Certified Verification for Algebraic Abstraction

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
bywang@iis.sinica.edu.tw
byyang@iis.sinica.edu.tw
[2] Georgia Institute of Technology
yufu@gatech.edu
[3] Institute of Software, Chinese Academy of Sciences
xshi0811@gmail.com
[4] National Institute of Cyber Security
mhtsai208@gmail.com
[5] Shenzhen University
jiaxiang0924@gmail.com(✉)

Abstract. We present a certified algebraic abstraction technique for verifying bit-accurate non-linear integer computations. In algebraic abstraction, programs are lifted to polynomial equations in the abstract domain. Algebraic techniques are employed to analyze abstract polynomial programs; SMT QF_BV solvers are adopted for bit-accurate analysis of soundness conditions. We explain how to verify our abstraction algorithm and certify verification results. Our hybrid technique has verified non-linear computations in various security libraries such as BITCOIN and OPENSSL. We also report the certified verification of Number-Theoretic Transform programs from the post-quantum cryptosystem KYBER.

1 Introduction

Bit-accurate non-linear integer computations are infamously hard to verify. Conventional bit-accurate techniques such as bit blasting do not work well for non-linear computations. Approximation techniques through floating-point computation on the other hand are inaccurate. Non-linear integer computation nonetheless is essential to computer cryptography. Analyzing complex non-linear computation in cryptographic libraries is still one of the most challenging problems of the utmost importance today.

In this paper, we address the verification problem through algebraic abstraction. In algebraic abstraction, abstract programs are represented by polynomial equations. Non-linear computation about abstract polynomial programs is analyzed algebraically and hence more efficiently through techniques from commutative algebra. Algebraic abstraction however is unsound due to overflow in bounded integer computation. We characterize soundness conditions with queries using the Quantifier-Free Bit-Vector (QF_BV) logic from Satisfiability Modulo Theories (SMT) [2]. SMT solvers are then used to check soundness conditions before applying algebraic abstraction.

Our hybrid technique takes advantages of both algebraic and bit-accurate analyses. Non-linear algebraic properties are verified algebraically. Polynomials are computed and analyzed by algorithms from commutative algebra. Coefficients, variables and arithmetic functions are atomic in such algorithms. Our algebraic analysis is hence very efficient for non-linear computation. Soundness conditions, on the other hand, require bit-accurate analysis. Our technique applies SMT QF_BV solvers to check soundness conditions. By combining algebraic with bit-accurate analyses, algebraic abstraction successfully verifies non-linear computation in real-world cryptographic programs.

Cryptographic programs undoubtedly are widely deployed critical software. Errors in their verification need to be minimized. To this end, we use the proof assistant COQ [4] to verify the soundness theorem for algebraic abstraction. To ensure the correctness of external algebraic and bit-accurate analysis tools, results from external tools are certified in our technique as well. With verified abstraction and certified external results, verification of bit-accurate non-linear integer computation through algebraic abstraction is certified. We explain how to certify our hybrid verification technique.

We evaluate our certified technique with cryptographic programs from security libraries in BITCOIN [27], BORINGSSL [8,12], NSS [20], OPENSSL [23] and PQCRYPTO-SIDH [18]. These programs compute field and group operations in elliptic curve cryptography. We also verify Number-Theoretic Transform (NTT) programs from the post-quantum cryptosystem KYBER [6]. In lattice-based post-quantum cryptography, computation in polynomial rings is needed. NTT is a discrete variant of the Fast Fourier Transform used for polynomial multiplication in KYBER. Our certified algebraic abstraction technique verifies cryptographic programs from elliptic curve and post-quantum cryptography successfully. Our contributions are summarized as follows.

– We detail algebraic abstraction for checking non-linear modular equations with multiple moduli;
– We certify algebraic abstraction and its verification;
– We report certified verification results for 39 real-world cryptographic programs in elliptic curve and post-quantum cryptography.

Related Work GFVERIF employs an ad hoc technique to verify non-linear computation in cryptographic programs with a computer algebra system [3]. CRYPTOLINE [24,9,29] is a tool designed for the specification and verification of cryptographic assembly codes. Its verification algorithm utilizes computer algebra systems in addition to SMT solvers. CRYPTOLINE is also leveraged to verify cryptographic C programs [17,9]. The optimized KYBER NTT program for avx2 is verified in [15], but the underlying verification algorithm is left unexplained. None of these works certified their verification results. Users had to trust these verification tools. BVCRYPTOLINE certifies algebraic abstraction but not soundness conditions [29]. It does not allow multiple moduli in modular equations either. Particularly, it cannot concisely specify NTT by the Chinese remainder theorem over polynomial rings. Compared with these works, our technique admits modular equations with multiple moduli in assumptions and assertions, and is fully certified. To explicate our advantages, consider the specification of multiplication in the field $\mathbb{Z}_{p434}/\langle x^2 + 1\rangle$ where $p434$ is a prime number. An element in the field is of the form $u_0 + u_1 x$ where $x^2 + 1 = 0$. To specify $r_0 + r_1 x$ is the product of $u_0 + u_1 x$ and $v_0 + v_1 x$, one can write two modular equations with one modulo: $r_0 \equiv u_0 v_0 - u_1 v_1 \bmod [p434]$ and $r_1 \equiv u_0 v_1 + u_1 v_0 \bmod [p434]$. With multiple moduli, we write $r_0 + r_1 x \equiv (u_0 + u_1 x)(v_0 + v_1 x) \bmod [p434, x^2 + 1]$

succinctly. Our simple specifications are most useful for complicated fields such as $\mathbb{Z}_{p381}/\langle x^2+1, y^3-x-1, z^2-y\rangle$. Each element of the complex field is of the form $\sum u_{i,j,k}x^i y^j z^k$ with $0 \leq i,k < 2$ and $0 \leq j < 3$. Twelve modular equations are needed previously. One modular equation with multiple moduli suffices to specify its field multiplication in this work. Furthermore, our technique is verified in Coq. The correctness of our abstraction algorithm and soundness theorem are formally proven in Coq. We also show how to certify results from external tools. In summary, the correctness of algebraic abstraction algorithm is verified and answers from external tools are certified. Verification results are therefore fully certified. We believe this is the best guarantee a model checker can offer. Our verified model checker is sufficiently practical to verify industrial cryptographic programs too!

Analysis of linear polynomial programs was discussed, for instance, in [22,21]. The reduction from the root entailment problem to the ideal membership problem is discussed in [14]. In this work, the computer algebra system Singular [13] is employed to compute standard bases of ideals and certificates. The certified SMT QF_BV solver CoqQFBV [26] is adopted to certify soundness conditions.

The paper is organized as follows. Section 2 gives the needed backgrounds. It is followed by the syntax and semantics of the language ToyLang. An implementation of the unsigned Montgomery reduction is given as a running example (Section 3). Section 4 presents algebraic abstraction and its verification algorithms. We briefly describe certified verification of algebraic abstraction in Section 5. Section 6 shows experimental results of real-world cryptographic programs. We conclude in Section 7.

2 Preliminaries

Let $\mathbb{N}$ and $\mathbb{Z}$ denote the set of non-negative and all integers respectively. Fix a set of variables $\overline{\mathbf{x}}$. We write $\mathbb{Z}[\overline{\mathbf{x}}]$ for the set of polynomials in variables $\overline{\mathbf{x}}$ with coefficients in $\mathbb{Z}$. A polynomial *equation* is of the form $e = e'$ with $e, e' \in \mathbb{Z}[\overline{\mathbf{x}}]$; a polynomial *modular equation* is of the form $e \equiv e' \bmod [f_0, f_1, \ldots, f_m]$ with $e, e', f_0, f_1, \ldots, f_m \in \mathbb{Z}[\overline{\mathbf{x}}]$. A *valuation* ρ of $\overline{\mathbf{x}}$ is a mapping from $\overline{\mathbf{x}}$ to $\mathbb{Z}$. Given a valuation ρ, a polynomial e *evaluates* to the integer $e[\rho]$ by replacing every variable x with $\rho(x)$. A valuation ρ is a *root* of the equation $e = e'$ if $(e - e')[\rho] = 0$. A valuation ρ is a *root* of the modular equation $e \equiv e' \bmod [f_0, f_1, \ldots, f_m]$ if $(e-e')[\rho] = z_0 f_0[\rho] + z_1 f_1[\rho] + \cdots + z_m f_m[\rho]$ for some $z_0, z_1, \ldots, z_m \in \mathbb{Z}$. A *(modular) equation* is an equation or a modular equation. A *system* of (modular) equations is a set of (modular) equations. A *root* of a system of (modular) equations is a common root of every (modular) equation in the system. Let Φ be a system of (modular) equations and ϕ a (modular) equation, roots of Φ *entail* roots of ϕ (written $\forall \overline{\mathbf{x}}.\Phi \implies \phi$) if all roots of Φ are also roots of ϕ. Given Φ and ϕ, the *root entailment* problem is to decide whether $\forall \overline{\mathbf{x}}.\Phi \implies \phi$.

An *ideal* in $\mathbb{Z}[\overline{\mathbf{x}}]$ generated by $f_0, f_1, \ldots, f_m \in \mathbb{Z}[\overline{\mathbf{x}}]$ is defined by $\langle f_0, f_1, \ldots, f_m\rangle = \{f_0h_0 + f_1h_1 + \cdots + f_mh_m | h_0, h_1, \ldots, h_m \in \mathbb{Z}[\overline{\mathbf{x}}]\}$. If $\langle f_0, f_1, \ldots, f_m\rangle$ and $\langle g_0, g_1, \ldots, g_n\rangle$ are ideals, define their *sum* $\langle f_0, f_1, \ldots, f_m\rangle + \langle g_0, g_1, \ldots, g_n\rangle = \langle f_0, f_1, \ldots, f_m, g_0, g_1, \ldots, g_n\rangle$. For instance, $\langle x\rangle = \{xf | f \in \mathbb{Z}[\overline{\mathbf{x}}]\}$ and $\langle 6\rangle + \langle 10\rangle = \langle 2\rangle$. Given $f \in \mathbb{Z}[\overline{\mathbf{x}}]$ and an ideal I, the *ideal membership problem* is to decide whether $f \in I$.

