Algebraic Linear Analysis for Number Theoretic Transform in Lattice-Based Cryptography

Anonymous Submission

Abstract. The topic of verifying postquantum cryptographic software has never been more pressing than today between the new NIST postquantum cryptosystem standards being finalized and various countries issuing directives to switch to postquantum or at least hybrid cryptography in a decade. One critical issue in verifying lattice-based cryptographic software is range-checking in the finite-field arithmetic assembly code which occurs frequently in highly optimized cryptographic software. For the most part these have been handled by Satisfiability Modulo Theory (SMT) but so far they mostly are restricted to Montgomery arithmetic and 16-bit precision. We add semi-automatic range-check reasoning capability to the CryptoLine toolkit via the Integer Set Library (wrapped via the python package `islpy`) which makes it easier and faster in verifying more arithmetic crypto code, including Barrett and Plantard finite-field arithmetic, and show experimentally that this is viable on production code.

Keywords: Integer Set Library · CryptoLine · Formal Verification · Assembly Code

1 Introduction

1.1 Motivation

Due to the recent issuance of NIST's new Postquantum Standards (FIPS-203–205) which are much more complex than their pre-quantum brethren, the topic of verifying postquantum cryptography, in particular lattice-based cryptography, has again come to the fore.

There have been already efforts to verify lattice-based cryptography. In particular, [8,24] both verified lattice-based crypto programs in different ways. However, these are mostly centered around KEMs and do not cover Dilithium and similar lattice-based Postquantum digital signatures. There are no published articles verifying Dilithium in the literature.

One possible reason for this is that when verifying range properties in the context of arithmetic cryptographic code involving multiplications, it seems that 16-bit multiplications with 32-bit products can be handled moderately well using current SMT technology. However, range checks for 32-bit multiplications with 64-bit products seem to be out of the capabilities of SMT(SAT) solvers. Furthermore, most of the code verified seems to involve Montgomery reductions and multiplications, which are easier to verify in an algebraic manner. Far fewer discussions exist on Barrett multiplications (currently the state of the art for ARM aarch64 code) and Plantard multiplications (state of the art for some cryptosystem-platform combinations, most prominent being Kyber on ARM Cortex-M4).

We conclude that there surely would be interest in (a) verification of the core component (NTT multiplications) of Dilithium, (b) verification for Barrett and Plantard multiplications, and (c) range verification in 32-bit arithmetic involving mulmods.

1.2 Contributions

We introduce an adaptation of the ISL (Integer Set Library, wrapped in python) library into the CryptoLine toolkit. Such usage of an integral set reasoning tool is new as far as we can check, and it handles ranges arising from linear arithmetic relations extremely well. This makes it useful to verify more lattice-based PQC implementations.

As mentioned above, most verification of postquantum arithmetic code restrict themselves to Montgomery mulmod arithmetic in 16 bits. Our ISL-based tool handles both Plantard and Barrett multiplications easily and extends effortlessly to 32-bit arithmetic.

As a result of the new addenda to the CryptoLine toolkit, we are able to verify several optimized Kyber and Dilithium NTT/iNTT and platform combinations, which we exhibit in Section 5. Both the Dilithium (i)NTT Barrett-based implementations and the Kyber (i)NTT Plantard-based implementations had not been verified (in print). All these are highly optimized current state-of-the-art implementations.

1.3 Related Work

There are many other current solutions for verifying cryptographic code that guarantees range properties. Some use Coq (Rocq) [1], and some EasyCrypt [2], such as in the well-known Jasmin code for Kyber [8]. Still others rely on Satisfiability Modulo Theory (SMT) solvers for range checking [18]. As far as we can determine, there are few if any cases wherein non-Montgomery mulmods or 32-bit arithmetic underwent range checks.

A possible reason for this is that it is difficult for SMT (represented by SAT solvers) to handle highly non-linear 32-bit operations (e.g., mulmods) and reason about ranges at the same time. In our own experimentation, it proved possible to handle a limited amount of 16-bit Barrett (and Plantard) mulmods, and 32-bit Montgomery mulmods, but not 32-bit Barrett mulmods. We conjecture that others may have run into the same problem.

There are many prior formal verifications [4–6, 10, 13, 28, 29, 42] of cryptographic programs, mostly in *symmetric cryptography.* Many of these use proof assistants that are non-(semi-)automated. Most of these techniques are not applied in practice to arithmetic-rich, highly optimized, cryptographic software dealing with Public-Key Cryptography. Some methods do produce verified arithmetic cryptographic code but prescribe a way of programming such as Fiat [16] and Jasmin with built-in proofs [8]. We rarely if at all see verification methods that are carried out on hand-optimized code "in the wild". Exceptions are the CryptoLine sequence of works started by [18,39] and [11] (work in progress, using HOL Light [19]) which verify (existing) optimized assembly programs. As can be seen below, we build onto CryptoLine here.

2 Preliminaries

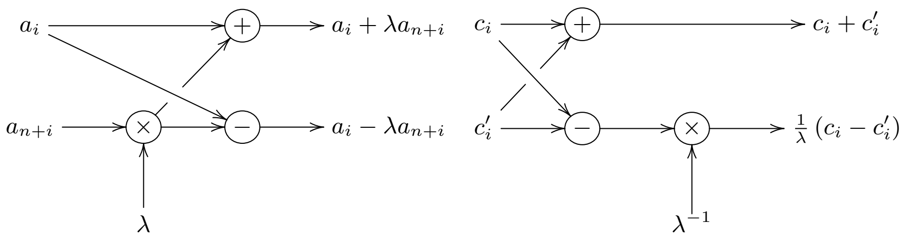
2.1 The Number Theoretic Transform

Kyber and Dilithium [27, 31, 32, 34] each builds a specific variant of the NTT (Number Theoretic Transform) into the specifications for polynomial multiplications. It is therefore vital to understand the mathematics behind NTT multiplications.

In the simplest form of NTTs, using the Cooley-Tukey (CT) formulation, we multiply in $\mathbb{F}_q[x]/\langle x^{2^k}-1\rangle$, for a prime field $\mathbb{F}_q$ with a principal root ζ of order 2^k with $\zeta^{2^{k-1}}=-1$.

The Chinese Remainder Theorem (CRT) applies to the quotient ring $\mathbb{F}_q[x]/\langle x^{2n}-\lambda^2\rangle \cong \mathbb{F}_q[x]/\langle x^n-\lambda\rangle \times \mathbb{F}_q[x]/\langle x^n+\lambda\rangle$ for the following ring isomorphism in one *level* of NTT:

$$\begin{aligned}
\mathbb{F}_q[x]/\langle x^{2n}-\lambda^2\rangle \quad &\longleftrightarrow \quad \mathbb{F}_q[x]/\langle x^n-\lambda\rangle \times \mathbb{F}_q[x]/\langle x^n+\lambda\rangle \\
\sum_{i=0}^{2n-1} a_i x^i \quad &\longrightarrow \quad \left(\sum_{i=0}^{n-1}(a_i+\lambda a_{n+i})x^i, \sum_{i=0}^{n-1}(a_i-\lambda a_{n+i})x^i\right) \\
\sum_{i=0}^{n-1}\frac{1}{2}(c_i+c_i')x^i+\sum_{i=0}^{n-1}\frac{1}{2\lambda}(c_i-c_i')x^{n+i} \quad &\longleftarrow \quad \left(\sum_{i=0}^{n-1}c_i x^i, \sum_{i=0}^{n-1}c_i' x^i\right)
\end{aligned}$$



(a) Cooley–Tukey (CT) Butterfly (b) Gentleman–Sande (GS) Butterfly

Figure 1: Butterflies in NTT

A one-level isomorphism is computed by butterflies. The mapping from $\mathbb{F}_q[x]/\langle x^{2n} - \lambda^2\rangle$ to $\mathbb{F}_q[x]/\langle x^n - \lambda\rangle \times \mathbb{F}_q[x]/\langle x^n + \lambda\rangle$ computes a product and followed by addition and subtraction. This is called a Cooley–Tukey (CT) butterfly (Figure 1a). Its inverse mapping computes a sum and a difference, followed by multiplication. This is called a Gentleman–Sande (GS) butterfly (Figure 1b). The constants λ and λ^{-1} are called *twiddles*.

For a positive integer $n = \sum_{i=0}^{k-1} n_i 2^i < 2^k$, where $n_i \in \{0, 1\}$, we may write $\mathrm{brv}_k(n) = \sum_{i=0}^{k-1} n_{k-1-i} 2^i$, the "length-$k$ bit-reversal of n", then apply the CRT repeatedly to get

$$
\begin{aligned}
& \mathbb{F}_q[x]/\langle x^{2^k} - 1\rangle \cong \mathbb{F}_q[x]/\langle x^{2^{k-1}} - 1\rangle \times \mathbb{F}_q[x]/\langle x^{2^{k-1}} + 1\rangle \\
\cong\ & \mathbb{F}_q[x]/\langle x^{2^{k-2}} - 1\rangle \times \mathbb{F}_q[x]/\langle x^{2^{k-2}} + 1\rangle \times \mathbb{F}_q[x]/\langle x^{2^{k-2}} - \zeta^{2^{k-2}}\rangle \times \mathbb{F}_q[x]/\langle x^{2^{k-2}} + \zeta^{2^{k-2}}\rangle \\
\cong\ & \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-2}} - \zeta^{\overbrace{0\cdots0_b}^{k}}\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-2}} - \zeta^{1\overbrace{0\cdots0_b}^{k-1}}\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-2}} - \zeta^{01\overbrace{0\cdots0_b}^{k-2}}\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-2}} - \zeta^{11\overbrace{0\cdots0_b}^{k-2}}\rangle} \\
\cong\ & \prod_{i=0}^{3} \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-2}} - \zeta^{\mathrm{brv}_k(i)}\rangle} \cong \cdots \cong \prod_{i=0}^{2^\ell-1} \frac{\mathbb{F}_q[x]}{\langle x^{2^{k-\ell}} - \zeta^{\mathrm{brv}_k(i)}\rangle} \cong \cdots \cong \prod_{i=0}^{2^k-1} \frac{\mathbb{F}_q[x]}{\langle x - \zeta^{\mathrm{brv}_k(i)}\rangle}
\end{aligned}
$$

If $k > \ell$, we do not end at $\mathbb{F}_q[x]$ modulo a linear polynomial (i.e., copies of $\mathbb{F}_q$) and we call that an *incomplete* NTT. For Kyber and Dilithium, we started with $\mathbb{F}_q[x]/\langle x^{256} + 1\rangle$, and the "negacyclic" transform can be considered half of an NTT starting from $\mathbb{F}_q[x]/\langle x^{512} - 1\rangle$. So Kyber has an incomplete negacyclic NTT and Dilithium a complete negacyclic NTT.

