IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–0. DOI:XXXXXXXX

Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU

Vincent Hwang[1,2], Jiaxiang Liu[3], Gregor Seiler[4], Xiaomu Shi[3], Ming-Hsien Tsai[5], Bow-Yaw Wang[1] and Bo-Yin Yang[1]

[1] Academia Sinica, Taipei, Taiwan
bywang@iis.sinica.edu.tw,by@crypto.tw
[2] National Taiwan University, Taipei, Taiwan
vincentvbh7@gmail.com
[3] Shenzhen University, Shenzhen
{jiaxiang0924,xshi0811}@gmail.com
[4] IBM Research Zurich, Zurich, Switzerland
gseiler@inf.ethz.ch
[5] National Applied Research Labs, Taipei, Taiwan
mhtsai208@gmail.com

Abstract. Postquantum cryptography requires a different set of arithmetic routines from traditional public-key cryptography such as elliptic curves. In particular, in each of the lattice-based NISTPQC Key Establishment finalists, every state-of-the-art optimized implementation for lattice-based schemes still in the NISTPQC round 3 currently uses a different complex multiplication based on the Number Theoretic Transform. We verify the NTT-based multiplications used in NTRU, Kyber, and SABER for both the AVX2 implementation for Intel CPUs and for the `pqm4` implementation for the ARM Cortex M4 using the tool CryptoLine. We extended CryptoLine and as a result are able to verify that in six instances multiplications are correct including range properties.
We demonstrate the feasibility for a programmer to verify his or her high-speed assembly code for PQC, as well as to verify someone else's high-speed PQC software in assembly code, with some cooperation from the programmer.

Keywords: NIST PQC · NTT · verification · NTRU · Kyber · Saber

1 Introduction

Shor's algorithm [Sho97] on a large-scale ("cryptographically relevant") quantum computer will solve today-intractable integer factorizations and discrete logarithms, hence breaking RSA and Elliptic Curve Cryptography (ECC) which make up almost all currently deployed public-key cryptography. The U.S. National Institute of Standards and Technology (NIST) has preemptively initiated a process (NISTPQC) to select new cryptosystems that withstand quantum computing. This research area is known as Postquantum Cryptography (PQC). This process naturally divides into categories of digital signatures and key encapsulation mechanisms (KEMs) [NIS] and is currently in the third round with 7 finalists and 8 alternate candidates still competing [AASA+20]. Schemes will be standardized when NISTPQC concludes. These will no doubt be important future computational workloads.

Since individual cryptographic operations are often slow themselves, and cryptography is then applied to much, much data, cryptography is always under a lot of pressure to be efficient. A common narrative has cryptographers developing new, faster, cryptographic primitives in reaction to this pressure. However, this is not an accurate depiction. Much

of the actual speed comes from optimization research that actually takes a mathematical function then finds faster ways to compute that function. This research then feeds back into cryptographic designs. Performance pressure also results in a vastly more complex cryptographic software ecosystem. Herein we find many different intricate and often cutting-edge speedups using new mathematical algorithms or new microarchitecture-specific optimizations.

Every round-3 submission in NISTPQC includes hand-optimized software. Contrary to common impression, this is usually solidly faster than generic code compiled with a state-of-the-art "optimizing" compiler. Because PQC needs to survive quantum attacks, they also tend to be also more complex than pre-quantum public-key cryptography. Thus, post-quantum public-key software is usually even more complicated than pre-quantum public-key software like ECC, which can be complicated already. This aggravates any implementation problems. Because we shall be forced to roll PQC software out in a few years, we are also forced to ask ourselves: How do we minimize bug in PQC software? Traditional tests will miss many bugs, as exemplified by the following quote:

> Produced signatures were valid but leaked information on the private key. ... The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. 'being super careful') has failed.
>
> — "OFFICIAL COMMENT" https://tinyurl.com/y5w46bde

Testing only checks that an implementation is correct on a fixed set of selected inputs. There is no guarantee on untested inputs. Given the essentially infinite possibilities in inputs to PQC software, the proportion of tested inputs is always negligible. The obvious answer when we look for a better mousetrap is *formal verification*. This is a process wherein a conclusion can be reached that the software computes the correct outputs for *all* possible inputs—there are no rare (corner) cases that are handled incorrectly.

CryptoLine was developed to help programmers write correct cryptographic assembly programs. In particular, it is designed to verify arithmetic subroutines that make up the operations between elements of finite algebraic structures (rings, finite fields, elliptic curve groups). Such arithmetic subroutine is a common feature to most public-key cryptosystems. It is usually the case that if you run some tests in symmetric crypto, it will catch the bugs. In general, arithmetic operations that make up public-key crypto are harder to test, because a carry or an overflow — the kind of errors in arithmetic that happens when the programmer overlooks something — might happen very, very rarely, and we do not know if a potential attacker has a way to trigger such an event. Biham *et al* discuss scenarios where *hardware* bugs result in attacks [BCS16]. Software bugs can have the same effect, and the attackers can identify bugs in the programs using CryptoLine, just like us.