A *bit-vector* is a bit sequence of a *width* w. A bit-vector denotes an integer between 0 and $2^w - 1$ inclusively using the most-significant-bit-first representation. The SMT QF_BV logic defines bit-vector functions. Assume bv_0 and bv_1 are bit-vectors of width w. The addition $(\mathit{bvadd}\ bv_0\ bv_1)$ and subtraction $(\mathit{bvsub}\ bv_0\ bv_1)$ functions return bit-vectors of width w representing the sum and difference respectively. The multiplication function $(\mathit{bvmul}\ bv_0\ bv_1)$ returns the least significant w bits of the product. The left shift function $(\mathit{bvshl}\ bv_0\ n)$ shifts bv_0 to the left by n bits; the logical right shift function $(\mathit{bvlshr}\ bv_0\ n)$ shifts bv_0 to the right by n bits. The zero extension function $(\mathit{zero\_extend}\ bv_0\ n)$ appends n most significant 0's to bv_0. The extraction function $(\mathit{bvextract}\ h\ l\ bv_0)$ extracts bits indexed h to l from bv_0 $(w > h \geq l \geq 0)$. An SMT QF_BV *expression* is constructed from bit-vector values, variables, and functions. An SMT QF_BV *assertion* is of the form $(\mathit{assert}\ \bot)$, $(\mathit{assert}\ (=\ be\ be'))$, or $(\mathit{assert}\ (\mathit{not}\ (=\ be\ be')))$ with SMT QF_BV expressions be and be'. An SMT QF_BV *query* is a set of SMT QF_BV assertions. A *store* is a mapping from bit-vector variables to bit-vector values. An SMT QF_BV expression *evaluates* to a bit-vector value on a store. An SMT QF_BV assertion $(\mathit{assert}\ (=\ be\ be'))$ is *satisfied* by a store if be and be' evaluate to the same bit-vector value on the store, and otherwise $(\mathit{assert}\ (\mathit{not}\ (=\ be\ be')))$ is satisfied. The SMT QF_BV assertion $(\mathit{assert}\ \bot)$ is never satisfied. An SMT QF_BV query is *satisfiable* if all assertions are satisfied by a store.

3 TOYLANG

We consider a register transfer language called TOYLANG to illustrate algebraic abstraction. For clarity, many programming constructs are removed from TOYLANG. The language nevertheless is sufficiently expressive to implement Montgomery reduction [19], an indispensable algorithm found in real-world cryptographic programs.

3.1 Syntax and Semantics

The syntax of TOYLANG is shown in Figure 1. For simplicity, we assume all numbers are unsigned and all variables are of widths 1 or w. Variables of width 1 are also called *bit* variables. An *atom* is a number or a variable.

$$
\begin{array}{rl}
\mathit{Num}\ n ::= 0 \mid 1 \mid 2 \mid \cdots \quad \mathit{Var}\ c, v ::= a \mid b \mid c \mid \cdots \quad \mathit{Atom}\ a ::= \mathit{Num} \mid \mathit{Var} & \\
\mathit{Inst}\ s ::= & v \leftarrow \text{ADD}\ a_0\ a_1 \mid v \leftarrow \text{ADC}\ a_0\ a_1\ d \mid v \leftarrow \text{SUB}\ a_0\ a_1 \mid \\
& c : v \leftarrow \text{ADDS}\ a_0\ a_1 \mid c : v \leftarrow \text{ADCS}\ a_0\ a_1\ d \mid c : v \leftarrow \text{SUBS}\ a_0\ a_1 \mid \\
& v \leftarrow \text{MUL}\ a_0\ a_1 \mid v \leftarrow \text{SHL}\ a_0\ n \mid \text{ASSUME}\ q \mid \\
& v_H : v_L \leftarrow \text{MULL}\ a_0\ a_1 \mid v \leftarrow \text{SHR}\ a_0\ n \mid \boxed{\text{ASSERT}\ q} \\
\mathit{Exp}\ e, f ::= & n \mid v \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 \times e_1 \mid e_0^n \\
\mathit{MEqn}\ q ::= & e_0 = e_1 \mid e_0 \equiv e_1 \bmod [f_0, f_1, \ldots] \\
\mathit{Program}\ P ::= & s \quad \mid \quad s\ P
\end{array}
$$

Fig. 1: TOYLANG – Syntax

TOYLANG supports several arithmetic instructions: addition (ADD), carrying addition (ADDS), addition-with-carry (ADC), carrying addition-with-carry (ADCS), subtraction (SUB), borrowing subtraction (SUBS), half- (MUL) and full-multiplication (MULL). Moreover, logical left shift (SHL) and logical right shift (SHR) instructions are allowed. In addition to assignments, (modular) equations can be specified in assumption (ASSUME) or assertion (ASSERT) instructions. A program is a sequence of instructions. We assume ASSERT instructions can only appear at the end of programs. They specify a (modular) equation to be verified and thus are emphasized with a framed box.

$$\frac{bv = \mathit{bvadd}\ \llbracket a_0 \rrbracket_\sigma\ \llbracket a_1 \rrbracket_\sigma}{\llparenthesis \sigma, v \leftarrow \text{ADD}\ a_0\ a_1, \sigma[v \mapsto bv] \rrparenthesis} \qquad \frac{bv = \mathit{bvadd}\ (\mathit{bvadd}\ \llbracket a_0 \rrbracket_\sigma\ \llbracket a_1 \rrbracket_\sigma)\ (\mathit{zero\_extend}\ \llbracket d \rrbracket_\sigma\ (w-1))}{\llparenthesis \sigma, v \leftarrow \text{ADC}\ a_0\ a_1\ d, \sigma[v \mapsto bv] \rrparenthesis}$$

$$\frac{bvx = \mathit{bvadd}\ (\mathit{zero\_extend}\ \llbracket a_0 \rrbracket_\sigma\ 1)\ (\mathit{zero\_extend}\ \llbracket a_1 \rrbracket_\sigma\ 1)}{\llparenthesis \sigma, c : v \leftarrow \text{ADDS}\ a_0\ a_1, \sigma[c \mapsto \mathit{bvextract}\ w\ w\ bvx, v \mapsto \mathit{bvextract}\ (w-1)\ 0\ bvx] \rrparenthesis}$$

$$\frac{bvx = \mathit{bvadd}\ (\mathit{bvadd}\ (\mathit{zero\_extend}\ \llbracket a_0 \rrbracket_\sigma\ 1)\ (\mathit{zero\_extend}\ \llbracket a_1 \rrbracket_\sigma\ 1))\ (\mathit{zero\_extend}\ \llbracket d \rrbracket_\sigma\ w))}{\llparenthesis \sigma, c : v \leftarrow \text{ADCS}\ a_0\ a_1\ d, \sigma[c \mapsto \mathit{bvextract}\ w\ w\ bvx, v \mapsto \mathit{bvextract}\ (w-1)\ 0\ bvx] \rrparenthesis}$$

$$\frac{bv = \mathit{bvsub}\ \llbracket a_0 \rrbracket_\sigma\ \llbracket a_1 \rrbracket_\sigma}{\llparenthesis \sigma, v \leftarrow \text{SUB}\ a_0\ a_1, \sigma[v \mapsto bv] \rrparenthesis}$$

$$\frac{bvx = \mathit{bvsub}\ (\mathit{zero\_extend}\ \llbracket a_0 \rrbracket_\sigma\ 1)\ (\mathit{zero\_extend}\ \llbracket a_1 \rrbracket_\sigma\ 1)}{\llparenthesis \sigma, c : v \leftarrow \text{SUBS}\ a_0\ a_1, \sigma[c \mapsto \mathit{bvextract}\ w\ w\ bvx, v \mapsto \mathit{bvextract}\ (w-1)\ 0\ bvx] \rrparenthesis}$$

$$\frac{bv = \mathit{bvshl}\ \llbracket a_0 \rrbracket_\sigma\ n}{\llparenthesis \sigma, v \leftarrow \text{SHL}\ a_0\ n, \sigma[v \mapsto bv] \rrparenthesis} \quad \frac{bv = \mathit{bvlshr}\ \llbracket a_0 \rrbracket_\sigma\ n}{\llparenthesis \sigma, v \leftarrow \text{SHR}\ a_0\ n, \sigma[v \mapsto bv] \rrparenthesis} \quad \frac{bv = \mathit{bvmul}\ \llbracket a_0 \rrbracket_\sigma\ \llbracket a_1 \rrbracket_\sigma}{\llparenthesis \sigma, v \leftarrow \text{MUL}\ a_0\ a_1, \sigma[v \mapsto bv] \rrparenthesis}$$

$$\frac{bv = \mathit{bvmul}\ (\mathit{zero\_extend}\ \llbracket a_0 \rrbracket_\sigma\ w)\ (\mathit{zero\_extend}\ \llbracket a_1 \rrbracket_\sigma\ w)}{\llparenthesis \sigma, v_H : v_L \leftarrow \text{MULL}\ a_0\ a_1, \sigma[v_H \mapsto \mathit{bvextract}\ (2w-1)\ w\ bv, v_L \mapsto \mathit{bvextract}\ (w-1)\ 0\ bv] \rrparenthesis}$$

$$\frac{\sigma \models q}{\llparenthesis \sigma, \text{ASSUME}\ q, \sigma \rrparenthesis} \quad \frac{\sigma \models q}{\llparenthesis \sigma, \boxed{\text{ASSERT}\ q}, \sigma \rrparenthesis} \quad \frac{\sigma \not\models q}{\llparenthesis \sigma, \boxed{\text{ASSERT}\ q}, \mathit{fail} \rrparenthesis} \quad \frac{\llparenthesis \sigma, s, \sigma'' \rrparenthesis \qquad \llparenthesis \sigma'', P, \sigma' \rrparenthesis}{\llparenthesis \sigma, s\ P, \sigma' \rrparenthesis}$$

Fig. 2: TOYLANG – Semantics

Let σ be a store. We write $\sigma[v \mapsto bv]$ for the store obtained by mapping v to the bit-vector bv and other variables u to $\sigma(u)$. $\llbracket v \rrbracket_\sigma$ represents the bit-vector $\sigma(v)$ for any variable v; otherwise, $\llbracket n \rrbracket_\sigma$ is the bit-vector representing the number n of width w.