Note that "twisting" $f(x) = a_0 + a_1 x + \cdots + a_i x^i + \cdots + a_{n-1} x^{n-1}$ via scaling variables linearly with $x = cy$ gives $a_0 + c a_1 y + \cdots + c^i a_i y^i + \cdots + c^{n-1} a_{n-1} y^{n-1}$, or $a_i \mapsto c^i a_i$. Twisting is also used for controlling the magnitude of coefficients: Just before coefficients potentially overflow, twisting eliminates that danger at little cost.

2.2 Lattice-Based Cryptography

We first describe the Crystals KEM and digital signature pair and then describe the main types of arithmetic used. Note each uses an NTT that is constant across parameter sets.

2.2.1 Kyber

Kyber or ML-KEM [32,34] is a NIST standard lattice-based Key Encapsulation Mechanism (KEM) based on the Module Learning With Errors (M-LWE) problem using an $\ell \times \ell$ matrix in the polynomial ring $R_q = \mathbb{F}_q[x]/\langle x^n + 1\rangle$, with $q = 3329$ and $n = 256$. The Kyber KEM is Hofheinz–Hövelmanns–Kiltz transformed [21] from a CPA-secure Public-Key Encryption (PKE) described in [32, 34]. Time-critical operations are one $(\ell \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplication (`MatrixVectoMul`), plus zero, one, or two `MatrixVectorMul` of $\ell \times 1$ inner products of polynomials (`InnerProd`) for keygen, encapsulation, and decapsulation respectively. The specifications explicitly enforce all multiplications to be via (incomplete) NTTs. The public matrix A is sampled in (incomplete) NTT domain by expanding a seed using `SHAKE128` [30]. Kyber's 7-level incomplete negacyclic NTT is $\frac{\mathbb{F}_q[x]}{\langle x^{2^8}+1\rangle} \cong \prod_{i=0}^{127} \frac{\mathbb{F}_q[x]}{\langle x^2 - \zeta^{\mathrm{brv}_8(128+i)}\rangle}$.

2.2.2 Dilithium

Dilithium or ML-DSA [27, 31] is a NIST standard digital signature scheme based on the M-SIS (Module Small Integer Solutions) and M-LWE problems, using a $k \times \ell$ matrix of polynomials in the ring $R_q = \mathbb{F}_q[x]/\langle x^{256}+1\rangle$ with $q = 2^{23} - 2^{13} + 1 = 8380417$.

For a full description see [27, 31]. The core operation of key generation, signature generation, and signature verification is the $(k \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplications (`MatrixVectorMul`). In signature generation, this operation is executed repeatedly in a rejection-sampling loop. Like Kyber, Dilithium builds an NTT into the specification, in that A is sampled "in NTT domain" using `SHAKE256` [30]. Dilithium's 8-level complete negacyclic NTT is $\frac{\mathbb{F}_q[x]}{\langle x^{2^8}+1\rangle} \cong \prod_{i=0}^{255} \frac{\mathbb{F}_q[x]}{\langle x - \zeta^{\mathrm{brv}_9(i+256)}\rangle}$.

2.2.3 (Signed) Montgomery multiplication or Hensel division [35, 36]

This ingenious variant of Peter Montgomery's method is initially due to Gregor Seiler as follows: Given any X and a suitable power of two R, we compute $q' = q^{-1} \bmod R$, and now can compute $XR^{-1} \bmod q$ by first computing $\ell = Xq' \bmod R$, then because $R|(X - \ell q)$, we have $XR^{-1} \equiv (X - \ell q)R^{-1} \equiv X_h - [\ell q]_h \pmod q$, where "high half" $[\bullet]_h = \lfloor \bullet / R \rfloor$.

This computation improves on a traditional Montgomery reduction in microarchitectures with a "high-half" product, especially a high-half-product-with-accumulation: Further, with b known, we can compute $ab \equiv [a \cdot B]_h - [q \cdot [a \cdot B']_l]_h \pmod q$ with $2 \times \text{high} + 1 \times \text{low}$ mults using precomputed $B = bR \bmod {}^{\pm} q$, $B' = Bq' \bmod {}^{\pm} R$. (Note: $[xy]_l = xy \bmod {}^{\pm} R$.)

2.2.4 Barrett multiplication [12]

Let $[\![\cdot]\!]$ be a function from the reals to the integers such that $|x - [\![x]\!]| \le 1$, then we say that $[\![\cdot]\!]$ is an *integer approximation* and $a \bmod^{[\![\cdot]\!]} b$ is defined to be $a - [\![a/b]\!]b$. When $[\![\cdot]\!]_0$, $[\![\cdot]\!]_1$ are integer approximations. We can compute a representative of $ab \bmod q$ via

$$ab - Lq \equiv ab \pmod q, \quad \text{where } L = \left[\!\!\left[\frac{a \left[\!\left[\frac{bR}{q} \right]\!\right]_0}{R} \right]\!\!\right]_1 .$$

The only question is whether the resulting range is useful, in particular, whether it falls into the data width. [12] showed that

$$ab - \left[\!\!\left[\frac{a \left[\!\left[\frac{bR}{q} \right]\!\right]_0}{R} \right]\!\!\right]_1 q = \frac{a \left(bR \bmod^{[\![\cdot]\!]_0} q\right) + \left(a \left(bR \bmod^{[\![\cdot]\!]_0} q\right)(-q^{-1}) \bmod^{[\![\cdot]\!]_1} R\right) q}{R}.$$

This means $\left| ab - \left[\!\!\left[\frac{a \left[\!\left[\frac{bR}{q} \right]\!\right]_0}{R} \right]\!\!\right]_1 q \right| \le \frac{|a| \left|\bmod^{[\![\cdot]\!]_0} q\right| + \left|\bmod^{[\![\cdot]\!]_1} R\right| q}{R}$, where $\left|\bmod^{[\![\cdot]\!]} X\right|$ means the maximal $\left|a \bmod^{[\![\cdot]\!]} X\right|$ for the integer approximation $[\![\cdot]\!]$. For uses of Barrett multiplication in instruction sets like the Neon, $[\![x]\!]_0 = [\![x]\!]_1 = 2\lfloor x/2 \rceil$ ("round to even"), and here [12] showed that if $|a| \le R/2$, then the result is between $\pm q$. *When the result is always within a signed word, we need not compute the higher half of either ab or Lq at all.*

Since $\hat{b} = \left[\!\!\left[\frac{BR}{q}\right]\!\!\right]_0$ *is precomputed, we only need to compute one higher-half* ($a\hat{b}$) *in addition to the two lower-half products* $ab + L(-q)$, *one of them with accumulation.*

2.2.5 Signed Plantard reduction and multiplication (formulated as in [23])

Thomas Plantard's reduction [33] was introduced into cryptographic NTTs in [9, 22] and provides the state-of-the-art signed 16-bit modular arithmetic on ARM Cortex-M4.

Let $[\![]\!]_1$, $[\![]\!]_2$, $[\![]\!]_3$ be integer approximations, q, $R > 1$ be coprime integers, $\tilde{R}$ be a factor of R, and B be a positive integer. If for all integers z of absolute value $\leq B$, we have

$$\begin{aligned}&\frac{z + \left(z \cdot (-q^{-1}) \mod^{[\![]\!]_1} R\right) q}{R} \\ = \quad &\left[\!\!\left[\frac{z + \left(z \cdot (-q^{-1}) \mod^{[\![]\!]_1} R\right) q - \left(z + \left(z \cdot (-q^{-1}) \mod^{[\![]\!]_1} R \mod^{[\![]\!]_2} \tilde{R}\right) q\right)}{R}\right]\!\!\right]_3\end{aligned}$$

then Plantard reduction computes a representative of $zR^{-1} \mod q$ as

$$\left[\!\!\left[\frac{\left[\!\!\left[\frac{z \cdot (-q^{-1}) \mod^{[\![]\!]_1} R}{\tilde{R}}\right]\!\!\right]_2 q}{R/\tilde{R}}\right]\!\!\right]_3 .$$

Usually $\tilde{R} = R/\tilde{R} = 2^{16}$, occasionally 2^{32}; $[\![\cdot]\!]_1 = \lfloor \cdot \rceil$ (so $\bmod^{[\![\cdot]\!]_1} = \bmod^{\pm}$), $[\![\cdot]\!]_2 = \lfloor \cdot \rfloor$ (so $\bmod^{[\![\cdot]\!]_2} =$ standard mod), $[\![\cdot]\!]_3 = \lfloor \cdot + r \rfloor$, with $r = \frac{1}{2}$, or $\alpha q/(R/\tilde{R})$ — usually for a suitably $\alpha > 0$ with $2\alpha q < R/\tilde{R}$ (in [22], $\alpha = 8$). For *Plantard Multiplication* $z = ab$ with a fixed b, we compute $ab \bmod^{\pm} q$ as the Plantard reduction of $a \cdot (bR \bmod^{\pm} q)$; in other words, with a precomputed $\hat{b} = (bR \bmod^{\pm} q)\left(-q^{-1} \bmod^{\pm} R\right) \bmod^{\pm} R$ compute

$$\left\lfloor \frac{\left\lfloor \frac{a\hat{b} \bmod^{\pm} R}{\tilde{R}} \right\rfloor q + 8q}{R/\tilde{R}} \right\rfloor \left[=_{\text{in [22]}} \text{bit 16–31 of } \left(\left(\left(\text{bit 16–31 of } a\hat{b}\right)_{\text{as } \texttt{sint16}}\right) q + 8q\right)_{\text{as } \texttt{sint16}}\right].$$

Plantard multiplication is uniquely tight, and we get a canonical $ab \mod^{\pm} q$ between $\pm\frac{q}{2}$ (if q odd). However, *it requires a higher-half multiplication in addition to a* <u>*middle word multiplication*</u> *of a double-word and a single-word integer. This latter operation can be simulated by one higher-half and one lower-half multiplication.*

2.3 Program Specifications

We will use the formalism in [17, 20] to specify intended program behaviors. Let P be a program, ϕ and ψ are predicates about program variables. A *Hoare triple* is of the form $\{\phi\}P\{\psi\}$. Given a Hoare triple $\{\phi\}P\{\psi\}$, the program P is expected to behave as follows. Starting from any state where program variables satisfy the *pre-condition* ϕ, the program P must end in a state where program variables satisfy the *post-condition* ψ. If this is indeed the case, we say the triple $\{\phi\}P\{\psi\}$ is *valid*. Observe that a Hoare triple is valid if the program satisfies the post-condition on *all* inputs satisfying the pre-condition.

Note that a program is correct only with respect to its specification in this formalism. In this work, we establish the correctness of 6 assembly implementations of NTTs. Each implementation will be a program. Pre- and post-conditions specify the isomorphisms between input and output polynomials. Moreover, coefficient ranges are crucial to program correctness. They appear in specifications of assembly implementations of NTTs as well.