1.1 Our Contributions

First verification of NTT multiplications in assembly. We produce the first (semi-automatic) verification result for postquantum crypto software. More precisely, we verify the highly complex polynomial multiplications based on the Number Theoretic Transform (NTT) in one instance of each in the NISTPQC Round 3 finalists Kyber, Saber, and NTRU. Our technique is applicable to any other software implementing small ideal-lattice-based cryptosystems, such as NTRU Prime, LAC, or NewHope [LLZ+18, PAA+19, BBC+20], that also use NTT-based multiplications.

As illustrative examples, we picked the fastest software for one instance (parameter set) of each of the three NISTPQC lattice KEM finalists (to the best of our knowledge):

NTRU Intel AVX2: `ntt-polymul`[1] build 3e42ffa ARM Cortex-M4: `pqm4`[2], build d26fee0,

[1] https://github.com/ntt-polymul/ntt-polymul
[2] https://github.com/mupq/pqm4

pull request https://github.com/mupq/pqm4/pull/219.

Kyber Intel AVX2: `PQClean`[3] build 688ff2f ARM Cortex-M4: `pqm4`[2], build 944b3c3

Saber Intel AVX2: `ntt-polymul`[1] build 3e42ffa ARM Cortex-M4: Strategy A by [ACC+22][4]

As shown in Section 5 the time used for these verification efforts is quite tolerable and would have been even less had the programmer been verifying his or her own code.

Extension of the CryptoLine tool. We extend CryptoLine, in particular we introduce *nonlocal compositional reasoning* in order to be able to finish all six instances. Without these extensions, the verification becomes either much slower or impossible.

1.2 Related Work

Faulty multiplication has been exploited as bug attacks [BCS08]. Formal verification on cryptographic programs aims to prove the absence of bugs and hence prevent such attacks. There exist many projects that (e.g. HACL [ZBPB17], Jasmin [ABB+17] and Fiat [EPG+19]) apply a correct-by-construction approach to build correct cryptography programs. This work is about verifying existing programs that have been not written in such a manner.

Various cryptography primitives have been formalized and manually verified in proof assistants (e.g. [Aff13, ANY12, AM07, MG07, MC13, App15, BPYA15, YGS+17]). We are trying to verify software in an automated or at least semi-automated fashion. Note that these are code "in the wild", programs written with an objective of speed or small size, and not with verifiability in mind.

The only verification result to our knowledge that is specifically conducted for postquantum crypto today is EasyCrypt [BBF+21] which verifies protocols, not programs.

Many if not most of the verifications mentioned above use CoQ. We instead use the tool CryptoLine [TWY17, PTWY18, LST+19, FLS+19], described in detail in Section 3.

Because Intel CPUs ("Haswell") and the ARM Cortex-M4 architectures were specified by NISTPQC as standard benchmarking platforms, there is so much literature on optimizing lattice-based cryptography and also so much software available for these chips that we do not claim that the implementations we verified are the best or fastest. They were merely the fastest among the implementations conveniently at hand.

2 Preliminaries

We briefly describe the targets of our verification, then some mathematics involved (modular reductions, and the Number Theoretic Transform).

2.1 NISTPQC3 Finalist Lattice Candidates

We only include enough detail for the reader to understand what we are verifying. For comprehensive coverage, see [NIS].

2.1.1 Kyber

The NISTPQC finalist candidate Kyber [ABD+20b] is a KEM based on the Module Learning With Errors (M-LWE) problem, using a dimension $\ell \times \ell$ module over the ring $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$, with $q = 3329$ and $n = 256$. Kyber is derived from a CPA-secure

[3] https://github.com/PQClean/PQClean
[4] https://github.com/multi-moduli-ntt-saber/multi-moduli-ntt-saber

Table 1: Kyber and Saber Parameter Sets

name	l	(d_1, d_2)	$\eta(s\|s')$	$\eta(e\|e'\|e")$	name	l	$T = 2^{e_T}$	η
`Kyber512`	2	$(10, 4)$	6	4	`LightSaber`	2	2^3	10
`Kyber768`	3	$(10, 4)$	4	4	`Saber`	3	2^4	8
`Kyber1024`	4	$(11, 5)$	4	4	`FireSaber`	4	2^6	6

Public-Key Encryption (PKE) scheme via a Hofheinz–Hövelmanns–Kiltz CCA-transform [HHK17]. For a detailed PKE description see [ABD+20b].