The semantics of TOYLANG is defined with SMT QF_BV bit-vector functions (Figure 2). In the figure, $\llparenthesis \sigma, s, \sigma' \rrparenthesis$ denotes that the store σ' is obtained after executing the instruction s on the store σ. The addition instruction ADD corresponds to the bit-vector addition function. For the addition with carry instruction, the carry bit is extended with $w - 1$ zeros and added to the sum of the first two operands. The two carrying addition instructions compute the bit-vector sums of width $w + 1$. The most significant bit is stored in the output carry bit. Subtraction instructions are similar; their semantics are defined with the bit-vector subtraction function *bvsub* instead. The semantics of SHL and SHR instructions are defined by corresponding bit-vector functions *bvshl* and *bvlshr* respectively. The semantics of half-multiplication instruction MUL uses the bit-vector multiplication function *bvmul*. For full-multiplication, both operands are extended to width $2w$ before computing their product.

$$\{\!|n|\!\}_\sigma = n \qquad \{\!|v|\!\}_\sigma = \mathsf{toZ}([\![v]\!]_\sigma)$$

$$\{\!|e_0 \pm e_1|\!\}_\sigma = \{\!|e_0|\!\}_\sigma \pm \{\!|e_1|\!\}_\sigma \qquad \{\!|e_0 \times e_1|\!\}_\sigma = \{\!|e_0|\!\}_\sigma \cdot \{\!|e_1|\!\}_\sigma$$

$$\frac{\{\!|e_0|\!\}_\sigma = \{\!|e_1|\!\}_\sigma}{\sigma \models e_0 = e_1} \qquad \frac{\{\!|e_0|\!\}_\sigma - \{\!|e_1|\!\}_\sigma \in \langle \{\!|f_0|\!\}_\sigma, \{\!|f_1|\!\}_\sigma, \ldots, \{\!|f_m|\!\}_\sigma \rangle}{\sigma \models e_0 \equiv e_1 \bmod [f_0, f_1, \ldots, f_m]}$$

Fig. 3: Semantics of (Modular) Equations

The ASSUME instruction filters computations by (modular) equations. Figure 3 defines when a store satisfies a (modular) equation. A number n denotes a non-negative integer. A variable denotes the integer $\mathsf{toZ}([\![v]\!]_\sigma)$ represented by the corresponding bit-vector $[\![v]\!]_\sigma$ in the store. Arithmetic operations denote corresponding integer operations. Particularly, the integer $\{\!|e|\!\}_\sigma$ is exact and not necessarily less than 2^w. Equality denotes integer equality. σ satisfies $e_0 \equiv e_1 \bmod [f_0, f_1, \ldots, f_m]$ if $\{\!|e_0|\!\}_\sigma - \{\!|e_1|\!\}_\sigma$ is in the ideal generated by $\{\!|f_0|\!\}_\sigma, \{\!|f_1|\!\}_\sigma, \ldots, \{\!|f_m|\!\}_\sigma$. The ASSERT instruction checks if the current store satisfies the given (modular) equation. The computation resumes if it *succeeds*. It is an error if the ASSERT instruction *fails*.

```
(* $R = 2^{64}, 0 \leq T < R^2$ *)
(* $N \cdot N' + 1 \equiv 0 \bmod R$ *)
$m \leftarrow ((T \bmod R) \cdot N') \bmod R$
$t \leftarrow (T + m \cdot N)/R$




(* $t \cdot R \equiv T \bmod N$ *)
```

(a) Algorithm

```
(* $T = 2^{64}T_H + T_L$ *)
ASSUME $N \times N' + 1 \equiv 0 \bmod [2^{64}]$
      $m$            $\leftarrow$ MUL $T_L$ $N'$
$mN_H : mN_L$        $\leftarrow$ MULL $m$ $N$
  $carry : t_L$      $\leftarrow$ ADDS $T_L$ $mN_L$
     $c : t_H$       $\leftarrow$ ADCS $T_H$ $mN_H$ $carry$
$\boxed{\text{ASSERT } t_L \equiv 0 \bmod [2^{64}]}$
ASSUME $t_L = 0$
$\boxed{\text{ASSERT } (c \times 2^{64} + t_H) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]}$
```

(b) TOYLANG code

Fig. 4: Simplified Montgomery Reduction

Montgomery reduction algorithm is widely used to compute remainders without division [19]. Figure 4a shows a simplified unsigned Montgomery reduction algorithm.[6] Suppose we want to compute the remainder of a number $0 \leq T < R^2$ modulo N on 64-bit architectures with $R = 2^{64}$. Montgomery reduction algorithm needs another number N' with $NN' + 1 \equiv 0 \bmod R$ as an input. It first computes $m = ((T \bmod R)N') \bmod R$ and then $t = (T + mN)/R$. Observe that the remainder and quotient divided by

[6] The complete algorithm requires range analysis not discussed in this work.

$R = 2^{64}$ amount to bit masking and shifting respectively. Arithmetic division is never used. To prove $tR \equiv T \bmod N$, we first show $T + mN \equiv 0 \bmod R$. Observe $T + mN = T + (((T \bmod R)N') \bmod R)N \equiv T + TN'N \equiv T(1 + N'N) \equiv 0 \bmod R$. Therefore, $T + mN$ is a multiple of R and $t = (T + mN)/R$ is an integer. Hence $tR = T + mN \equiv T \bmod N$.

In the ToyLang implementation (Figure 4b), we represent T by two 64-bit variables T_H and T_L with $T = 2^{64}T_H + T_L$. Hence $T_L = T \bmod 2^{64}$. m is computed by the half-multiplication instruction MUL. The full-multiplication computes the product mN of m and N. The following two addition instructions compute the sum of T and the product mN. After adding T, the least significant 64 bits (t_L) should be zeros. We hence assert $t_L \equiv 0 \bmod [2^{64}]$. If the assertion succeeds, t_L is in fact 0 since it is a 64-bit variable. We thus assume $t_L = 0$. The last assertion checks that the result $2^{64}(2^{64}c + t_H)$ is indeed congruent to T modulo N.

4 Algebraic Abstraction

Algebraic abstraction is a technique to lift computation to an algebraic domain. In the abstract algebraic domain, program instructions are transformed to polynomial equations. Computation in turn is characterized by the roots of systems of polynomial equations. Algebraic abstraction hence allows us to apply algebraic tools from commutative algebra. The abstraction technique requires programs in the static single assignment form. We hence assume input programs are in the static single assignment form.

$$\lceil v \leftarrow \text{ADD}\ a_0\ a_1 \rceil = \{v = a_0 + a_1\} \qquad \lceil v \leftarrow \text{ADC}\ a_0\ a_1\ d \rceil = \{v = a_0 + a_1 + d\}$$
$$\lceil c : v \leftarrow \text{ADDS}\ a_0\ a_1 \rceil = \{c \cdot (c-1) = 0, c \cdot 2^w + v = a_0 + a_1\}$$
$$\lceil c : v \leftarrow \text{ADCS}\ a_0\ a_1\ d \rceil = \{c \cdot (c-1) = 0, c \cdot 2^w + v = a_0 + a_1 + d\}$$
$$\lceil v \leftarrow \text{SUB}\ a_0\ a_1 \rceil = \{v = a_0 - a_1\} \qquad \lceil v \leftarrow \text{MUL}\ a_0\ a_1 \rceil = \{v = a_0 \cdot a_1\}$$
$$\lceil c : v \leftarrow \text{SUBS}\ a_0\ a_1 \rceil = \{c \cdot (c-1) = 0, v = a_0 - a_1 + c \cdot 2^w\}$$
$$\lceil v_H : v_L \leftarrow \text{MULL}\ a_0\ a_1 \rceil = \{v_H \cdot 2^w + v_L = a_0 \cdot a_1\}$$
$$\lceil v \leftarrow \text{SHL}\ a\ n \rceil = \{v = a \cdot 2^n\} \qquad \lceil v \leftarrow \text{SHR}\ a\ n \rceil = \{v \cdot 2^n = a\}$$
$$\lceil \text{ASSUME}\ q \rceil = \{q\} \qquad \lceil s\ P \rceil = \lceil s \rceil \cup \lceil P \rceil$$

Fig. 5: Algebraic Abstraction

Figure 5 lifts ToyLang instructions to polynomial equations. Intuitively, we would like the semantics of each instruction characterized by roots of corresponding polynomial equations. For instance, $v \leftarrow$ ADD $a_0\ a_1$ is lifted to $v = a_0 + a_1$. The ADC instruction is similar. The carrying addition instruction $c : v \leftarrow$ ADDS $a_0\ a_1$ is lifted to two equations: $c \cdot (c-1) = 0$ and $c \cdot 2^w + v = a_0 + a_1$. Since c is a carry, it must be 0 or 1, and hence a root of $c \cdot (c-1) = 0$. The carrying addition-with-carry instruction ADCS is similar, as well as subtraction instructions SUB and SUBS.

