, CoqCryptoLine verifies all field operations in 6 minutes. It takes a few thousand seconds to verify group operations. The most complex implementation (x25519_scalar_mult_generic) from boringSSL (4274 statements) takes about 1.5 hours.[2] For Kyber, CoqCryptoLine verifies in 2642 and 1048 seconds, respectively, that the reference and avx2 NTT implementations indeed compute the isomorphism. The unverified CryptoLine in comparison finishes verification in about 95 seconds. A summary of the comparison between CoqCryptoLine and CryptoLine is shown in Fig. 3a. Though CoqCryptoLine is much slower than CryptoLine, the running time (1.5 hours) for the most complex implementation is still acceptable.

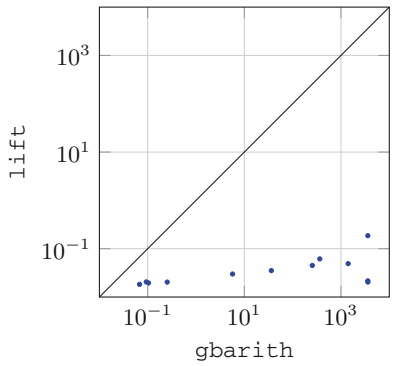
[2] Two (out of three) modular polynomial equations in the post-condition are certified.



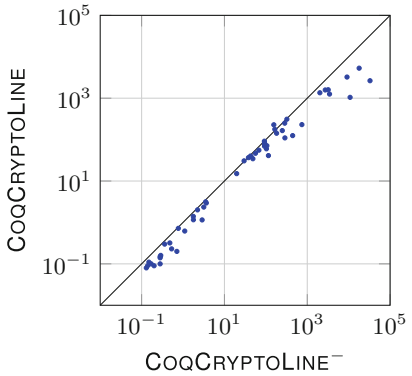
(a) CoqCryptoLine versus CryptoLine



(b) Percentages of average running time for CoqCryptoLine internal OCaml code (INT), external SMT QF_BV solver (SMT), and external computer algebra system (CAS)



(c) gbarith versus lift



(d) CoqCryptoLine$^-$ versus CoqCryptoLine

Fig. 3. Running time (in seconds) comparisons

Figure 3b shows the percentages of average running time for CoqCryptoLine internal OCaml code (INT), external SMT QF_BV solver (SMT), and external computer algebra system (CAS). External solvers take much more time than the internal OCaml program does. Between external solvers, the external computer algebra system takes 4.63% of the time and the external SMT QF_BV solver spends 93.28% of the time.

To show the performance of the `lift` optimization, we run CoqCryptoLine and bvCryptoLine on root entailment problems generated from the benchmarks. Here we only consider 12 root entailment problems that trigger `gbarith` in bvCryptoLine. Figure 3c shows the running time of Singular in solving

root entailment problems based on `gbarith` in bvCryptoLine and `lift` in CoqCryptoLine. bvCryptoLine fails to solve 3 root entailment problems in one hour. For the other 9 root entailment problems, `lift` outperforms `gbarith`.

We also compare CoqCryptoLine with and without slicing. The version of CoqCryptoLine without slicing is denoted by CoqCryptoLine$^-$. The running time comparison between CoqCryptoLine and CoqCryptoLine$^-$ in Fig. 3d shows that slicing reduces the running time obviously.

5 Conclusion

CoqCryptoLine is a verified model checker for cryptographic programs with certified results. Its modules are formally verified in Coq with MathComp. CoqCryptoLine moreover employs external tools and validates their answers with certificates. We evaluate CoqCryptoLine on benchmarks from industrial security libraries (Bitcoin, boringSSL, nss and OpenSSL) and a post-quantum cryptography standard candidate (Kyber). In our experiments, CoqCryptoLine verifies most cryptographic programs with certificates in a reasonable time (6 min). Benchmarks with thousands of lines are verified in 1.5 h. To our knowledge, this is the first certified verification on operations of the elliptic curve secp256k1 used in Bitcoin, and the avx2 and reference implementations of Kyber number-theoretic transform.

Acknowledgments. The authors in Academia Sinica are partially funded by National Science and Technology Council grants NSTC110-2221-E-001-008-MY3, NSTC111-2221-E-001-014-MY3, NSTC111-2634-F-002-019, the Sinica Investigator Award AS-IA-109-M01, the Data Safety and Talent Cultivation Project AS-KPQ-109-DSTCP, and the Intel Fast Verified Postquantum Software Project. The authors in Shenzhen University and ISCAS are partially funded by Shenzhen Science and Technology Innovation Commission (JCYJ20210324094202008), the National Natural Science Foundation of China (62002228, 61836005), and the Natural Science Foundation of Guangdong Province (2022A1515011458, 2022A1515010880).