2.4 Integer Set Library

Many polytope libraries are available. Most of them however use native machine numbers and hence are of a fixed finite precision. Since cryptographic programs perform multiprecision computations, typical polytope libraries are not useful. Among libraries manipulating polytopes with exact integers, we have tested the Z3 SMT solver (Z3), the Parma Polyhedra Library (PPL), and the Integer Set Library (ISL). ISL is found to be most the efficient for the analysis of cryptographic programs.

Integer Set Library (ISL) is an open-sourced C library for manipulating relations over exact integers bounded by linear constraints. It supports all standard set operations such as intersection, union, projection, and emptiness check [40, 41]. Among others, the library has been used for program analysis such as loop optimization in GCC and LLVM.

In ISL, a *space* defines the (named) dimension of an integer space. An ISL set is an integer set in an ISL space. An ISL *set* is a union of basic sets. An ISL *basic set* in turn is a conjunction of affine constraints over integers or a projection of a basic set. An ISL *affine constraint* is of the form

$$c_0 + c_1 D_1 + c_2 D_2 + \cdots c_n D_n \geq 0$$

where $c_i \in \mathbb{Z}$ and D_i are dimension names for all i.

For instance, consider an ISL space with dimensions X and Y. Define the ISL basic set

$$bset = \{(X, Y) | \begin{array}{ccccccccccc} X & - & 2Y & \geq & 0 & \wedge & -X & + & 2Y & \geq & 0 & \wedge \\ 99 & - & X & \geq & 0 & \wedge & -1 & + & X & \geq & 0 & \end{array} \}$$

Then $bset$ represents the set $\{(X, Y) | 0 < X < 100 \text{ and } X = 2Y\}$. If the dimension Y is projected out of *bset*, we obtain an ISL basic set comprising all even integers between 0 and 100. Although one can construct an ISL basic set for all even integers between 0 and 100 by these steps, ISL actually provides a function to convert a string to ISL basic sets. The basic set of even integers between 0 and 100 can be obtained by the following string:

```
{ [X] : exists (Y : X = 2Y and 1 <= X and X <= 99) }
```

In addition to set construction, it is easy to check emptiness of the set `bset` in ISLPY by calling `bset.is_empty()`.

3 Formal Verification with CryptoLine

3.1 CryptoLine Overview

CRYPTOLINE is an automatic formal verification toolkit for cryptographic programs. To verify a cryptographic program with CRYPTOLINE, a formal program model is needed. A program model specifies how the cryptographic program executes. Verifiers use the CRYPTOLINE modeling language to construct such a program model. The CRYPTOLINE language is based on assembly languages and thus most suitable for cryptographic assembly programs. After a program model is constructed, verifiers specify what the cryptographic program is intended to compute. For instance, it may compute the field multiplication operation over a large finite field. Given a program model and its functional specification, the CRYPTOLINE toolkit tries to prove the model conforms to the specification for all inputs automatically. CRYPTOLINE may fail to finish in a reasonable time. Verifiers can annotate the program model with lemmas as hints. CRYPTOLINE will also prove annotated lemmas and use them to speed up verification.

Consider the following 32-bit ARM Cortex-M4 code for Montgomery reduction:

```
smulbt m, T, QQ
smlabb t, QQ, m, T
```

The **smulbt** `m, T, QQ` instruction multiplies the bottom 16 bits of the register `T` with the top 16 bits of the register `QQ`, and stores the 32-bit product in the register `m`. The **smlabb** `t, QQ, m, T` instruction first multiplies the bottom 16 bits of the registers `QQ` and `m`. It then adds the 32-bit product with the 32-bit register `T` and stores the sum in the 32-bit register `t`. If the bottom 16 bits of `QQ` contains a modulus q and the top 16 bits of `QQ` contains the negation of the inverse of q modulo 2^{16}, then the register `t` is $Tq^{-1}q + T \equiv T \cdot (q^{-1}q + 1) \equiv T \cdot 0 (\text{mod } 2^{16})$ and has bottom 16 bits all zeroes.

To verify the ARM Cortex-M4 code, we construct the following CryptoLine model:

```
(* smulbt m, T, QQ *)
spl T_t T_b T 16;
mull m_t m_b T_b QQ_t;
(* smlabb t, QQ, m, T *)
mulj tmp QQ_b m_b;
add t tmp T;
```

The 32-bit register `T` is modeled by a 32-bit CryptoLine variable `T`. The bottom and top 16 bits of the register `QQ` are modeled by 16-bit variables $\mathtt{QQ}_\mathtt{b}$ and $\mathtt{QQ}_\mathtt{t}$ respectively. The CryptoLine **spl** $\mathtt{T}_\mathtt{t}$ $\mathtt{T}_\mathtt{b}$ `T 16` instruction splits the 32-bit variable `T` into two 16-bit variables $\mathtt{T}_\mathtt{t}$ (top) and $\mathtt{T}_\mathtt{b}$ (bottom). The **mull** $\mathtt{m}_\mathtt{t}$ $\mathtt{m}_\mathtt{b}$ $\mathtt{T}_\mathtt{b}$ $\mathtt{QQ}_\mathtt{t}$ computes the 32-bit product of $\mathtt{T}_\mathtt{b}$ and $\mathtt{QQ}_\mathtt{t}$, and stores the bottom and top 16 bits of the product in $\mathtt{m}_\mathtt{b}$ and $\mathtt{m}_\mathtt{t}$ respectivey. **mulj** `tmp` $\mathtt{QQ}_\mathtt{b}$ $\mathtt{m}_\mathtt{b}$ on the other hand stores the 32-bit product of $\mathtt{QQ}_\mathtt{b}$ and $\mathtt{m}_\mathtt{b}$ in the variable `tmp`. Finally, **add** `t tmp T` puts the sum of the 32-bit variables `tmp` and `T` in the variable `t`. One can see the model construction is mostly straightforward. Using the formal semantics for CryptoLine in Coq [38], one could also prove the correctness of model construction with respect to another Coq model for ARM Cortex-M4 programs.

We are ready to give the pre-condition for the program model formally. Recall that the variables $\mathtt{QQ}_\mathtt{b}$ and $\mathtt{QQ}_\mathtt{t}$ contain a modulus q and the negation of its inverse modulo 2^{16}. Formally, we assume

$$R = 2^{16} \wedge \mathtt{QQ}_\mathtt{b} \cdot \mathtt{QQ}_\mathtt{t} + 1 \equiv 0 \mod R \tag{1}$$

Moreover, the number $\mathtt{QQ}_\mathtt{b}$ and the input `T` are not arbitrary. They must be in the proper ranges to prevent overflow. Concretely, we need

$$\mathtt{QQ}_\mathtt{b} < 2^{14} \wedge -\lfloor \mathtt{QQ}_\mathtt{b} \cdot R/2 \rfloor \leq \mathtt{T} \leq \lfloor \mathtt{QQ}_\mathtt{b} \cdot R/2 \rfloor \tag{2}$$

The pre-condition of our small program model is therefore (1) $\wedge$ (2). At the end of the program, we wish to show the output `t` is congruent to `T` modulo $\mathtt{QQ}_\mathtt{b}$ and congruent to 0 modulo R. That is,

$$\mathtt{t} \equiv \mathtt{T} \mod \mathtt{QQ}_\mathtt{b} \wedge \mathtt{t} \equiv 0 \mod R \tag{3}$$

Additionally, we wish to show the top 16-bit of output is between $\pm 2^{16}$ times the modulus $\mathtt{QQ}_\mathtt{b}$. Precisely, we want to show

$$-R \cdot \mathtt{QQ}_\mathtt{b} \leq \mathtt{t} \leq R \cdot \mathtt{QQ}_\mathtt{b} \tag{4}$$

After specifying the pre-condition (1) $\wedge$ (2) and the post-condition (3) $\wedge$ (4), we use CryptoLine to prove whether our program model computes the output `t` correctly for all inputs $\mathtt{QQ}_\mathtt{b}$, $\mathtt{QQ}_\mathtt{t}$, and `T` under our assumptions. CryptoLine verifies it in seconds.

In this example, CryptoLine verifies the program model without further annotations. If more hints were needed, we would state them as assertions. CryptoLine will prove annotated assertions automatically. Once an assertion is proven, it can be assumed as a hint to verify post-conditions.

3.2 Algebraic Abstraction

Conventional program verification techniques such as SMT solvers do not work well for cryptographic programs for two reasons. First, cryptographic programs often perform non-linear computations but typical programs do not. Verification of non-linear computation hence has very limited support in program verification. Second, cryptographic program verification requires bit-accurate techniques for large integers but typical programs need not. Machine integers suffice for most computation. Overflow thus can be overlooked at first and verified by interval arithmetic later. Non-linear and bit-accurate analyses for large integers are missing in conventional program verification techniques.

CryptoLine employs two verification techniques to address these issues. For bit-accurate analysis, CryptoLine uses an SMT QFBV solver to verify simple linear computations. SMT QFBV solvers essentially translate bounded arithmetic computation to Boolean circuits via *bit blasting*. SAT solvers are then invoked to verify Boolean circuits. Since the computation in the program is verified through bit blasting, the technique is clearly bit-accurate. SMT QFBV solvers have been used in program verification and testing. Despite their popularity, bit-accurate SMT QFBV solvers fail to verify most cryptographic programs satisfactorily. The Montgomery reduction program in the last section, for instance, cannot be verified by the most advanced SMT QFBV solver within a week. This is perhaps unsurprising. If SMT QFBV solvers could verify arbitrary non-linear computation, the RSA1024 factoring challenge would have been resolved by now.

Algebraic abstraction is the distinct technique employed by CryptoLine in order to verify non-linear computation [37]. Roughly, algebraic abstraction transforms a cryptographic program to a system of multivariate polynomial equations such that each program trace corresponds to a solution to the system of equations. To verify an equality about program variables, it suffices to check whether all solutions to the system of equations are also solutions to the equality. Most importantly, such solutions can be checked by algebraic techniques. Since non-linear computation is verified algebraically, CryptoLine performs especially well for cryptographic programs.

Going back to the Montgomery reduction example, CryptoLine transforms the program into the following system of polynomial equations:

$$\begin{array}{rcll}
R & = & 2^{16} & \text{pre-condition} \\
\mathtt{QQ_b} \cdot \mathtt{QQ_t} + 1 & \equiv & 0 \mod R & \text{pre-condition} \\
2^{16}\mathtt{T_t} + \mathtt{T_b} & = & \mathtt{T} & \mathbf{spl}\ \mathtt{T_b}\ \mathtt{T_t}\ \mathtt{T}\ \mathtt{16} \\
2^{16}\mathtt{m_t} + \mathtt{m_b} & = & \mathtt{T_b} \cdot \mathtt{QQ_t} & \mathbf{mull}\ \mathtt{m_t}\ \mathtt{m_b}\ \mathtt{T_b}\ \mathtt{QQ_t} \\
\mathtt{tmp} & = & \mathtt{QQ_b} \cdot \mathtt{m_b} & \mathbf{mulj}\ \mathtt{tmp}\ \mathtt{QQ_b}\ \mathtt{m_b} \\
\mathtt{t} & = & \mathtt{tmp} + \mathtt{T} & \mathbf{add}\ \mathtt{t}\ \mathtt{tmp}\ \mathtt{T}
\end{array}$$

The first two equations are from the pre-condition (1). For each CryptoLine instruction, there is a corresponding polynomial equation. Assume no overflow occurs in the **add** `t tmp T` instruction. It is seen that all program traces are solutions to the system of equations. In order to prove whether the post-condition (3) holds for all program traces, it suffices to check whether all solutions to the system of equations are also solutions to $\mathtt{t} \equiv \mathtt{T} \mod \mathtt{QQ_b}$ and $\mathtt{t} \equiv 0 \mod R$.