The half-multiplication instruction $v \leftarrow$ MUL $a_0\ a_1$ is lifted to $v = a_0 \cdot a_1$; the full-multiplication instruction $v_H : v_L \leftarrow$ MULL $a_0\ a_1$ corresponds to $v_H \cdot 2^w + v_L = a_0 \cdot a_1$.

$$\begin{array}{lll}
\textsc{assume}\ N \times N' + 1 \equiv 0 \bmod [2^{64}] & & N \times N' + 1 \equiv 0 \bmod [2^{64}] \\
m & \leftarrow \textsc{mul}\ T_L\ N' & m = T_L \cdot N', \\
mN_H : mN_L & \leftarrow \textsc{mull}\ m\ N & mN_H \cdot 2^{64} + mN_L = m \cdot N, \\
\mathit{carry} : t_L & \leftarrow \textsc{adds}\ T_L\ mN_L & \begin{array}{c} \mathit{carry} \cdot (\mathit{carry} - 1) = 0, \\ \mathit{carry} \cdot 2^{64} + t_L = T_L + mN_L, \end{array} \\
c : t_H & \leftarrow \textsc{adcs}\ T_H\ mN_H\ \mathit{carry} & \begin{array}{c} c \cdot (c - 1) = 0, \\ c \cdot 2^{64} + t_H = T_H + mN_H + \mathit{carry}, \end{array} \\
\multicolumn{3}{c}{\boxed{\textsc{assert}\ t_L \equiv 0 \bmod [2^{64}]}} \\
\textsc{assume}\ t_L = 0 & & t_L = 0 \\
\multicolumn{3}{c}{\boxed{\textsc{assert}\ (c \times 2^{64} + t_H) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]}}
\end{array}$$

Fig. 6: Abstract Montgomery Reduction

The logical left shift instruction $v \leftarrow \textsc{shl}\ a\ n$ corresponds to $v = a \cdot 2^n$; the logical right shift instruction $v \leftarrow \textsc{shr}\ a\ n$ is lifted to $v \cdot 2^n = a$. The ASSUME q instruction is lifted to the (modular) equation q. All computations thus must satisfy q. A TOYLANG program is lifted to the system of (modular) equations from its instructions. The system of (modular) equations is called the *abstract polynomial program*. Figure 6 shows the abstract polynomial program for the Montgomery reduction program.

4.1 Soundness Conditions

Algebraic abstraction in Figure 5 however is unsound. The TOYLANG semantics is defined over bounded integers of bit width w. Polynomial equations in algebraic abstraction are interpreted over integers. When overflow occurs in TOYLANG instructions, for instance, its computation is not captured by corresponding polynomial equations. Consider the instruction $v \leftarrow \textsc{add}\ 2^{w-1}\ 2^{w-1}$. By the TOYLANG semantics, v has the bit-vector value *bvadd* $[\![2^{w-1}]\!]_\sigma [\![2^{w-1}]\!]_\sigma = 0$ after execution. Clearly, 0 is not a root of the equation $v = 2^{w-1} + 2^{w-1}$. The abstraction is unsound.

In order to check soundness for algebraic abstraction, we define soundness conditions for TOYLANG instructions to ensure that all computations are captured by corresponding polynomial equations. Intuitively, we give an SMT QF_BV query for each instruction in a TOYLANG program such that the query is satisfiable if and only if the computation at the instruction can overflow.

To this end, we first use SMT QF_BV logic to characterize computations in TOYLANG programs. Recall TOYLANG programs are in the static single assignment form. Figure 7 defines an SMT QF_BV query $\lfloor P \rceil$ for any TOYLANG program P. Except the ASSUME instruction, the figure follows the semantics of TOYLANG. For instance, $\lfloor v \leftarrow \textsc{adc}\ a_0\ a_1\ d \rceil$ asserts v equal to the bit-vector sum of a_0 and a_1 with d extended by $w - 1$ zeros in the SMT QF_BV query. Others are similar. It is not hard to see that all computations of a TOYLANG program satisfy the corresponding SMT QF_BV query.

Lemma 1. *Let P be a* TOYLANG *program without* ASSERT *instructions and σ, σ' stores with $(\!|\sigma, P, \sigma'|\!)$. Then the SMT QF_BV query $\lfloor P \rceil$ is satisfied by the store σ'.*

$$\begin{aligned}
\lfloor v \leftarrow \text{ADD}\ a_0\ a_1 \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvadd}\ a_0\ a_1)))\} \\
\lfloor v \leftarrow \text{ADC}\ a_0\ a_1\ d \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvadd}\ (\textit{bvadd}\ a_0\ a_1)\ (\textit{zero\_extend}\ d\ (w-1)))))\} \\
\lfloor c : v \leftarrow \text{ADDS}\ a_0\ a_1 \rceil &= \left\{ \begin{array}{l} (\textit{assert}\ (=\ c\ (\textit{bvextract}\ w\ w\ \textit{bvx}))), \\ (\textit{assert}\ (=\ v\ (\textit{bvextract}\ (w-1)\ 0\ \textit{bvx}))) \end{array} \right\} \\
&\quad \text{where } \textit{bvx} \text{ is } (\textit{bvadd}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
\lfloor c : v \leftarrow \text{ADCS}\ a_0\ a_1\ d \rceil &= \left\{ \begin{array}{l} (\textit{assert}\ (=\ c\ (\textit{bvextract}\ w\ w\ \textit{bvx}))), \\ (\textit{assert}\ (=\ v\ (\textit{bvextract}\ (w-1)\ 0\ \textit{bvx}))) \end{array} \right\} \\
&\quad \text{where } \textit{bvx} \text{ is } (\textit{bvadd}\ (\textit{bvadd}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
&\qquad\qquad\qquad (\textit{zero\_extend}\ d\ w)) \\
\lfloor v \leftarrow \text{SUB}\ a_0\ a_1 \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvsub}\ a_0\ a_1)))\} \\
\lfloor c : v \leftarrow \text{SUBS}\ a_0\ a_1 \rceil &= \left\{ \begin{array}{l} (\textit{assert}\ (=\ c\ (\textit{bvextract}\ w\ w\ \textit{bvx}))), \\ (\textit{assert}\ (=\ v\ (\textit{bvextract}\ (w-1)\ 0\ \textit{bvx}))) \end{array} \right\} \\
&\quad \text{where } \textit{bvx} \text{ is } (\textit{bvsub}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
\lfloor v \leftarrow \text{MUL}\ a_0\ a_1 \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvmul}\ a_0\ a_1)))\} \\
\lfloor v_H : v_L \leftarrow \text{MULL}\ a_0\ a_1 \rceil &= \left\{ \begin{array}{l} (\textit{assert}\ (=\ v_H\ (\textit{bvextract}\ (2w-1)\ w\ \textit{bvx}))), \\ (\textit{assert}\ (=\ v_L\ (\textit{bvextract}\ (w-1)\ 0\ \textit{bvx}))) \end{array} \right\} \\
&\quad \text{where } \textit{bvx} \text{ is } (\textit{bvmul}\ (\textit{zero\_extend}\ a_0\ w)\ (\textit{zero\_extend}\ a_1\ w)) \\
\lfloor v \leftarrow \text{SHL}\ a_0\ n \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvshl}\ a_0\ n)))\} \\
\lfloor v \leftarrow \text{SHR}\ a_0\ n \rceil &= \{(\textit{assert}\ (=\ v\ (\textit{bvlshr}\ a_0\ n)))\} \\
\lfloor \text{ASSUME}\ q \rceil &= \emptyset \\
\lfloor s\ P \rceil &= \lfloor s \rceil \cup \lfloor P \rceil
\end{aligned}$$

Fig. 7: Soundness Conditions I

Our next task is to define SMT QF_BV queries for instructions such that their algebraic abstraction is unsound if and only if the corresponding SMT QF_BV query is satisfiable (Figure 8). The instruction $v \leftarrow \text{ADD}\ a_0\ a_1$ is lifted to $v = a_0 + a_1$. The abstraction is unsound when there is carry. That is, $(\textit{bvextract}\ w\ w\ (\textit{bvadd}\ (\textit{zero\_extend}\ a_0\ 1)$ $(\textit{zero\_extend}\ a_1\ 1)))$ is 1. The instructions ADC and SUB are similar. Algebraic abstraction for the instructions ADDS, ADCS and SUBS is always sound. Their corresponding SMT QF_BV queries are not satisfiable $(\textit{assert}\ \bot)$. For the half-multiplication $v \leftarrow \text{MUL}\ a_0\ a_1$, its abstraction $v = a_0 \cdot a_1$ is unsound when the most significant w bits of the product of a_0 and a_1 are not all zeros. The corresponding SMT QF_BV query is hence $(\textit{assert}\ (\textit{not}\ (=\ 0\ (\textit{bvextract}\ (2w-1)\ w\ \textit{bvx}))))$ where $\textit{bvx}$ is the bit-vector product of a_0 and a_1. The abstraction for full-multiplication instruction is never unsound. For the $v \leftarrow \text{SHL}\ a_0\ n$ instruction, its algebraic abstraction is unsound if the most significant n bits of a_0 are not zeros. The algebraic abstraction of the $v \leftarrow \text{SHR}\ a_0\ n$ instruction is unsound when the least significant n bits of a_0 are not zeros. Relevant bits are obtained by *bvextract* respectively. The abstraction for ASSUME is always sound.