References

1. CoqCryptoLine GitHub repository (2023). https://github.com/fmlab-iis/coq-cryptoline
2. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. Innov. Syst. Softw. Eng. **9**(2), 59–77 (2013)
3. Almeida, J.B., et al.: Jasmin: High-assurance and high-speed cryptography. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1807–1823. ACM (2017)
4. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Programm. Lang. Syst. **37**(2), 7:1–7:31 (2015)
5. Barbosa, M., et al.: Sok: Computer-aided cryptography. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021, pp. 777–795. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00008

6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
8. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
9. Bond, B., et al.: Vale: Verifying high-performance cryptographic assembly code. In: USENIX Security Symposium, pp. 917–934. USENIX Association (2017)
10. Bos, J., et al.: CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. In: Smith, M., Piessens, F. (eds.) IEEE European Symposium on Security and Privacy, pp. 353–367. IEEE (2018)
11. Chalupa, M., Strejcek, J.: Evaluation of program slicing in software verification. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11918, pp. 101–119. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_6
12. Chen, Y.F., et al.: Verifying Curve25519 software. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 299–309. ACM (2014)
13. Cimatti, A., et al.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: IEEE Symposium on Security and Privacy, pp. 1202–1219. IEEE (2019)
15. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 463–478. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_31
16. Fu, Y.F., Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Signed cryptographic program verification with typed cryptoline. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 1591–1606. ACM (2019)
17. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formalized Reason. **3**(2), 95–152 (2010)
18. Google: Boringssl (2021). https://boringssl.googlesource.com/boringssl/
19. Greuel, G.M., Pfister, G.: A Singular Introduction to Commutative Algebra. Springer-Verlag (2002)
20. Harrison, J.: Automating elementary number-theoretic proofs using Gröbner bases. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 51–66. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_5
21. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)

22. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002). http://research.microsoft.com/users/lamport/tla/book.html
23. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1-2), 134–152 (1997). https://doi.org/10.1007/s100090050010
24. Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic in cryptographic C programs. In: Lawall, J., Marinov, D. (eds.) IEEE/ACM International Conference on Automated Software Engineering, pp. 552–564. IEEE (2019)
25. Mozilla: Network security services (2021). https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
26. Myreen, M.O., Curello, G.: Proof Pearl: a verified bignum implementation in x86-64 machine code. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 66–81. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_5
27. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8471, pp. 105–114. Springer (2014). https://doi.org/10.1007/978-3-319-12154-3_7
28. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisfiability, Boolean Modeling Comput. **9**(1), 53–58 (2014)
29. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
30. OpenSSL: OpenSSL library. https://github.com/openssl/openssl (2021)
31. Polyakov, A., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic assembly programs in cryptographic primitives. In: Schewe, S., Zhang, L. (eds.) International Conference on Concurrency Theory, pp. 4:1–4:16. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
32. Pottier, L.: Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008). http://ceur-ws.org/Vol-418/paper5.pdf
33. Shi, X., Fu, Y.F., Liu, J., Tsai, M.H., Wang, B.Y., Yang, B.Y.: CoqQFBV: a scalable certified SMT quantifier-free bit-vector solver. In: Leino, R., Silva, A. (eds.) International Conference on Computer Aided Verification. Springer, Lecture Notes in Computer Science (2021)
34. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1384, pp. 167–183. Springer (1998). https://doi.org/10.1007/BFb0054171
35. The Bitcoin Developers: Bitcoin source code (2021). https://github.com/bitcoin/bitcoin
36. Tsai, M.H., Fu, Y.F., Shi, X., Liu, J., Wang, B.Y., Yang, B.Y.: Automatic certified verification of cryptographic programs with COQCRYPTOLINE. IACR Cryptol. ePrint Arch. p. 1116 (2022)

37. Tsai, M.H., Wang, B.Y., Yang, B.Y.: Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In: Evans, D., Malkin, T., Xu, D. (eds.) ACM SIGSAC Conference on Computer and Communications Security, pp. 1973–1987. ACM (2017)
38. Wimmer, S.: Munta: A verified model checker for timed automata. In: André, É., Stoelinga, M. (eds.) Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 236–243. Springer (2019). https://doi.org/10.1007/978-3-030-29662-9_14
39. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10805, pp. 61–78. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_4
40. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1789–1806. ACM (2017)