Note that program traces with overflow will not be solutions to the system of equations. Algebraic abstraction therefore is sound when no overflow occurs. Note also that algebraic abstraction is only for equational reasoning. Since overflow detection and properties such as the post-condition (4) require range analysis, equational reasoning is not applicable. Range properties were verified by the bit-accurate technique with an SMT QFBV solver in CryptoLine [18].

3.3 Algebraic Linear Analysis

In addition to Montgomery reduction, Barrett and Plantard multiplication can also be implemented very efficiently on architectures with rounding instructions [12,23]. Although rounding to integers can be checked by complex equational reasoning through algebraic abstraction [12], it is best verified by bit-accurate analysis. Consequently, SMT QFBV solvers appear to be suitable for verifying cryptographic programs using Barrett or Plantard multiplication. However, bit-accurate SMT QFBV solvers are not very scalable even for cryptographic programs with linear computation. Using an SMT QFBV solver, the PQClean ARM aarch64 Dilithium NTT using Barrett multiplication cannot be verified by CryptoLine in a week (Section 5). A more effective technique is needed.

We aim to develop a more scalable verification technique for NTT implementations using various rounding instructions. As an example, consider the ARM aarch64 instruction `sqrdmulh Vd, Vn, Vm` used in the PQClean ARM aarch64 implementation of Dilithium NTT [26]. The instruction computes the signed products of corresponding elements in `Vn` and `Vm`, doubles the products, and stores the most significant half of the saturated results in `Vd` after rounding. It is modeled by 5 CryptoLine instructions:

```
(* sqrdmulh Vd, Vn, Vm *)
mulj %Pnm %Vn %Vm;              (* product *)
shl %Pnm2 %Pnm [1, 1, 1, 1];    (* double *)
spl %H33 %dc0 %Pnm2 31;         (* get top 33 bits *)
add %R33 %H33 [1, 1, 1, 1];     (* rounding *)
spl %Vd %dc1 %R33 1;            (* get top 32 bits *)
```

In CryptoLine, vector variable names start with the percentage sign (`%`). In this example, each vector variable has 4 signed 32-bit values. The `mulj` instruction computes 4 64-bit signed products of corresponding elements in `%Vn` and `%Vm`. The `shl` instruction shifts all elements to the left by 1 bit. The `spl %H33 %dc %Pnm2 31` instruction splits each element in `%Pnm2` into top 33- and bottom 31-bit values. The 4 top 33-bit values are put in `%H33`. The `add %R33 %H33 [1, 1, 1, 1]` instruction rounds the least significant bit. Finally, the 4 top 32-bit rounded values are stored in `%Vd` by the last instruction. Note that saturation is not modeled here. The model is correct only when no overflow occurs during the execution of `shl` and `add` instructions. Indeed, the `sqrdmulh` instruction is used to implement Barrett multiplication. If saturation occurs, the implementation is incorrect.

To develop an efficient linear analysis technique, we need a feature in NTT computation. Recall that an NTT butterfly only requires addition and multiplication with constants. The computation of NTTs is therefore linear. In the context of algebraic abstractions, it means all equalities are linear. Concretely, let us consider the system of polynomial equations corresponding to `sqrdmulh Vd, Vn, Vm`:

$$
\begin{array}{rcll}
 & & & \texttt{(* sqrdmulh Vd, Vn, Vm *)} \\
\texttt{Pnm[i]} & = & \texttt{Vn[i]} \cdot \texttt{Vm[i]} & \texttt{mulj \%Pnm \%Vn \%Vm} \\
\texttt{Pnm2[i]} & = & \texttt{Pnm[i]} \cdot 2 & \texttt{shl \%Pnm2 \%Pnm [1, 1, 1, 1]} \\
2^{31}\ \texttt{H33[i]} + \texttt{dc0[i]} & = & \texttt{Pnm2[i]} & \texttt{spl \%H33 \%dc0 \%Pnm2 31} \\
\texttt{R33[i]} & = & \texttt{H33[i]} + 1 & \texttt{add \%R33 \%H33 [1, 1, 1, 1]} \\
2\ \texttt{Vd[i]} + \texttt{dc1[i]} & = & \texttt{R33[i]} & \texttt{spl \%Vd \%dc1 \%R33 1}
\end{array}
$$

In Barrett multiplication, the vector register `Vm` contains constants $\hat{\lambda} = [\![\frac{\lambda R}{q}]\!]$ where λ is a twiddle factor, $R = 2^{16}$ and $q = 8380417$. After constant substitution, we obtain a system of *linear* equations. Solving linear equations is much easier than solving general polynomial equations. More efficient techniques are applicable for analysis. Solving the above linear equations nevertheless is insufficient. Consider the last linear equation:

$$2\ \texttt{Vd[i]} \quad + \quad \texttt{dc1[i]} \quad = \quad \texttt{R33[i]}$$

Suppose `R33[0]` is the 33-bit value 4. The instruction **`spl %Vd %dc1 %R33 1`** splits 4 into top 32 and bottom 1 bit, sets `Vd[0]` and `dc1[0]` to 2 and 0 respectively. We have 2 `Vd[0]` + `dc1[0]` = `R33[0]` as expected. Nevertheless, more solutions are possible for the linear equation. For instance, `Vd[0]` = 0 and `dc1[0]` = 4 is another solution to 2 `Vd[0]` + `dc1[0]` = `R33[0]` even though it does not correspond to any program trace. Such spurious solutions do not reflect rounding in **`sqrdmulh`**. Yet they need to satisfy post-conditions in algebraic abstraction. Verification may fail due to spurious solutions.

A simple way to address this problem is to remove spurious solutions. Consider the following linear constraints for the instruction **`spl %Vd %dc1 %R33 1`**:

$$\begin{array}{rcccl} 2\,\texttt{Vd[i]} & + & \texttt{dc1[i]} & = & \texttt{R33[i]} \\ -2^{31} & \leq & \texttt{Vd[i]} & < & 2^{31} \\ 0 & \leq & \texttt{dc1[i]} & < & 2^{1} \end{array}$$

Recall that `R33[i]` is a 33-bit signed value. The additional linear inequalities make the solution to the linear equation unique. No spurious solution is possible. Barrett multiplication can be verified by solving linear constraints derived from CryptoLine programs. Since our new technique allows both equalities and inequalities in linear constraints, it is called *algebraic linear analysis* to differentiate from existing equational reasoning in algebraic abstractions.

Finally, note that constants in cryptographic programs can easily exceed 32- or 64-bit machine integers. Typical linear constraint libraries such as lp_solve or SCIP [3, 14] can induce overflow and give incorrect verification results for cryptographic programs. For verification, it is necessary to use linear constraint libraries with exact integers. For instance, the Integer Set Library uses the GNU Multiple Precision Arithmetic Library GMP to solve linear constraints (Section 2.4). This is essential for algebraic linear analysis of cryptographic programs.

3.4 Algebraic Soundness Checking

Recall that the absence of overflow is required to capture all program traces in algebraic abstraction. Since equational reasoning is unsuitable for range analysis, overflow detection for algebraic abstraction still requires bit-accurate analysis (Section 3.2). Overflow detection however can be formulated as linear constraints easily. Can we apply algebraic linear analysis to detecting overflow and get rid of bit-accurate analysis entirely?

Applying algebraic linear analysis to overflow detection is slightly more complicated. The problem is as follows. Overflow detection is necessary for the soundness of algebraic abstraction and hence algebraic linear analysis. How can algebraic linear analysis be applied to overflow detection without securing soundness? Should the absence of overflow not be checked *before* applying algebraic linear analysis? Could it be circular reasoning?

The answer is NO. It is sound to check overflow through algebraic linear analysis. To see it, consider the program P; **`add`** `a b c`. Suppose `a` is a signed 16-bit variable. The **`add`** instruction is transformed to `a` = `b` + `c` in algebraic abstraction. To ensure all traces are captured by the equation, the absence of overflow is checked by the linear constraint $-32768 \leq \texttt{b} + \texttt{c} < 32768$. We proceed by induction on the length of P.

If P is the empty program, it suffices to check $-32768 \leq \texttt{b} + \texttt{c} < 32768$. This is clearly a linear constraint. If P is not empty, we know how to detect overflow by linear constraints for each instruction in P by induction. If no overflow can occur for all instructions in P, we apply algebraic linear analysis and transform P to a system of linear constraints Π. All program traces of P are hence captured in solutions to Π. Now consider two systems of linear constraints: $\Pi_0 = \Pi \cup \{-32768 > \texttt{b} + \texttt{c}\}$ and $\Pi_1 = \Pi \cup \{\texttt{b} + \texttt{c} \geq 32768\}$. If Π_0 or Π_1 has a solution, then overflow can occur while executing **`add`** `a b c`. If neither has a solution, there cannot be overflow for all traces of P; **`add`** `a b c`. In any case, overflow detection is formulated as linear constraints. Other instructions are checked similarly.

Informally, the argument says that algebraic linear analysis for P suffices to detect overflow for the **add** instruction. If overflow cannot occur for the **add** instruction, algebraic linear analysis is then applied to P; **add** `a b c`. Since overflow detection for the **add** instruction only depends on P but not P; **add** `a b c`, there is no circle. Applying algebraic linear analysis to overflow detection is therefore sound. We call it *algebraic soundness checking* to differentiate from conventional soundness checking by bit-accurate techniques.

3.5 Multitrack Verification

CryptoLine supports compositional reasoning using the **cut** instruction. With compositional reasoning, the correctness reasoning of a long program can be decomposed into the correctness of two shorter programs. For example, consider a Hoare triple $\{\phi\}P_0; P_1\{\psi\}$. If we can find a *mid-condition* ρ such that both $\{\phi\}P_0\{\rho\}$ and $\{\rho\}P_1\{\psi\}$ are valid, then we conclude the validity of $\{\phi\}P_0; P_1\{\psi\}$. Such mid-conditions are specified using **cut** instructions in CryptoLine.