To check soundness of the algebraic abstraction $\lceil s \rceil$ for the instruction s in the TOYLANG program $P\ s$, we apply Lemma 1 to obtain a computation of P through $\lfloor P \rceil$ and check if $\lfloor s \rfloor$ for s is unsatisfiable. We say the soundness condition for the instruction s in the TOYLANG program $P\ s$ *holds* if $\lfloor P\ s \rfloor$ is unsatisfiable. In order to ensure the soundness of the abstract polynomial program $\lceil P \rceil$ for the TOYLANG program P, soundness conditions for all instructions in P must hold. That is, soundness conditions

$$
\begin{aligned}
\lfloor v \leftarrow \textsc{add}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ (=\ 1\ (\textit{bvextract}\ w\ w\ bvx)))\} \\
&\quad \text{where } bvx \text{ is } (\textit{bvadd}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
\lfloor v \leftarrow \textsc{adc}\ a_0\ a_1\ d \rfloor &= \{(\textit{assert}\ (=\ 1\ (\textit{bvextract}\ w\ w\ bvx)))\} \\
&\quad \text{where } bvx \text{ is } (\textit{bvadd}\ (\textit{bvadd}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
&\qquad\qquad\qquad (\textit{zero\_extend}\ d\ w) \\
\lfloor c : v \leftarrow \textsc{adds}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ \bot)\} \\
\lfloor c : v \leftarrow \textsc{adcs}\ a_0\ a_1\ d \rfloor &= \{(\textit{assert}\ \bot)\} \\
\lfloor v \leftarrow \textsc{sub}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ (=\ 1\ (\textit{bvextract}\ w\ w\ bvx)))\} \\
&\quad \text{where } bvx \text{ is } (\textit{bvsub}\ (\textit{zero\_extend}\ a_0\ 1)\ (\textit{zero\_extend}\ a_1\ 1)) \\
\lfloor c : v \leftarrow \textsc{subs}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ \bot)\} \\
\lfloor v \leftarrow \textsc{mul}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ (\textit{not}\ (=\ 0\ (\textit{bvextract}\ (2w-1)\ w\ bvx))))\} \\
&\quad \text{where } bvx \text{ is } (\textit{bvmul}\ (\textit{zero\_extend}\ a_0\ w)\ (\textit{zero\_extend}\ a_1\ w)) \\
\lfloor v_H : v_L \leftarrow \textsc{mull}\ a_0\ a_1 \rfloor &= \{(\textit{assert}\ \bot)\} \\
\lfloor v \leftarrow \textsc{shl}\ a_0\ n \rfloor &= \{(\textit{assert}\ (\textit{not}\ (=\ 0\ (\textit{bvextract}\ (w-1)\ (w-n)\ a_0))))\} \\
\lfloor v \leftarrow \textsc{shr}\ a_0\ n \rfloor &= \{(\textit{assert}\ (\textit{not}\ (=\ 0\ (\textit{bvextract}\ (n-1)\ 0\ a_0))))\} \\
\lfloor \textsc{assume}\ q \rfloor &= \{(\textit{assert}\ \bot)\} \\
\lfloor P\ s \rfloor &= \lfloor P \rceil \cup \lfloor s \rfloor
\end{aligned}
$$

Fig. 8: Soundness Conditions II

for s in all prefixes $P'\ s$ of P must hold. Define the valuation ρ_σ of the store σ by $\rho_\sigma(v) = \mathsf{toZ}(\llbracket v \rrbracket_\sigma)$ for every $v \in \overline{\mathbf{x}}$. The next theorem gives the soundness condition.

Proposition 1 (Soundness). *Let P be a* TOYLANG *program without* ASSERT *instructions and σ, σ' stores with $(\!|\sigma, P, \sigma'|\!)$. $\rho_{\sigma'}$ is a root of the system of (modular) equations $\lceil P \rceil$ if soundness conditions for s in every prefix $P'\ s$ of P hold.*

We say that the soundness condition for P *holds* if soundness conditions for s in all prefixes $P'\ s$ of P hold. Let us take a closer look at the abstract Montgomery reduction program (Figure 6). The half-multiplication instruction $m \leftarrow \textsc{mul}\ T_L\ N'$ is lifted to $m = T_L \cdot N'$. However, the soundness condition for the instruction requires the most significant 64 bits of the product to be zeros (Figure 8). Since T_L is arbitrary, the soundness condition does not hold in general. To obtain a sound algebraic abstraction for Montgomery reduction, we modify the TOYLANG program slightly (Figure 9).

In the revised program, the first full-multiplication instruction is used to compute the least significant 64 bits of the product of T_L and N' (marked by $\surd$). The most significant 64 bits of the product are stored in the variable dc (for *don't care*). Note that the soundness condition of the revised program holds trivially. The algebraic abstraction for the revised Montgomery reduction program is sound by Proposition 1.

4.2 Polynomial Program Verification

Let P be a TOYLANG program without ASSERT instructions. Our goal is to verify $\boxed{P\ \textsc{assert}\ \phi}$ with algebraic abstraction. Consider the system of (modular) equations $\Phi = \lceil P \rceil$. For any stores σ and σ' with $(\!|\sigma, P, \sigma'|\!)$, $\rho_{\sigma'}$ is a root of Φ if the soundness

$$\begin{array}{lrll}
 & \textsc{assume}\ N \times N' + 1 \equiv 0 \bmod [2^{64}] & & N \times N' + 1 \equiv 0 \bmod [2^{64}] \\
\surd & dc : m & \leftarrow \textsc{mull}\ T_L\ N' & dc \cdot 2^{64} + m = T_L \cdot N', \\
 & mN_H : mN_L & \leftarrow \textsc{mull}\ m\ N & mN_H \cdot 2^{64} + mN_L = m \cdot N, \\
 & carry : t_L & \leftarrow \textsc{adds}\ T_L\ mN_L & carry \cdot (carry - 1) = 0, \\
 & & & carry \cdot 2^{64} + t_L = T_L + mN_L, \\
 & c : t_H & \leftarrow \textsc{adcs}\ T_H\ mN_H\ carry & c \cdot (c - 1) = 0, \\
 & & & c \cdot 2^{64} + t_H = T_H + mN_H + carry,
\end{array}$$

$$\boxed{\textsc{assert}\ t_L \equiv 0 \bmod [2^{64}]}$$

$$\textsc{assume}\ t_L = 0 \qquad\qquad t_L = 0$$

$$\boxed{\textsc{assert}\ (c \times 2^{64} + t_H) \times 2^{64} \equiv T_H \times 2^{64} + T_L \bmod [N]}$$

Fig. 9: Abstract Montgomery Reduction (Revised)

condition for P holds by Proposition 1. To verify ASSERT ϕ on σ', we need to check if $\rho_{\sigma'}$ is also a root of the (modular) equation ϕ. That is, we want to show if $\forall \mathbf{x}.\Phi \implies \phi$.

Proposition 2. *Let P be a* TOYLANG *program without* ASSERT *instructions and ϕ a (modular) equation. Suppose the soundness condition for P holds. The assertion in* $\boxed{P\ \textsc{assert}\ \phi}$ *succeeds if* $\forall \mathbf{x}.\lceil P \rceil \implies \phi$.

We extend [14] to check the root entailment problem. Recall that Φ is a system of (modular) equations. We first simplify it to a system of equations. This is best seen by an example. Consider $\forall x\ y\ u\ v. x \equiv y \bmod [3u^2, u+v] \implies 0 = 0$. We have

$$\begin{aligned}
&\forall x\ y\ u\ v. x \equiv y \bmod [3u^2, u+v] \implies 0 = 0 \\
\text{iff }&\forall x\ y\ u\ v. [\exists k_0\ k_1 (x - y = 3u^2 \cdot k_0 + (u+v) \cdot k_1)] \implies 0 = 0 \\
\text{iff }&\forall x\ y\ u\ v\ k_0\ k_1. x - y = 3u^2 \cdot k_0 + (u+v) \cdot k_1 \implies 0 = 0.
\end{aligned}$$

Therefore, it suffices to consider the problem of checking $\forall \mathbf{x}.\Psi \implies \phi$ where Ψ is a system of equations and ϕ is a (modular) equation. We solve the simplified problem by constructing instances of the ideal membership problem.

Let $\Psi = \{e_0 = e_0', e_1 = e_1', \ldots, e_n = e_n'\}$. Consider the ideal $I = \langle e_0 - e_0', e_1 - e_1', \ldots, e_n - e_n' \rangle$ generated by the polynomial equations in Ψ. Suppose the polynomial $e - e' \in I$. We claim $\forall \mathbf{x}.\Psi \implies e = e'$. Indeed, $e - e' = (e_0 - e_0') \cdot h_0 + (e_1 - e_1') \cdot h_1 + \cdots + (e_n - e_n') \cdot h_n$ for some $h_0, h_1, \ldots, h_n \in \mathbb{Z}[\overline{\mathbf{x}}]$ since $e - e' \in I$. For any root ρ of Ψ, $(e_0 - e_0')[\rho] = (e_1 - e_1')[\rho] = \cdots = (e_n - e_n')[\rho] = 0$. Hence $(e - e')[\rho] = ((e_0 - e_0') \cdot h_0)[\rho] + ((e_1 - e_1') \cdot h_1)[\rho] + \cdots + ((e_n - e_n') \cdot h_n)[\rho] = 0$. ρ is also a root of $e - e' = 0$ and thus $\forall \mathbf{x}.\Psi \implies e = e'$.

Now suppose the polynomial $e - e' \in I + \langle f_0, f_1, \ldots, f_m \rangle$. We claim $\forall \mathbf{x}.\Psi \implies e \equiv e' \bmod [f_0, f_1, \ldots, f_m]$. Since $e - e' \in I + \langle f_0, f_1, \ldots, f_m \rangle$, $e - e' = (e_0 - e_0') \cdot h_0 + (e_1 - e_1') \cdot h_1 + \cdots + (e_n - e_n') \cdot h_n + f_0 \cdot k_0 + f_1 \cdot k_1 + \cdots + f_m \cdot k_m$ for some $h_0, h_1, \ldots, h_n, k_0, k_1, \ldots, k_m \in \mathbb{Z}[\overline{\mathbf{x}}]$. For any root ρ of Ψ, $(e - e')[\rho] = ((e_0 - e_0') \cdot h_0)[\rho] + ((e_1 - e_1') \cdot h_1)[\rho] + \cdots + ((e_n - e_n') \cdot h_n)[\rho] + f_0 \cdot k_0[\rho] + f_1 \cdot k_1[\rho] + \cdots + f_m \cdot k_m[\rho] = 0 + f_0[\rho]k_0[\rho] + f_1[\rho]k_1[\rho] + \cdots + f_m[\rho]k_m[\rho]$. We again have $\forall \mathbf{x}.\Psi \implies e \equiv e' \bmod [f_0, f_1, \ldots, f_m]$ as required.