A problem of using **cut** instructions is that a program cannot be decomposed in different ways at the same time. For example, consider a program $P_0; P_1; P_2$ with a precondition ϕ and a postcondition $\psi_0 \wedge \psi_1$. The verification of ψ_0 requires a mid-condition ρ_0 between P_1 and P_2, which decomposes into $\{\phi\}P_0; P_1\{\rho_0\}$ and $\{\rho_0\}P_2\{\psi_0\}$. On the other hand, the verification of ψ_1 requires a mid-condition ρ_1 right after P_0. The validity of $\{\phi\}P_0; P_1; P_2\{\psi_1\}$ is therefore established by the validity of $\{\phi\}P_0\{\rho_1\}$ and $\{\rho_1\}P_1; P_2\{\psi_1\}$. Recall that we wish to establish the validity of $\{\phi\}P_0; P_1; P_2\{\psi_0 \wedge \psi_1\}$. How do we decompose it?

The natural and only way is to divide the program into three parts. $\{\phi\}P_0\{\rho_1\}$, $\{\rho_1\}P_1\{\rho_0\}$, and $\{\rho_0\}P_2\{\psi_0 \wedge \psi_1\}$. But it would not do. To establish the post-condition ρ_0 in $\{\rho_1\}P_1\{\rho_0\}$, information about P_0 may be necessary despite $\{\phi\}P_0; P_1\{\rho_0\}$ is valid. Such information nonetheless may not appear in ρ_1. Similarly, the post-condition ψ_1 in $\{\rho_0\}P_2\{\psi_1\}$ may not be established because it does not necessarily follow from $\{\rho_1\}P_1; P_2\{\psi_1\}$.

To resolve this issue, we propose multitrack verification and implement it in CryptoLine. With the multitrack feature, an annotation (including pre-conditions, mid-conditions, and post-conditions) can be placed on certain tracks. The verification is then carried out by tracks. This allows a program to be decomposed by different ways in different tracks. Take the previous example for demonstration. The pre-condition ϕ can be placed on track 0 and track 1. The mid-condition ρ_0 and post-condition ψ_0 are both placed on track 0. The mid-condition ρ_1, and post-condition ψ_1 are put on track 1. To verify track 0, CryptoLine only considers $\{\phi\}P_0; P_1; P_2\{\psi_0\}$ with mid-condition ρ_0 between P_1 and P_2. This Hoare triple is then decomposed into $\{\phi\}P_0; P_1\{\rho_0\}$ and $\{\rho_0\}P_2\{\psi_0\}$, which can be proved successfully. To verify track 1, CryptoLine only considers $\{\phi\}P_0; P_1; P_2\{\psi_1\}$ with the mid-condition ρ_1 between P_0 and P_1. The Hoare triple for track 1 is then decomposed into $\{\phi\}P_0\{\rho_1\}$ and $\{\rho_1\}P_1; P_2\{\psi_1\}$, which can be verified separately.

4 Case Studies

To illustrate the generality of algebraic linear analysis in NTT verification, we discuss 6 NTT implementations for Dilithium and Kyber on Intel AVX2, ARM aarch64 and Cortex-M4. We explain how NTTs are implemented on different architectures and their CryptoLine specifications.

4.1 Dilithium

The Dilithium specification uses NTT for polynomial multiplications in the ring $R_q = \mathbb{F}_q[x]/\langle x^{256}+1\rangle$ with $q = 8380417$ [31]. The PQClean project provides Intel AVX2 and ARM aarch64 assembly implementations of NTT and inverse NTT [26]. Observe that an element in $\mathbb{F}_q$ where $q = 2^{23} - 2^{13} + 1 < 2^{32}$. A field element hence can be stored in a 32-bit word. These assembly implementations are verified by CryptoLine using algebraic linear analysis. We explain how they are verified in this subsection.

4.1.1 Intel AVX2

The PQClean Intel AVX2 Dilithium NTT implementation performs 8 levels of CT butterflies for an input polynomial $f(x) = \sum_{i=0}^{255} a_i x^i \in \mathbb{F}_q[x]/\langle x^{256}+1\rangle$. In the implementation, 4 32-bit coefficients of a polynomial are packed into a 256-bit vector register. Due to the number of available vector registers, CT butterflies are performed by 4 groups of 64 32-bit coefficients. All the coefficients in one group are loaded into 8 256-bit vector registers in a CT butterfly. The implementation first transforms the input polynomial to 4 63-degree polynomials through levels 0 and 1 of CT butterflies in 4 groups. All the coefficients of the polynomials are stored back to memory. The implementation then performs levels 2 to 7 of CT butterflies similarly by groups, in which one 63-degree polynomial is transformed to 64 constant polynomials.

The PQClean Intel AVX2 Dilithium NTT uses Montgomery multiplication (Section 2.2.3). Consider the following fragment from the implementation:

```
vpmuldq %ymm1,%ymm8,%ymm13
vpmuldq %ymm2,%ymm8,%ymm8
vpmuldq %ymm0,%ymm13,%ymm13
```

The `ymm8` register contains 8 32-bit polynomial coefficients a_i $(0 \le i < 8)$. Let $R = 2^{32}$. The `ymm2` and `ymm1` registers each contains 8 twiddles $B_i = \lambda_i R \bmod q$ and 8 pre-computed values $B'_i = B_i q^{-1} \bmod R$ $(0 \le i < 8)$ respectively. The **`vpmuldq`** `%ymm1, %ymm8, %ymm13` instruction computes the products of the 4 corresponding 32-bit values with even indices in `ymm8` and `ymm1`, and stores the 4 64-bit products in `ymm13`. Hence the `ymm13` register contains $a_i B'_i$ $(i = 0, 2, 4, 6)$. Similarly, the `ymm8` register contains $a_i B_i$ after executing **`vpmuldq`** `%ymm2, %ymm8, %ymm8` $(i = 0, 2, 4, 6)$. Since `ymm0` contains 8 copies of q, `ymm13` contains $q(a_i B'_i \bmod R)$ after **`vpmuldq`** `%ymm0, %ymm13, %ymm13`. Consider the 4 64-bit differences of the values in `ymm8` and `ymm13`. By Montgomery multiplication, the top 32 bits of the differences are $a_i \lambda_i \bmod q$ and the bottom 32 bits are all zeroes for $i = 0, 2, 4, 6$. The products $a_i \lambda_i \bmod q$ for odd indices are computed similarly.

We verify the PQClean AVX2 Dilithium NTT implementation using CryptoLine with the input polynomial $f(x) = \sum_{i=0}^{255} a_i x^i \in \mathbb{F}_q[x]/\langle x^{256}+1\rangle$ and the following pre-condition

$$-q < a_i < q \text{ for } 0 \le i < 256.$$

CryptoLine verifies the ranges of 256 output coefficients c_i are between $-9q$ and $9q$ for $0 \le i < 256$. Moreover, the following post-conditions are also verified $(\zeta = 1753)$

$$f(x) \equiv c_i \bmod [q, x - \zeta^{\mathrm{brv}_9(256+i)}] \text{ for } 0 \le i < 256.$$

The PQClean Intel AVX2 implementation for Dilithium inverse NTT is similar. All the coefficients are arranged into 4 groups. Levels 7 to 2 of GS butterflies are performed for each group with results stored back to memory. It is then followed by levels 1 to 0 of GS butterflies. We also use CryptoLine to verify Intel AVX2 implementation for Dilithium inverse NTT. Assume the input coefficients c_i are between $-q$ and q for $0 \le i < 256$. We view these input coefficients correspond to a polynomial $f(x) \in \mathbb{F}_q/\langle x^{256}+1\rangle$. That is, we have 256 additional pre-conditions

$$f(x) \equiv c_i \mod [q, x - \zeta^{\mathrm{brv}_9(256+i)}] \text{ for } 0 \leq j < 256.$$

CryptoLine verifies that the output coefficients a_i of the inverse NTT are between $-q$ and q for $0 \leq i < 256$. Moreover, the polynomial $F(x) = \sum_{i=0}^{255} a_i x^i$ formed by the output coefficients satisfies the post-condition

$$F(x) \equiv 2^{32} f(x) \mod [q, x^{256}+1].$$

4.1.2 ARM aarch64

In the PQClean ARM aarch64 implementation, a 128-bit register contains 4 32-bit words. The optimized implementation uses 2 groups of 12 128-bit registers for butterflies. Each group performs 16 butterflies. In each group, 4 registers are for Barrett multiplication (Section 2.2.4); the other 8 registers contain polynomial coefficients. In Dilithium, each NTT level has 128 CT butterflies. Eight groups are therefore needed for an NTT level. The implementation interleaves every two groups of butterflies.

Consider the following fragment from the optimized implementation:

```
mul v16.4s, v30.4s, v23.4s
sqrdmulh v30.4s, v30.4s, v22.4s
mls v16.4s, v30.4s, v4.s[0]
```

The 128-bit registers `v30` and `v23` contain 4 polynomial coefficients a_i and 4 NTT twiddle factors λ_i for $0 \leq i < 4$ respectively. After the **`mul`** instruction, the register `v16` contains the 4 32-bit half products of $a_i\lambda_i$ for $0 \leq i < 4$.

Let $R = 2^{32}$. The register `v22` is initialized to 4 constants $\left[\!\left[\frac{\lambda_i R}{2q}\right]\!\right]_0$ with $0 \leq i < 4$. Recall the **`sqrdmulh`** computes double of the product of the two source operands `v30` and `v22`. The factor 2 in the denominator is added to compensate for the doubling in the **`sqrdmulh`** instruction. After executing the instruction, the register `v30` has 4 32-bit values

$$\left[\!\left[\frac{a_i \left[\!\left[\frac{\lambda_i R}{q}\right]\!\right]_0}{R}\right]\!\right]_1 \text{ for } 0 \leq i < 4.$$

The **`mls`** instruction first computes the 64-bit product of `v30` and `v4.s[0]`, subtracts the product from `v16`, and stores the difference in `v16`. Since `v4.s[0]` contains the value q, `v16` contains the following values after executing the instruction:

$$a_i\lambda_i - q\left[\!\left[\frac{a_i \left[\!\left[\frac{\lambda_i R}{q}\right]\!\right]_0}{R}\right]\!\right]_1 \text{ for } 0 \leq i < 4.$$

That is, `v16` contains the values $a_i\lambda_i \bmod q$ for $0 \leq i < 4$ by Barrett multiplication.

Using CryptoLine, we verify the PQClean ARM aarch64 implementation for the Dilithium NTT. Assume the 256 coefficients of the input polynomial are between $-\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$. The implementation outputs 256 values between $-\lfloor 8.5q \rfloor$ and $\lceil 8.5q \rceil$. Moreover, let $f(x)$ denote the input function, c_i the output values, and $\zeta = 1753$. CryptoLine verifies the following 256 post-conditions

$$f(x) \equiv c_i \bmod [q, x - \zeta^{\mathrm{brv}_9(256+i)}] \text{ for } 0 \leq i < 256.$$

PQClean ARM aarch64 implementation for Dilithium inverse NTT is similar. It also uses 2 groups of 12 128-bit registers. For inverse NTT, Barrett multiplication is also employed in GS butterfly. We use CryptoLine to verify the ARM aarch64 implementation for Dilithium inverse NTT as well. Assume the input coefficients c_i are between $-q$ and q and $f(x) \equiv c_i \bmod [q, x - \zeta^{\mathrm{brv}_9(256+i)}]$ for $0 \le i < 256$. CryptoLine shows that coefficients a_i of the output function are between $-\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$. Moreover, the output function $F(x)$ is 2^{32} times the function $f(x)$. Precisely, we have

$$F(x) = \sum_{i=0}^{255} a_i x^i \equiv 2^{32} f(x) \bmod [q, x^{256} + 1].$$

4.2 Kyber

The Kyber specification requires NTT for multiplication in the polynomial ring $R_q = \mathbb{F}_q[x]/\langle x^{256} + 1 \rangle$ with $q = 3329$ [32]. Its field element can hence be stored in a 16-bit word. We discuss Intel AVX2 and ARM aarch64 assembly implementations of NTT and inverse NTT from the PQClean project [26] and two ARM Cortex-M4 implementations from the IPA [22] and pqm4 [25] projects.

4.2.1 Intel AVX2

The PQClean Intel AVX2 implementation for Kyber NTT was first verified in [24]. Later, a variant was verified in [7]. The optimized implementation transforms a 255-degree polynomial in $\mathbb{F}_q[x]/\langle x^{256} + 1 \rangle$ to 128 linear polynomials through 7 levels of Kyber NTT. At each level, 128 butterflies are needed for 256 polynomial coefficients.

The PQClean Intel AVX2 implementation stores 16 16-bit polynomial coefficients in a 256-bit register. The optimized implementation performs 64 butterflies with 12 256-bit vector registers in parallel: 8 256-bit vector registers are for 128 polynomial coefficients and 4 256-bit vector registers for Montgomery multiplication (Section 2.2.3). The computation of 64 parallel butterflies repeats twice to perform 128 butterflies at each level. The Kyber AVX2 implementation uses similar instructions as the PQClean Dilithium AVX2 implementation but with different word sizes. See Section 4.1.1 for details.

Assume the 256 coefficients of the input polynomial all start between $-q$ and q. CryptoLine verifies the coefficients of 128 linear output polynomials are between $-8q$ and $8q$. The following 128 modular equations are verified ($\zeta = 17$)

$$f(x) \equiv c_i + d_i x \bmod [q, x^2 - \zeta^{\mathrm{brv}_8(128+i)}] \text{ for } 0 \le i < 128 \tag{5}$$

where $f(x)$ is the input polynomial and $c_i + d_i x$ are the output polynomials.

The PQClean Intel AVX2 implementation for Kyber inverse NTT is similar. 128 coefficients are computed in parallel at each level. Assume the coefficients of 128 linear input polynomials $c_i + d_i x$ are between $-q$ and q for $0 \le i < 128$. They moreover represents a polynomial $f(x) \in \mathbb{F}_q/\langle x^{256} + 1 \rangle$ such that

$$f(x) \equiv c_i + d_i x \bmod [q, x^2 - \zeta^{\mathrm{brv}_8(128+i)}] \text{ for } 0 \le i < 128. \tag{6}$$

CryptoLine verifies the 256 coefficients a_i of output polynomial are between -31625 and 31625. Moreover, the output polynomial $F(x)$ and the polynomial $f(x)$ satisfy

$$F(x) = \sum_{i=0}^{255} a_i x^i \equiv 2^{16} f(x) \bmod [q, x^{256} + 1]. \tag{7}$$

4.2.2 ARM aarch64

Different from the Intel AVX2 implementation, Barrett multiplication is employed in the PQClean ARM aarch64 implementation of Kyber NTT. In the optimized implementation, each 128-bit register stores 8 coefficients. As in the Dilithium ARM aarch64 implementation, similar instructions but different word sizes are used to implement Barrett multiplication. Please consult Section 4.1.2 for details.

The PQClean ARM aarch64 implementation for Kyber NTT also uses 2 groups of 12 128-bit registers for butterflies. Four of them are for Barrett multiplication and the others are for polynomial coefficients. Each group hence computes 32 butterflies. A level of Kyber NTT has 128 CT butterflies and requires 4 groups of computation. The implementation computes a level of Kyber NTT by interleaving the 2 register groups.

The optimized implementation moreover divides Kyber NTT into two phases. The top phase transforms the input polynomial to 32 polynomials of degree 7 through 5 levels of Kyber NTT. The bottom phase then transforms 32 polynomials of degree 7 to 128 linear polynomials. Recall an ARM aarch64 128-bit register can store 8 polynomial coefficients. After the top phase, coefficients of a 7-degree polynomial can be loaded in a 128-bit register. It is easier to schedule 128-bit registers in the bottom phase.

Assume the 256 input polynomial coefficients are between $-\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$. Our verification shows all coefficients of 128 linear output polynomials are between $-q$ and q for the PQClean ARM aarch64 implementation of Kyber NTT. Moreover, the same post-condition in (5) is verified.

The PQClean ARM aarch64 implementation for Kyber inverse NTT is similar. Two groups of 12 128-bit registers are used to compute GS butterflies. It also divides 7 levels of computation into two. The bottom phase transforms 128 linear input polynomials to 32 polynomials of degree 7; the top phase transforms these 32 polynomials to the output polynomial of degree 255. Assume all coefficients of 128 linear input polynomials $c_i + d_i x$ are between $-q$ and q and they represent a polynomial $f(x)$ such that (6) holds. CryptoLine verifies that coefficients a_i of the output polynomial $F(x)$ are between $-q$ and q. Moreover, $F(x)$ is congruent to 2^{16} times the polynomial $f(x)$ in $\mathbb{F}_q[x]/\langle x^{256}+1\rangle$. That is, the post-condition (7) is verified.

4.2.3 ARM Cortex-M4

ARM Cortex-M4 is a 32-bit architecture. We verify two ARM Cortex-M4 implementations for Kyber NTT. One implemented CT and GS butterflies with Montgomery multiplication and was verified in [24]; the other uses Plantard multiplication [22] and is yet to be verified.

Montgomery Multiplication. This implementation uses specialized 32-bit instructions to optimize butterfly computation. Specifically, ARM Cotex-m4 supports 16-bit operations within the 32-bit architecture. For example, **smulbb**, **smultb**, and **smulbt** are multiplication instructions that compute 32-bit products of signed 16-bit integers from bottom and top halves of 32-bit registers. They are used for efficient multiplication in Kyber NTT.

For example, the following fragment computes a product with Montgomery multiplication (Section 2.2.3):

```
smultb  r6, r6, r10
smulbt r12, r6, r11
smlabb r12, r11, r12, r6
```

The 32-bit `r6` register contains two polynomial coefficients and the bottom half (16 bits) of `r10` has the value $B = \lambda R \bmod {}^{\pm} q$ for some twiddle λ and $R = 2^{16}$. The **smultb** `r6, r6, r10` computes the 32-bit value aB for the polynomial coefficient a stored in the top 16 bits of `r6`. The top half of `r11` contains q' such that $qq' + 1 \equiv 0 \bmod R$. After the **smulbt** `r12, r6, r11`, the `r12` register contains the 32-bit value $(aB \bmod R)q'$. Finally, the bottom half of `r11` contains q. The **smlabb** `r12, r11, r12, r6` instruction computes the 32-bit product of the 2 bottom halves of `r11` and `r12`, and stores the 32-bit sum of the product, and `r6` in `r12`. The `r12` register hence has the 32-bit value

$$(aBq' \bmod R)q + aB.$$

By unsigned Montgomery multiplication, the top and bottom halves of `r12` are (a mod$q$ representative of) $a\lambda$ and zero respectively [24].

Let $f(x)$ be the input polynomial in $\mathbb{F}_q[x]/\langle x^{256}+1\rangle$ with coefficients between $-q$ and q. CryptoLine verifies the coefficients of 128 linear output polynomials $c_i + d_i x$ are between 0 and q. Moreover, the linear output polynomials satisfy the post-condition (5).

The ARM Cortex-M4 implementation for Kyber inverse NTT also uses unsigned Montgomery multiplication in its GS butterflies. Assume all coefficients of the linear input polynomials $c_i + d_i x$ are between $-q$ and q. The linear input polynomials moreover represent a polynomial $f(x)$ such that (6) holds. Then the 256 coefficients a_i must be between $-q$ and q. The post-condition (7) is verified by CryptoLine as well.

Plantard Multiplication. As of early 2025, the most efficient ARM Cortex-M4 implementation for Kyber NTT is reported in [22]. It multiplies polynomial coefficients with Plantard multiplication (Section 2.2.5). Using ARM Cortex-M4's **`smulwb`** instruction, the implementation performs a multiplication, an arithmetic right shift followed by bit masking in one cycle. Concretely, consider the following two instructions from the implementation:

```
smulwb  lr, r10, r6
smlabt  lr, lr, r12, r0
```

The bottom half of the register `r6` contains a 16-bit polynomial coefficient a. The register `r10` is the pre-computed 32-bit value $\hat{b} = -\lambda(R \bmod q)(q^{-1} \bmod \llbracket\rrbracket_1 R) \bmod \llbracket\rrbracket_1 R$ with a twiddle factor λ and $R = 2^{32}$. The **`smulwb`** `lr, r10, r6` instruction takes the 16-bit value in the bottom of `r6` and the 32-bit value in `r10`, performs a signed multiplication, and then stores the top 32-bit value of the 48-bit product in `lr` (note: of course, there is a `smulwt` for the top half). Recall $\tilde{R} = 2^{16}$. The bottom halve of `lr` is

$$p_1 = \left\llbracket \frac{a\hat{b} \bmod \llbracket\rrbracket_1 R}{\tilde{R}} \right\rrbracket_2 = \left\lfloor \frac{a\hat{b} \bmod^{\pm} 2^{32}}{2^{16}} \right\rfloor.$$

Now the top 16 bits of `r12` contains q. The `r0` register has the value $8q$. The **`smlabt`** `lr, lr, r12, r0` instruction computes the product of p_1 (the bottom half of `lr`) and the top half of `r12`, adds the 32-bit value of `r0`, then stores the result in `lr`. After executing the **`smlabt`** `lr, lr, r12, r0` instruction, the top half of `lr` has the value

$$\left\llbracket \frac{qp_1}{R/\tilde{R}} \right\rrbracket_3 = \left\lfloor \frac{qp_1 + 8q}{2^{16}} \right\rfloor = a\lambda \bmod^{\pm} q.$$

Thanks to **`smulwb`**, a mulmod 3329 on the ARM Cortex-M4 is two instructions. After 7 levels of Kyber NTT, the implementation returns 128 linear polynomials $c_i + d_i x$ such that

$$f(x) \equiv c_i + d_i x \bmod [q, x^2 - \zeta^{\mathrm{brv}_8(128+i)}] \text{ and } -8\lfloor q/2 \rfloor < c_i, d_i < 8\lceil q/2 \rceil$$

where $f(x)$ is the input polynomial in $\mathbb{F}_q/\langle x^{256}+1\rangle$ and $0 \le i < 128$.