Our discussion is summarized as follows.

$$e = e' \rightsquigarrow \langle e - e' \rangle$$

$$e \equiv e' \bmod [f_0, f_1, \ldots, f_m] \rightsquigarrow \langle e - e' - f_0 \cdot k_0 - f_1 \cdot k_1 - \cdots - f_m \cdot k_m \rangle$$

$$k_0, k_1, \ldots, k_m : \text{fresh variables}$$

$$\frac{}{\emptyset \rightsquigarrow \langle 0 \rangle} \qquad \frac{\phi \rightsquigarrow I \qquad \Phi \rightsquigarrow J}{\{\phi\} \cup \Phi \rightsquigarrow I + J}$$

Fig. 10: Polynomial Programs to Ideals

Proposition 3. *Let P be a* TOYLANG *program without* ASSERT *instructions and I the ideal with $\lceil P \rceil \rightsquigarrow I$ (Figure 10). Then*

1. $\forall \overline{\mathbf{x}}. \lceil P \rceil \implies e = e'$ *if* $e - e' \in I$;
2. $\forall \overline{\mathbf{x}}. \lceil P \rceil \implies e \equiv e' \bmod [f_0, f_1, \ldots, f_m]$ *if* $e - e' \in I + \langle f_0, f_1, \ldots, f_m \rangle$.

In order to verify (modular) equations with algebraic abstraction, Proposition 1 is applied to ensure the soundness of abstraction. Proposition 3 then checks whether (modular) equations indeed are satisfied for abstract polynomial programs. The main theorem summarizes our theoretical developments.

Theorem 1. *Let P be a* TOYLANG *program without* ASSERT *instructions, σ, σ' stores with $(\!|\sigma, P, \sigma'|\!)$ and I the ideal with $\lceil P \rceil \rightsquigarrow I$. If the soundness condition for P holds,*

1. *the assertion in* $\boxed{P \text{ ASSERT } e = e'}$ *succeeds provided* $e - e' \in I$;
2. *the assertion in* $\boxed{P \text{ ASSERT } e \equiv e' \bmod [f_0, f_1, \ldots, f_m]}$ *succeeds provided* $e - e' \in I + \langle f_0, f_1, \ldots, f_m \rangle$.

The ideal membership problem can be solved by computing Gröbner bases for ideals [7]. Many computer algebra systems compute Gröbner bases for ideals with simple commands. For instance, the **`groebner`** command in SINGULAR [13] computes a Gröbner basis for any ideal by a user-specified monomial ordering. The **`reduce`** command then checks if a polynomial belongs to the ideal via its Gröbner basis.

Recall the abstract polynomial program for revised Montgomery reduction in Figure 9. Figure 11a shows the ideal for the abstract polynomial program before ASSUME $t_L = 0$. To verify the two ASSERT instructions, Figures 11b and 11c show the instances of the ideal membership problem corresponding to the two assertions. Observe the ideal $\langle t_L \rangle$ corresponds to ASSUME $t_L = 0$ in Figure 11c. Since the soundness condition for the abstract polynomial program holds trivially (Section 4.1), it remains to check the ideal membership problem. Both instances are verified immediately.

5 Certified Verification

In TOYLANG, we only highlight necessary instructions to verify unsigned Montgomery reduction. For real-world programs performing non-linear computation, more instructions are needed and the signed representation of bit-vectors is also used. In order to

$$I = \left\langle \begin{array}{c} N \cdot N' + 1 - k_0 \cdot 2^{64}, dc \cdot 2^{64} + m - T_L \cdot N', mN_H \cdot 2^{64} + mN_L - m \cdot N, \\ carry \cdot (carry - 1), carry \cdot 2^{64} + t_L - (T_L + mN_L), c \cdot (c - 1), \\ c \cdot 2^{64} + t_H - (T_H + mN_H + carry) \end{array} \right\rangle$$

(a) Ideal I

$$t_L \in I + \langle 2^{64} \rangle$$

(b) $\boxed{\text{ASSERT } t_L \equiv 0 \bmod [2^{64}]}$

$$(c \cdot 2^{64} + t_H) \cdot 2^{64} - (T_H \cdot 2^{64} + T_L) \in I + \langle t_L \rangle + \langle N \rangle$$

(c) $\boxed{\text{ASSERT } (c \times 2^{64} + t_H) \times R \equiv T_H \times 2^{64} + T_L \bmod [N]}$

Fig. 11: Instances of Ideal Membership Problem

verify real-world cryptographic programs, we extend algebraic abstraction with these features found in CRYPTOLINE [9,29]. For such complicated languages, algebraic abstraction can be tedious to implement. Its verification algorithm moreover relies on complex algorithms from computer algebra systems and SMT QF_BV solvers. It is unclear whether these external tools function correctly on given instances. In order to improve the quality of verification results, we have verified algebraic abstraction with the proof assistant COQ, and certified results from external tools with COQ and a verified certificate checker. We briefly describe how to verify our algorithms and certify results from external tools. Please see the technical report [28] for details.

5.1 Verified Abstraction Algorithm

The proof assistant COQ with the SSREFLECT library [4,11] is used to verify our algebraic abstraction technique. We define the TOYLANG syntax as a COQ data type (Figure 1). The COQ-NBITS theory [26] is adopted to formalize the semantics of TOYLANG (Figure 2). The COQ binary integer theory Z is used to formalize the semantics of (modular) equations (Figure 3). We formalize polynomial expressions with integral coefficients by the COQ polynomial expression theory PExpr Z.

To see how our algebraic abstraction algorithm is verified, consider Proposition 2. Let program be the COQ data type for TOYLANG programs and meqn the data type for (modular) equations. We define the predicate algsnd : program $\rightarrow$ Prop for the soundness condition for a given program (Figures 7 and 8). Similarly, we define the function algabs : program $\rightarrow$ seq meqn for our algebraic abstraction algorithm where seq meqn is the COQ data type for sequences of meqn (Figure 5). To write down the formal statement for Proposition 2, it remains to formalize the root entailment. Let exp and valuation be the data types for expressions and valuations respectively. Define the function eval_exp : exp $\rightarrow$ valuation $\rightarrow$ Z which evaluates an expression to an integer on a valuation; and eval_exps : seq exp $\rightarrow$ valuation $\rightarrow$ seq Z evaluates expressions to integers on a valuation. Consider the predicate eval_bexp : meqn $\rightarrow$ valuation $\rightarrow$

Prop defined by

$$\begin{aligned}\text{eval\_bexp (e = e') rho} &:= \text{eval\_exp e rho = eval\_exp e' rho}\\ \text{eval\_bexp (e = e' mod fs) rho} &:= \exists\text{ks, (eval\_exp e rho) - (eval\_exp e' rho) =}\\ &\qquad\qquad\text{zadds (zmuls ks (eval\_exps fs rho))}\end{aligned}$$

where zadds zs := foldl Z.add 0 zs and zmuls xs ys := map2 Z.mul xs ys. The predicate eval_bexp (e = e') rho checks if the expressions e and e' evaluate to the same integer on the valuation rho; eval_bexp (e = e' mod fs) rho checks if the difference of eval_exp e rho and eval_exp e' rho is equal to a linear combination of the integers eval_exps fs rho. The predicate eval_bexp meq rho thus checks if rho is a root of the (modular) equation meq.

We are ready to formalize the root entailment. Consider the predicate entails (Phi : seq meqn) (psi : meqn) : Prop defined by

$$\forall\text{rho}, (\forall\text{phi}, \text{phi} \in \text{Phi} \to \text{eval\_bexp phi rho}) \to \text{eval\_bexp psi rho}.$$

That is, every common root of the system Phi is also a root of psi. The following proposition formalizes Proposition 2 and is proved in COQ.

Proposition 4. *Let P : program be without assert instructions and psi : meqn. If algsnd P and entails (algabs P) psi, then the assertion in* $\boxed{\textit{P assert psi}}$ *succeeds.*

To apply this proposition to a given program P and a (modular) equation psi, one needs to show algsnd P and entails (algabs P) psi in COQ. In principle, both predicates algsnd P and entails (algabs P) psi could be proved manually in COQ. However, it would be impractical even for programs of moderate sizes. To address this problem, we establish these predicates through certificates computed by external tools.

5.2 Verification through Certification

To show algsnd P for an arbitrary program P, we follow the certified verification technique developed in the SMT QF_BV solver COQQFBV [26]. More concretely, we specify our bit-blasting algorithm for soundness conditions in COQ (Figures 7 and 8). The algorithm converts soundness conditions to Boolean formulae in the conjunctive normal form. We then formally verify that soundness conditions hold if and only if the corresponding Boolean formulae are unsatisfiable in COQ. The constructed Boolean formulae are sent to the SAT solver KISSAT [5]. For each Boolean formula, KISSAT checks its satisfiability with a certificate. We then use the verified certificate checker GRATCHK [16] to validate these certificates.

Our next goal is to show entails (algabs P) psi. More generally, we show entails Phi psi with arbitrary Phi : seq meqn and psi : meqn via the COQ polynomial ring theory and the computer algebra system SINGULAR [13]. To this end, we first formulate the root entailment of polynomial expressions in the COQ polynomial ring theory. Recall PExpr Z is the COQ data type for polynomial expressions with integral coefficients. Given integers, the function zpeval : PExpr Z $\to$ seq Z $\to$ Z evaluates a polynomial expression to an integer. We formalize the root entailment of polynomial expressions by the predicate zpentails (Pi : seq (PExpr Z)) (tau : PExpr Z):

$$\forall \text{zs}, (\forall \text{pi}, \text{pi} \in \text{Pi} \rightarrow \text{ zpeval pi zs = 0}) \rightarrow \text{ zpeval tau zs = 0}.$$

We proceed to connect the root entailment of (modular) equations to the root entailment of polynomial expressions. Let the functions zpexprs_of_exprs : seq expr $\rightarrow$ seq (PExpr Z) and zpexprs_of_meqns : seq meqn $\rightarrow$ seq (PExpr Z) convert expressions and (modular) equations to polynomial expressions respectively (Figure 10). When the consequence of root entailment is a modular equation, recall that moduli in the consequence become ideal generators (Proposition 3). To extract moduli from consequences, define zpexpr_of_conseq : meqn $\rightarrow$ PExpr Z $\times$ seq (PExpr Z) by

```
          zpexpr_of_conseq (e = e') := (e - e', [::])
zpexpr_of_conseq (e = e' mod fs) := (e - e', zpexprs_of_exprs fs)
```

The following Coq lemma shows how to check the root entailment of (modular) equations through the root entailment of polynomial expressions:

Lemma 2. $\forall$ *(Phi : seq meqn) (psi : meqn), zpentails (Pi ++ zpexprs_of_meqns Phi) tau implies entails Phi psi where (tau, Pi) = zpexpr_of_conseq psi.*

Note that moduli in the consequence psi are added to the antecedents Phi.