One would expect that the inverse NTT would be the same process in reverse, but not quite. In contrast to standard GS butterflies, the ARM Cortex-M4 implementation from [22] uses CT butterflies throughout its inverse NTT implementation. The idea is to transform polynomial rings $\mathbb{F}_q[x]/\langle x^n - i\rangle$ to $\mathbb{F}_q[y]/\langle y^n \pm 1\rangle$ through twisting and then add/subtract coefficients. Since twisting is implemented by Plantard multiplication, the computation is exactly CT butterflies but with different twiddle factors.

To see how CT butterflies are used to implement inverse NTT. Recall $\zeta^{2^{k-1}} = -1$. Consider the following isomorphism:

$$
\begin{aligned}
& \mathbb{F}_q[x]/\langle x^{2^k} - 1\rangle \cong \mathbb{F}_q[x]/\langle x^{2^{k-1}} - 1\rangle \times \mathbb{F}_q[x]/\langle x^{2^{k-1}} + 1\rangle \quad \text{substitute } x_0 = x,\ x_1 = \zeta^{-1}x \\
\cong\ & \mathbb{F}_q[x]/\langle x_0^{2^{k-1}} - 1\rangle \times \mathbb{F}_q[x]/\langle \zeta^{-2^{k-1}} x_1^{2^{k-1}} + 1\rangle \cong \mathbb{F}_q[x]/\langle x_0^{2^{k-1}} - 1\rangle \times \mathbb{F}_q[x]/\langle x_1^{2^{k-1}} - 1\rangle \\
\cong\ & \frac{\mathbb{F}_q[x]}{\langle x_0^{2^{k-2}} - 1\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x_0^{2^{k-2}} + 1\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x_1^{2^{k-2}} - 1\rangle} \times \frac{\mathbb{F}_q[x]}{\langle x_1^{2^{k-2}} + 1\rangle} \quad \text{substitute } x_2 = \zeta^{-2}x_0,\ x_3 = \zeta^{-2}x_1 \\
\cong\ & \mathbb{F}_q[x]/\langle x_0^{2^{k-2}} - 1\rangle \times \mathbb{F}_q[x]/\langle x_2^{2^{k-2}} - 1\rangle \times \mathbb{F}_q[x]/\langle x_1^{2^{k-2}} - 1\rangle \times \mathbb{F}_q[x]/\langle x_3^{2^{k-2}} - 1\rangle \\
\cong\ & \prod_{0\le i<2^2} \frac{\mathbb{F}_q[x]}{\langle x_{\mathrm{brv}_2(i)}^{2^{k-2}} - 1\rangle} \cong \cdots \cong \prod_{0\le i<2^k} \frac{\mathbb{F}_q[x]}{\langle x_{\mathrm{brv}_k(i)} - 1\rangle} \cong \prod_{0\le i<2^k} \frac{\mathbb{F}_q[x]}{\langle x - \zeta^{\mathrm{brv}_k(i)}\rangle} \quad \text{since } x_i = \zeta^{-i}x.
\end{aligned}
$$

Recall that variable substitutions ($x_i = \zeta^{-i}x$) are implemented by twisting. It can be then seen from the above that twisting switches between GS and CT butterflies leaving the result and the overall computational effort constant. The reason to use GS or "twisted" NTTs is that its inverse uses CT butterflies throughout, avoiding the repeated potential doubling of coefficients when using GS butterflies when lazy reductions are used.

Let us give a concrete example from the ARM Cortex-M4 implementation with Plantard multiplication in [22]. Recall Kyber's incomplete negacyclic NTT is $\frac{\mathbb{F}_q[x]}{\langle x^{2^8}+1\rangle} \cong \prod_{i=0}^{127} \frac{\mathbb{F}_q[x]}{\langle x^2 - \zeta^{\mathrm{brv}_8(128+i)}\rangle} \cong \prod_{i=0}^{127} \frac{\mathbb{F}_q[x,y]}{\langle x^2 - y, y - \zeta^{\mathrm{brv}_8(128+i)}\rangle}$. Consider

$$
\begin{array}{llll}
c_8 + d_8 x & \in \mathbb{F}_q[x,y]/\langle x^2 - y, y - \zeta^{17}\rangle & c_9 + d_9 x & \in \mathbb{F}_q[x,y]/\langle x^2 - y, y + \zeta^{17}\rangle \\
c_{10} + d_{10} x & \in \mathbb{F}_q[x,y]/\langle x^2 - y, y - \zeta^{81}\rangle & c_{11} + d_{11} x & \in \mathbb{F}_q[x,y]/\langle x^2 - y, y + \zeta^{81}\rangle.
\end{array}
$$

Take $y_{17} = \zeta^{-17}y$ and $y_{81} = \zeta^{-81}y$. Recall $\zeta^{128} \equiv -1 \mod q$. We have

$$
\begin{array}{rcccl}
c_8 + d_8 x & \in & \mathbb{F}_q[x,y]/\langle x^2 - y, y - \zeta^{17}\rangle & = & \mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17} - 1\rangle \\
c_9 + d_9 x & \in & \mathbb{F}_q[x,y]/\langle x^2 - y, y + \zeta^{17}\rangle & = & \mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17} + 1\rangle \\
c_{10} + d_{10} x & \in & \mathbb{F}_q[x,y]/\langle x^2 - y, y - \zeta^{81}\rangle & = & \mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81} - 1\rangle \\
c_{11} + d_{11} x & \in & \mathbb{F}_q[x,y]/\langle x^2 - y, y + \zeta^{81}\rangle & = & \mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81} + 1\rangle.
\end{array}
$$

Therefore

$$
\tfrac{1}{2}((c_8 + c_9) + (d_8 + d_9)x + [(c_8 - c_9) + (d_8 - d_9)x]y_{17}) \quad \longleftarrow \quad (c_8 + d_8 x, c_9 + d_9 x)
$$

is the inverse NTT mapping from $\mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17} - 1\rangle \times \mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17} + 1\rangle$ to $\mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17}^2 - 1\rangle$. Similarly,

$$
\tfrac{1}{2}((c_{10} + c_{11}) + (d_{10} + d_{11})x + [(c_{10} - c_{11}) + (d_{10} - d_{11})x]y_{81}) \quad \longleftarrow \quad (c_{10} + d_{10} x, c_{11} + d_{11} x)
$$

is the inverse NTT mapping from $\mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81} - 1\rangle \times \mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81} + 1\rangle$ to $\mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81}^2 - 1\rangle$.

Now recall $y = \zeta^{17}y_{17} = \zeta^{81}y_{81}$ and hence $y_{81} = \zeta^{-64}y_{17}$. Thus

$$
\begin{array}{rrcl}
 & b_0 + b_1 x + (b_2 + b_3 x)y_{81} & \in & \mathbb{F}_q[x,y_{81}]/\langle x^2 - \zeta^{81}y_{81}, y_{81}^2 - 1\rangle \\
= & b_0 + b_1 x + \zeta^{-64}(b_2 + b_3 x)y_{17} & \in & \mathbb{F}_q[x,y_{17}]/\langle x^2 - \zeta^{17}y_{17}, y_{17}^2 + 1\rangle
\end{array}
$$

The twisting $y_{81} = \zeta^{-64}y_{17}$ is computed by Plantard multiplication in [22]. Moreover,

$$
\begin{array}{rc}
 & \frac{1}{2}\left(\begin{array}{c} (a_0 + b_0) + (a_1 + b_1)x + (a_2 + \zeta^{-64}b_2) + (a_3 + \zeta^{-64}b_3)x)y_{17} + \\ ((a_0 - b_0) + (a_1 - b_1)x)y_{17}^2 + ((a_2 - \zeta^{-64}b_2) + (a_3 - \zeta^{-64}b_3)x)y_{17}^3 \end{array}\right) \\
\longleftarrow & (a_0 + a_1 x + (a_2 + a_3 x)y_{17}, b_0 + b_1 x + \zeta^{-64}(b_2 + b_3 x)y_{17})
\end{array}
$$

is the inverse NTT mapping from $\mathbb{F}_q[x, y_{17}]/\langle x^2 - \zeta^{17} y_{17}, y_{17}^2 - 1\rangle \times \mathbb{F}_q[x, y_{81}]/\langle x^2 - \zeta^{81} y_{81}, y_{81}^2 - 1\rangle$ to $\mathbb{F}_q[x, y_{17}]/\langle x^2 - \zeta^{17} y_{17}, y_{17}^4 - 1\rangle$.

Assume the coefficients of 128 input linear polynomials are between $-\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$. The input polynomials moreover represent a polynomial $f(x)$ such that (6) holds. CryptoLine verifies the ranges of output coefficients a_i are between $-\lfloor q/2 \rfloor$ and $\lceil q/2 \rceil$. Moreover, the output polynomial $F(x)$ satisfies

$$F(x) = \sum_{i=0}^{255} a_i x^i \equiv -2^{32} f(x) \bmod [q, x^{256} + 1].$$

5 Evaluation

We implement our algebraic linear analysis in the CryptoLine toolkit and compare our technique with others by verifying the latest Intel AVX2, ARM aarch and Cortex-M4 assembly implementations for the Kyber and Dilithium NTTs in packages PQClean [26], IPA [22], and pqm4 [25]. Table 1 lists the verified assembly implementations[1]. The column *Multiplication* shows the name of efficient multiplication used in the implementation. *ASM* indicates the number of vector assembly instructions while *CL* counts the number of scalar instructions in the corresponding CryptoLine model for the assembly code.

The CryptoLine models for the PQClean Intel AVX2 and the pqm4 ARM Cortex-M4 implementations for Kyber NTTs are taken from [24]. We construct the CryptoLine models for the other implementations by extracting a running trace from each implementation and translating the running trace to a CryptoLine model. Since the verified implementations do not have conditional branches, a running trace is representative. We then give the specifications of the CryptoLine models as described in Section 4.

We compare three verification techniques in the experiments. The first technique is our algebraic linear analysis where polytope libraries are used to solve linear integer constraints. The second technique is the bit-accurate SMT QFBV solver in CryptoLine. The third technique is based on our algebraic linear analysis but uses SMT LIA (Linear Integer Arithmetic) solvers instead of polytope libraries. For our technique, we use pplpy in the pqm4 ARM Cortex-M4 and the PQClean Intel AVX2 implementations for Kyber inverse NTT[2] and islpy in the other implementations. We use the SMT solvers Boolector and Z3 respectively for SMT QFBV and SMT LIA. Boolector is specially designed for solving SMT QFBV queries and is the default solver of CryptoLine for range checks. Z3 is a general and efficient SMT solver that supports multiple theories.

All implementations contain range and algebraic properties (which involve modular equations) to be verified. We use our technique, SMT QFBV, and SMT LIA to verify range properties (including algebraic soundness checking). For algebraic properties, we use the computer algebra system Singular for implementations with Montgomery multiplication; for those using Barrett or Plantard multiplication, our technique and SMT LIA are used. Singular was used to verify 4 implementations with Montgomery multiplication in [24]. We also verify algebraic properties in the same implementations with Singular. Range properties in these implementations are verified by algebraic linear analysis for comparison. Algebraic linear analysis is used for implementations with Barrett or Plantard multiplication because the correctness of both multiplications involves complex equational reasoning intractable for Singular.

All our experiments are running on a Ubuntu 24.04.1 server with 3.5GHz AMD EPYC 7763 and 2TB RAM. Table 2 shows the experimental results. $\mathrm{T}_{\mathrm{ISL}}$, $\mathrm{T}_{\mathrm{QFBV}}$, and $\mathrm{T}_{\mathrm{LIA}}$ represent the running time of CryptoLine where range checks are carried out by our

[1] After we extracted the CryptoLine models, function names of Kyber implementations in PQclean were changed as a result of NIST's standardization.

[2] islpy does not perform well in the two examples compared with pplpy.

Table 1: Benchmarks with Line of Code Information

Scheme	Package	Arch	Function[3]	Multiplication	ASM	CL
Dilithium	PQClean	AVX2	ntt_avx	Montgomery	2337	25696
			invntt_avx	Montgomery	2265	25904
		aarch64	ntt	Barrett	2016	22994
			invntt_tomont	Barrett	2505	28341
Kyber	PQClean	AVX2	polyvec_ntt	Montgomery	585	14352
			polyvec_invntt_tomont	Montgomery	637	16224
		aarch64	ntt_SIMD_top	Barrett	400	9716
			ntt_SIMD_bot	Barrett	621	11234
			intt_SIMD_top	Barrett	463	11311
			intt_SIMD_bot	Barrett	629	11248
	IPA	Cortex-M4	ntt_fast_plant	Plantard	4160	14471
			invntt_fast_plant	Plantard	4215	15260
	pqm4	Cortex-M4	ntt_fast	Montgomery	5976	13989
			invntt_fast	Montgomery	6243	16053

[1] These function names are suffixes of their original names.

algebraic linear analysis, SMT QFBV, and SMT LIA, respectively. TO indicates a 2-hour timeout. The results show that our algebraic linear analysis outperforms SMT QFBV and SMT LIA significantly. Our technique can verify most implementations using Montgomery, Barrett, and Plantard multiplication in 8 minutes. For the PQClean AVX2 the pqm4 Cortex-M4 implementations for Kyber inverse NTT, our approach requires 53 and 22 minutes, respectively. A reason our approach requires more time in those two implementations is that both implementations are originally specified by relations between the output polynomial and each pair of input coefficients (since Kyber has an incomplete NTT) in [24]. Our new specifications used in the other inverse NTT implementations on the other hand describe relations between the input polynomial of NTT and the output polynomial of inverse NTT, which involve much fewer predicates.

SMT QFBV is slower than our approach in all the implementations. SMT QFBV successfully verifies range checks of Kyber NTT implementations but fails for most Dilithium NTT implementations. Recall the prime number in Kyber is much smaller than that in Dilithium. 16-bit computation is sufficient for Kyber, but 32-bit computation is needed for Dilithium. SMT QFBV does not scale well for 32-bit verification. SMT LIA can verify implementations using Montgomery multiplication but fails to verify all implementations using Barrett and Plantard multiplication. Of the six implementations using Montgomery multiplication, SMT LIA's performance is comparable to ours in four, worse in one, and significantly better in another. We actually wait for the SMT QFBV solver for over two hours beyond the timeout limit on two implementations. In this experiment, the SMT QFBV technique cannot verify the PQClean aarch64 Dilithium NTT within a week, whereas it verifies the PQClean AVX2 Dilithium inverse NTT in approximately one month.

6 Discussion

Multiplication in finite polynomial rings is essential to lattice-based cryptography. For efficiency, lattice-based schemes like Kyber and Dilithium require polynomial multiplication to be implemented by NTTs [31, 32]. Even for polynomial rings unsuitable for NTTs, ingenious techniques have been developed to multiply polynomials through NTTs indirectly [15]. Optimized NTT implementations have become a critical component in lattice-based cryptography.

Efficient NTT implementations however are diverse. Depending on the instruction set architecture, different algorithms have been applied to attain optimal NTT implementations on different architectures. Montgomery multiplication is currently used in Intel AVX2

Table 2: Experimental Results

Scheme	Package	Arch	Function	T_{ISL}	T_{QFBV}	T_{LIA}
Dilithium	PQClean	AVX2	ntt_avx	96s	474s	88s
			invntt_avx	443s	TO[4]	447s
		aarch64	ntt	279s	TO[4]	TO[4]
			invntt_tomont	161s	TO[4]	TO[4]
Kyber	PQClean	AVX2	polyvec_ntt	51s	84s	50s
			polyvec_invntt_tomont	3160s	3666s	669s
		aarch64	ntt_SIMD_top	80s	229s	TO[4]
			ntt_SIMD_bot	115s	215s	TO[4]
			intt_SIMD_top	125s	197s	TO[4]
			intt_SIMD_bot	79s	142s	TO[4]
	IPA	Cortex-M4	ntt_fast_plant	177s	454s	TO[4]
			invntt_fast_plant	99s	218s	TO[4]
	pqm4	Cortex-M4	ntt_fast	162s	218s	419s
			invntt_fast	1291s	1298s	1289s

[4] TO indicates timeout (which is 2 hours)

Dilithium and Kyber NTTs (Section 4.1.1 and 4.2.1). Barrett multiplication is employed in ARM aarch64 Dilithium and Kyber NTTs (Section 4.1.2 and 4.2.2). The optimal ARM Cortex-M4 Kyber NTT currently uses Plantard multiplication instead (Section 4.2.3). The optimized ARM Cortex-M4 implementation moreover twists variables to avoid reduction in inverse NTT. With so many optimizations on different NTT implementations, the correctness of each and every implementation is far from clear. Verifying diverse NTT implementations is an important yet challenging problem.

Algebraic linear analysis is our answer to verify diverse NTT implementations on different architectures. Based on the insight of algebraic abstraction, algebraic linear analysis employs algebraic techniques to verify linear computation in NTT implementations. In contrast to traditional bit-accurate techniques such as SMT QFBV, algebraic linear analysis is more scalable and verifies 32-bit computation in Dilithium NTT easily (Section 5). It moreover outperforms SMT QFBV conclusively for efficient Barrett and Plantard multiplication employed in Kyber NTT. The generality and efficacy of algebraic linear analysis are supported by our extensive experiments. It would be interesting to verify more sophisticated NTT implementations with our technique. Investigations about the limitations of algebraic linear analysis are certainly welcome.

To our knowledge, the PQClean Intel AVX2 and ARM aarch64 implementations for Dilithium NTT have never been verified. The fastest ARM Cortex-M4 Kyber NTT implementation with Plantard multiplication is never verified until now. Due to the generality of algebraic linear analysis, we report the first verification results on 3 NTT implementations for Dilithium and Kyber on Intel AVX2, ARM aarch64 and Cortex-M4. Without our new technique, the verification of Intel AVX2 and ARM aarch64 Dilithium NTT implementations is infeasible for the existing bit-accurate technique SMT QFBV.

References

[1] The Coq proof assistant. To be renamed Rocq, https://coq.inria.fr/.

[2] EasyCrypt: Computer-aided cryptographic proofs. https://github.com/EasyCrypt/easycrypt.

[3] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, Jul 2009.

[4] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013. https://staff.aist.go.jp/reynald.affeldt/documents/arilib-affeldt.pdf.

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

[6] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Science of Computer Programming*, 77(10–11):1058–1074, 2012.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber: Episode IV: Implementation correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, June 2023.

[8] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying kyber - episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 384–421. Springer, Cham, August 2024.

[9] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient word size modular multiplication over signed integers. In *29th IEEE Symposium on Computer Arithmetic, ARITH 2022, Lyon, France, September 12-14, 2022*, pages 94–101. IEEE, 2022.

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

[11] Hanno Becker, John Harrison, and Matthias J. Kannwischer. Works in progress — personal communication. https://github.com/jargh/s2n-bignum-dev/tree/mlkem and https://github.com/pq-code-package/mlkem-native.

[12] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022.

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

[14] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve: Open source (mixed-integer) linear programming system. https://lpsolve.sourceforge.net/.

[15] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings new speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021. https://doi.org/10.46586/tches.v2021.i2.159-188.

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

[17] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19–32):1, 1967.

[18] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1591–1606, New York, NY, USA, 2019. Association for Computing Machinery.

[19] John R. Harrison. HOL Light, a higher-order logic proof assistant. https://github.com/jrh13/hol-light/.

[20] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969.

[21] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography*, volume 10677, pages 341–371, 2017. https://eprint.iacr.org/2017/604.

[22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[23] Vincent Hwang. Private communication. Unpublished thesis.

[24] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR TCHES*, 2022(4):718–750, 2022.

[25] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[26] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[27] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

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

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

[30] NIST, the National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015. https://csrc.nist.gov/pubs/fips/202/final.

[31] NIST, the National Institute of Standards and Technology. Module-lattice-based digitial signature standard, 2024. https://csrc.nist.gov/pubs/fips/204/final.

[32] NIST, the National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard, 2024. https://csrc.nist.gov/pubs/fips/203/final.

[33] Thomas Plantard. Efficient word size modular arithmetic. In *28th IEEE Symposium on Computer Arithmetic, ARITH 2021, Lyngby, Denmark, June 14-16, 2021*, page 139. IEEE, 2021.

[34] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[35] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018.

[36] Mark Shand and Jean Vuillemin. Fast implementations of RSA cryptography. In Earl E. Swartzlander Jr., Mary Jane Irwin, and Graham A. Jullien, editors, *11th Symposium on Computer Arithmetic, 29 June - 2 July 1993, Windsor, Canada, Proceedings*, pages 252–259. IEEE Computer Society/, 1993.

[37] Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification for algebraic abstraction. In Constantin Enea and Akash Lal, editors, *International Conference on Computer Aided Verification*, volume 13966 of *LNCS*, pages 329–349. Springer, 2023.

[38] Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. CoqCryptoLine: A verified model checker with certified results. In Constantin Enea and Akash Lal, editors, *International Conference on Computer Aided Verification*, volume 13966 of *LNCS*, pages 227–240. Springer, 2023.

[39] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1973–1987. ACM Press, October / November 2017.

[40] Sven Verdoolaege. Presburger formulas and polyhedral compilation. Technical report, Polly Labs and KU Leuven, 2021.

[41] Sven Verdoolaege. Integer set library. https://libisl.sourceforge.io/, 2024.

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