Our last step is to show zpentails (Pi ++ zpexprs_of_meqns Phi) tau. Again, we establish the generalized form zpentails Pi tau for polynomial expressions Pi and a polynomial expression tau. We prove the predicate by showing that tau can be expressed as a combination of expressions in Pi. Consider the predicate validate_zpentails (Xi : seq (PExpr Z)) (Pi : seq (PExpr Z)) (tau : PExpr Z) defined by

```
size Xi = size Pi ∧
ZPeq (ZPnorm tau) (ZPnorm (foldl ZPadd 0 (map2 ZPmul Xi Pi))).
```

The predicate validate_zpentails checks if the Xi and Pi are of the same size. It then normalizes the polynomials tau and foldl ZPadd 0 (map2 ZPmul Xi Pi) using ZPnorm. If normalized polynomials are equal (ZPeq), the predicate is true. In foldl ZPadd 0 (map2 ZPmul Xi Pi), ZPadd and ZPmul are the constructors for polynomial expression addition and multiplication respectively. The expression map2 ZPmul Xi Pi hence returns products of elements in Xi with corresponding elements in Pi. The expression foldl ZPadd 0 (map2 ZPmul Xi Pi) then computes the sum of these products. The predicate validate_zpentails Xi Pi tau therefore checks if tau is equal to a polynomial combination of expressions in Pi. In other words, tau belongs to the ideal generated by Pi. Using Lemma 2, we prove the following variant of Proposition 3 in Coq:

Proposition 5. $\forall$ *Phi psi Xi, validate_zpentails Xi (Pi ++ zpexprs_of_meqns Phi) tau implies entails Phi psi where (tau, Pi) = zpexpr_of_conseq psi.*

The main difference between Propositions 3 and 5 lies in certifiability. There are many ways to establish ideal membership. Proposition 5 asks for witnesses Xi to justify ideal membership explicitly. Most importantly, such Xi need not be constructed manually. They are in fact computed by external tools. Precisely, these polynomial expressions are computed by the **`lift`** command in the computer algebra system SINGULAR [13]. The **`lift`** command computes polynomial expressions representing tau in the ideal generated by Pi ++ zpexprs_of_meqns Phi. After SINGULAR computes these polynomial expressions, we convert them to polynomial expressions Xi in COQ. The predicate validate_zpentails Xi (Pi ++ zpexprs_of_meqns Phi) tau checks if tau is indeed represented by Xi using the COQ polynomial ring theory. If the check succeeds, we obtain entails Phi psi by Proposition 5. Otherwise, the predicate entails Phi psi is not established. Note that SINGULAR need not be trusted. If Xi is computed incorrectly, the check validate_zpentails Xi (Pi ++ zpexprs_of_meqns Phi) tau will fail in COQ. Proposition 5 allows us to show entails Phi psi with certification.

5.3 Optimization

Lots of optimizations are needed and verified to make algebraic abstraction feasible for TOYLANG programs with thousands of instructions. For instance, the static single assignment transformation and program slicing algorithms are both specified and verified in COQ. Furthermore, the bit blasting algorithm is extended significantly to check soundness conditions effectively. For example, the soundness condition for the half-multiplication instruction MUL requires *bvmul* (Figure 8). This could not work well because of complicated non-linear bit-vector computation. To reduce the complexity of overflow checking in half-multiplication, we implement and verify the algorithm from [10]. Last but not least, algebraic abstraction almost surely induces ideals with hundreds of polynomial generators if not thousands. Computing Gröbner bases for such ideals is infeasible. To address this problem, we develop heuristics to reduce the number of generators in ideals through rewriting. Our heuristics are also specified and verified in COQ. These optimizations are essential in our experiments.

6 Evaluation

We have implemented certified algebraic abstraction in the tool COQCRYPTOLINE [1]. COQCRYPTOLINE is built upon OCAML codes extracted from our COQ development. It calls the computer algebra system SINGULAR [13] and certifies answers from the algebraic tool. The certified SMT QF_BV solver COQQFBV [26] is used to verify soundness conditions. We choose two classes of real-world cryptographic programs in experiments. For elliptic curve cryptography, we verify various field or group operations from BITCOIN [27], BORINGSSL [8,12], NSS [20], OPENSSL [23], and PQCRYPTO-SIDH [18]. For post-quantum cryptography, we verify the C reference and optimized Intel avx2 implementations of the Number-Theoretic Transform in the cryptosystem KYBER [6]. Experiments are conducted on an Ubuntu 22.04.1 Linux server with 3.20 GHz 32-core Xeon Gold 6134M and 1TB RAM.

We compare COQCRYPTOLINE with the uncertified CRYPTOLINE [24,9]. Table 1 shows the experimental results. L_{CL} shows the number of instructions. T_{CCL} and T_{CL} give the verification time of COQCRYPTOLINE and CRYPTOLINE in seconds respectively. $\%_{Int}$ shows the percentage of time spent in extracted OCAML programs in COqCryptoLine. $\%_{CAS}$ and $\%_{SMT}$ give the percentages of time spent on Singular and CoqQFBV respectively.

Table 1: Experimental Results on Industrial Cryptographic Programs

$Function$	L_{CL}	$\%_{Int}$	$\%_{SMT}$	$\%_{CAS}$	T_{CCL}	T_{CL}	$Function$	L_{CL}	$\%_{Int}$	$\%_{SMT}$	$\%_{CAS}$	T_{CCL}	T_{CL}
bitcoin/asm/secp256k1_fe_*													
mul_inner	167	0.13	99.52	0.34	91.96	2.41	sqr_inner	151	0.28	99.13	0.59	28.30	1.17
bitcoin/field/secp256k1_fe_*													
mul_inner	132	0.09	98.81	1.11	58.34	1.44	mul_int	6	0.14	95.21	4.65	1.17	0.02
negate	10	0.37	95.60	4.04	0.61	0.02	sqr_inner	119	0.12	98.60	1.28	34.08	0.91
bitcoin/group/													
secp256k1_ge_neg								31	1.82	90.48	7.70	0.24	0.03
secp256k1_gej_double_var.part.14								948	0.53	98.93	0.54	1091.28	25.50
bitcoin/scalar/secp256k1_scalar_*													
mul	918	1.19	98.26	0.54	167.97	6.28	mul_512	338	0.50	98.51	0.98	36.97	2.20
sqr	929	1.49	97.81	0.70	147.07	5.41	sqr_512	349	0.66	98.10	1.23	27.45	3.11
secp256k1_scalar_reduce								104	2.50	91.18	6.32	1.21	0.09
secp256k1_scalar_reduce_512								580	1.62	97.50	0.88	47.83	1.88
boringssl/fiat_curve25519/fe_*													
mul_impl	114	0.04	99.67	0.29	70.85	1.65	sqr_impl	96	0.09	99.38	0.53	25.30	0.75
fe_mul121666								54	1.31	95.61	3.08	0.84	0.07
x25519_scalar_mult_generic[7]								1068	0.27	99.55	0.18	1019.43	279.95
boringssl/fiat_curve25519_x86/fe_*													
mul_impl	375	0.38	99.28	0.34	81.67	1.79	sqr_impl	299	0.52	99.08	0.40	39.89	0.97
fe_mul121666								96	1.96	95.02	3.02	1.07	0.08
x25519_scalar_mult_generic[7]								3287	0.45	99.40	0.15	4454.87	240.00
nss/Hacl_Curve25519_51/													
fmul0	127	0.03	99.67	0.30	136.53	31.11	fmul1	67	0.09	98.85	1.06	12.65	0.26
fsqr0	98	0.03	99.64	0.33	75.10	2.90	fsqr20	196	0.06	99.55	0.38	105.24	3.15
fmul20								238	0.06	99.65	0.29	200.54	35.29
point_add_and_double[7]								1165	0.13	99.65	0.22	2611.51	355.34
point_double								582	0.17	99.49	0.35	975.02	17.06
openssl/curve25519/fe51_*													
mul	111	0.06	99.66	0.28	57.91	1.20	sq	93	0.08	99.34	0.58	23.06	0.69
fe51_mul121666								55	1.27	95.95	2.78	0.70	0.07
x25519_scalar_mult[7]								1042	0.29	99.54	0.17	912.24	281.26
PQCrypto-SIDH/P434/x86_64/													
fpmul434	266	91.74	0.02	8.24	0.39	0.05	fp2mul434	1161	1.10	98.62	0.29	726.40	42.44
PQCrypto-SIDH/P503/arm64/													
fpmul	553	2.43	96.19	1.39	249.24	5.49	fpmul-fixed	554	2.39	95.75	1.86	250.41	5.46
PQClean/kyber/NTT													
PQCLEAN_KYBER512_CLEAN_ntt								6273	4.78	34.21	61.01	1113.92	46.54
PQCLEAN_KYBER768_AVX2_ntt								8975	5.41	83.63	10.96	433.31	29.63

[7] One (out of three) modular polynomial equation in post-conditions fails to certify due to stack overflow.

6.1 Field and Group Operation in Elliptic Curves

In elliptic curve cryptography, a rational point on a curve is represented by field elements from a large finite field. Rational points on the curve form a group. The group operation in turn is computed by operations in the underlying finite field. In BITCOIN, the finite field is $\mathbb{Z}_{p256k1}$ with $p256k1 = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. The underlying field for Curve25519 is $\mathbb{Z}_{p25519}$ with $p25519 = 2^{255} - 19$. PQCRYPTO-SIDH however uses slightly more complicated fields $\mathbb{Z}_{p434}/\langle x^2+1\rangle$ and $\mathbb{Z}_{p503}/\langle x^2+1\rangle$ with $p434 = 2^{216} \cdot 3^{137} - 1$ and $p503 = 2^{250} \cdot 3^{159} - 1$. Field elements in $\mathbb{Z}_{p256k1}$ and $\mathbb{Z}_{p25519}$ are represented by multiple *limbs* of 64-bit numbers. Field multiplication, for instance, is implemented by a number of 64-bit arithmetic instructions. Field elements in $\mathbb{Z}_{p434}/\langle x^2 + 1\rangle$ and $\mathbb{Z}_{p503}/\langle x^2 + 1\rangle$ are of the form $u + vx$ where $u, v \in \mathbb{Z}_{p434}$ or $\mathbb{Z}_{p503}$ and $x^2 = -1$. Two moduli are used to specify multiplication for such fields: $p434$, $x^2 + 1$ for $\mathbb{Z}_{p434}/\langle x^2 + 1\rangle$, and $p503$, $x^2 + 1$ for $\mathbb{Z}_{p503}/\langle x^2 + 1\rangle$. Multiplication of PQCRYPTO-SIDH is easily specified by modular equations with multiple moduli.

COQCRYPTOLINE verifies every field operation with certification within 12.1 minutes. Group operations are implemented by field operations. Their certified verification thus takes more time. The most complicated case x25519_scalar_mult_generic (3287 instructions) from BORINGSSL takes about 1.3 hours.[7] In comparison, CRYPTOLINE verifies the same program in 4 minutes without certification. In almost all cases, a majority of time is spent on COQQFBV. Running time for extracted OCAML programs is negligible. Interestingly, COQCRYPTOLINE finds a bug in the arm64 multiplication code for $\mathbb{Z}_{p503}/\langle x^2 + 1\rangle$ from PQCRYPTO-SIDH. Towards the end of multiplication, the programmer incorrectly stores the register x25 in memory *before* adding a carry. After fixing the bug, COQCRYPTOLINE finishes certified verification in about 5 minutes.

6.2 Number-Theoretic Transform in KYBER

The United States National Institute of Standards and Technology (NIST) is currently determining next-generation post-quantum cryptography (PQC) standards. In July 2022, Crystals-KYBER (or simply KYBER) was announced to be the winner for key establishment mechanisms.

One of the most critical steps in KYBER is modular polynomial multiplication over the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^{256} + 1\rangle$ with $q = 3329$. In $\mathcal{R}_q$, coefficients are elements in the field $\mathbb{Z}_q$. A polynomial in $\mathcal{R}_q$ is obtained by modulo $x^{256} + 1$ and hence has a degree less than 256. Consider $x^{256} \in \mathbb{Z}_q[x]$. Since $x^{256} \equiv -1 \bmod (x^{256} + 1)$, x^{256} is -1 in $\mathcal{R}_q$. Unsurprisingly, polynomial multiplication is one of the most expensive computations in KYBER. An efficient way to multiply polynomials is through a discretized Fast Fourier Transform called the Number-Theoretic Transform (NTT).

Recall the Chinese remainder theorem for integers is but a ring isomorphism between residue systems. For instance, $\mathbb{Z}_{42} \cong \mathbb{Z}_6 \times \mathbb{Z}_7$. For polynomial rings, we have the following ring isomorphism

$$\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2\rangle \cong \mathbb{Z}_q[x]/\langle x^n - \omega\rangle \times \mathbb{Z}_q[x]/\langle x^n + \omega\rangle \qquad (\omega \in \mathbb{Z}_q).$$

Observe that x^n is equal to ω in $\mathbb{Z}_q[x]/\langle x^n - \omega\rangle$ for $x^n \equiv \omega \mod (x^n - \omega)$. Similarly, x^n is equal to $-\omega$ in $\mathbb{Z}_q[x]/\langle x^n + \omega\rangle$. Recall polynomials in $\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2\rangle$ have degrees less than $2n$. We can rewrite any polynomial in $\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2 \rangle$ as $f(x) + g(x)x^n$ where degrees of f and g are both less than n. The polynomial $f(x) + g(x)x^n$ is then equal to $f(x) + \omega g(x)$ in $\mathbb{Z}_q[x]/\langle x^n - \omega \rangle$; and it is equal to $f(x) - \omega g(x)$ in $\mathbb{Z}_q[x]/\langle x^n + \omega \rangle$. NTT computes the following ring isomorphism between $\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2 \rangle$ and $\mathbb{Z}_q[x]/\langle x^n - \omega \rangle \times \mathbb{Z}_q[x]/\langle x^n + \omega \rangle$ by substituting $\pm\omega$ for x^n in $f(x) + g(x)x^n$:

$$f(x) + g(x)x^n \leftrightarrow (f(x) + \omega g(x), f(x) - \omega g(x)) \,. \tag{1}$$

Multiplication in $\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2 \rangle$ can therefore be computed by respective multiplications in $\mathbb{Z}_q[x]/\langle x^n \pm \omega \rangle$ through the isomorphism. That is, a multiplication for polynomials of degrees less than $2n$ (in $\mathbb{Z}_q[x]/\langle x^{2n} - \omega^2 \rangle$) is replaced by two multiplications for polynomials of degrees less than n (in $\mathbb{Z}_q[x]/\langle x^n \pm \omega \rangle$).

In KYBER, ring isomorphisms are applied repeatedly until linear polynomials are obtained. That is, KYBER NTT computes the isomorphism

$$\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^{256} + 1 \rangle \cong \mathbb{Z}_q[x]/\langle x^2 - \zeta_0 \rangle \times \cdots \times \mathbb{Z}_q[x]/\langle x^2 - \zeta_{127} \rangle \tag{2}$$

where ζ_j's are the principal 256-th roots of unity. A polynomial of a degree less than 256 is hence mapped via KYBER NTT to 128 linear polynomials, each modulo a different $x^2 - \zeta_j$. In PQCLEAN [25], a reference C implementation and a hand-optimized Intel avx2 assembly implementation of KYBER NTT are provided. In addition to degree reduction, the two implementations utilize signed Montgomery reduction extensively for efficient multiplication over $\mathbb{Z}_q$. We verify whether the two NTT implementations compute the ring isomorphism correctly.

To specify the correctness requirements of KYBER NTT, one could write down modular equations (1) according to its computation. Each equation would require explicit substitution. Thanks to modular equations with multiple moduli, a more intuitive and mathematical specification based on (2) is also expressible. Let $F = \Sigma_{k=0}^{255} f_k x^k$ denote the input polynomial in $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^{256} + 1 \rangle$ and the coefficients f_k's are input variables with $-q < f_k < q$ $(0 \leq k < 256)$. Let $G_j = g_{j,0} + g_{j,1}x$ be the j-th final output linear polynomial from the implementations. The modular equations

$$F \equiv G_j \bmod [q, x^2 - \zeta_j], \text{ for all } 0 \leq j < 128$$

specify the correctness of the KYBER NTT implementations. Observe that our specification is almost identical to (2). Modular equations with multiple moduli allow cryptographic programmers to express mathematical specification naturally. They greatly improve usability and reduce specification efforts in algebraic abstraction.

COQCRYPTOLINE verifies the C reference implementation in about 18.6 minutes. The highly optimized avx2 implementation is verified in about 7.2 minutes. Observe that each layer of ring isomorphism requires 128 signed Montgomery reductions. KYBER NTT therefore has $7 \times 128 = 896$ Montgomery reductions similar to the running example in Figure 4b. Algebraic abstraction successfully verifies the two KYBER NTT implementations within 20 minutes. In comparison, CRYPTOLINE verifies both NTT implementations in 1 minute without certification.

7 Conclusion

Verification through algebraic abstraction combines both algebraic and bit-accurate analyses. Non-linear computation is analyzed algebraically; soundness conditions are checked with bit-accurate SMT QF_BV solvers. We describe how to verify the technique and certify its results. In the experiments, the hybrid technique successfully verifies non-linear integer computation found in cryptographic programs from elliptic curve and post-quantum cryptography with certification. We plan to explore more applications of algebraic abstraction in programs from post-quantum cryptography in near future.

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. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
3. Bernstein, D.J., Schwabe, P.: gfverif. `http://gfverif.cryptojedi.org` (2015)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and treengeling entering the SAT competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Suda, M.J.M. (eds.) Competition 2020 - Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 50–53. University of Helsinki (2020)
6. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: 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)
7. Buchberger, B., Winkler, F.: Gröbner bases and applications, vol. 17. Cambridge University Press Cambridge (1998)
8. 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)
9. 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)

10. Gok, M., Schulte, M.J., Arnold, M.G.: Integer multipliers with overflow detection. IEEE Transactions on Computers **55**(8), 1062–1066 (August 2006)
11. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. Journal of Formalized Reasoning **3**(2), 95–152 (2010)
12. Google: Boringssl (2021), `https://boringssl.googlesource.com/boringssl/`
13. Greuel, G.M., Pfister, G.: A Singular Introduction to Commutative Algebra. Springer-Verlag (2002)
14. Harrison, J.: Automating elementary number-theoretic proofs using Gröbner bases. In: Pfenning, F. (ed.) International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 4603, pp. 51–66. Springer (2007)
15. Hwang, V., Liu, J., Seiler, G., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**, 718–750 (2022)
16. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 10395, pp. 237–254. Springer (2017)
17. 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)
18. Microsoft Research: PQCrypto-SIDH (2022), `https://github.com/microsoft/PQCrypto-SIDH`
19. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**, 519–521 (1985)
20. Mozilla: Network security services (2021), `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS`
21. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Information Processing Letters **91**, 233–244 (2004)
22. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Leroy, X. (ed.) POPL. pp. 330–341. ACM (2004)
23. OpenSSL: OpenSSL library. `https://github.com/openssl/openssl` (2021)
24. 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)
25. PQClean: The PQClean project. `https://github.com/PQClean/PQClean` (2021)
26. 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. Lecture Notes in Computer Science, Springer (2021)
27. The Bitcoin Developers: Bitcoin source code (2021), `https://github.com/bitcoin/bitcoin`
28. 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), `https://eprint.iacr.org/2022/1116`
29. 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)