There is 1 $(\ell \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplication (`MatrixVectorMul`) and 0, 1, and 2 $(\ell \times 1)$ inner products of polynomials (`InnerProd`) in each of key generation, encapsulation, and decapsulation respectively. This is because decapsulation needs a full re-encryption. [ABD+20b] specifies that *we do all multiplications via incomplete NTT, and NTT results are in bit-reversed order*. All polynomial multiplications involve one random polynomial mod$q$ and one polynomial (s or s') with coefficients between $\pm\eta/2$, with some multiplicands already in NTT form. E.g., the public matrix A is sampled in (incomplete) NTT domain from a seed via the extendable-output function (XOF) `SHAKE128`.

Parameters. See Table 1 for parameters: Module dimensions ℓ, and widths of the centered binomial distribution η (twice the bound of the coefficients in "small" polynomials; rounding parameters (d_1, d_2) need not concern us), vary according to the parameter sets `Kyber-512`, `-768` (what we verified), and `-1024` (targeting NIST security levels 1, 3, and 5).

2.1.2 Saber

The NISTPQC finalist candidate Saber [DKRV20] is a KEM based on the Module Learning With Rounding (M-LWR) problem, using a module of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$, with $q = 2^{13}$ and $n = 256$.

Saber KEM is also built on top of a CPA-secure PKE via the CCA-transform of Hofheinz-Hövelmanns-Kiltz [HHK17]. For an algorithmic description see [DKRV20].

Just in as Kyber, there is 1 `MatrixVectorMul` in key generation; 1 `MatrixVectorMul` + 1 `InnerProd` in encapsulation; and 1 `MatrixVectorMul` + 2 `InnerProd` in decapsulation. All polynomial multiplications involve one random polynomial mod$q$ and one polynomial (marked s or s') with coefficients between $\pm\eta/2$,

In the Saber base ring $\mathbb{Z}_{2^{13}}$, 2 is not invertible and there are no appropriate principal roots (see 2.3.3 below), making it NTT-unfriendly. Accordingly, the specification samples the public matrix A in the polynomial domain.

Parameters. Module dimensions l and secret distribution parameters η (twice the bound of the coefficients in "small" polynomials; the rounding parameter T need not concern us) vary according to the parameter sets `Lightsaber`, `Saber`, and `Firesaber` (targeting the NIST security levels 1, 3, and 5, cf. Table 1). We verified Saber.

2.1.3 NTRU

The NISTPQC finalist NTRU [CDH+20] is a KEM based on the hardness of the Ring-LWE and NTRU problems. It is based on NTRU as proposed by Hoffstein, Pipher, and Silverman in 1998 [HPS98]. It operates in the three polynomial rings $\mathbb{Z}_3[X]/\mathbf{\Phi_n}$, $\mathbb{Z}_q[X]/\mathbf{\Phi_n}$, and $\mathbb{Z}_q[X]/(\mathbf{\Phi_1} \cdot \mathbf{\Phi_n})$ with $\mathbf{\Phi_1} = (X - 1)$ and $\mathbf{\Phi_n} = (X^{n-1} + X^{n-2} + \cdots + 1)$.

For algorithmic descriptions see [CDH+20]. NTRU achieves its CCA-secure KEM with a variation [SXY18] on the FO transform [FO99], avoiding having to re-encrypt the message during the decapsulation. NTRU is also not NTT-friendly by design, and one of the multiplicands in each product always has coefficients in $\{-1, 0, +1\}$.

Table 2: NTRU Parameter Sets

name	q	n
`ntruhps2048509`	$2048 = 2^{11}$	509
`ntruhps2048677`	$2048 = 2^{11}$	677
`ntruhrss701`	$8192 = 2^{13}$	701
`ntruhps4096821`	$4096 = 2^{12}$	821

Parameters. NTRU proposes 4 parameter sets (Table 2) of which we verified `ntruhps2048509`.

2.2 Modular Reductions

Reductions modulo a small prime q is usually conducted through *signed Montgomery Reduction* [Sei18]: We pick a power of 2 as the "radix" $R > q$, and pre-compute $Q = 1/q \bmod R$. We can then compute $L = (A \bmod R)Q \bmod R$, then $(A - Lq)/R \equiv A/R \pmod q$. Since $R|(A - Lq)$, computing $(A - Lq)/R$ does not require a real division, and in fact only needs a high-limb multiplication (if available) when R has exactly the limb size.

In NTTs we are usually multiplying by known constants $\omega \pmod q$, and Seiler went further, introducing *Montgomery Multiplication* [Sei18]: Pre-compute $\omega' = \omega Q \bmod R$, then $b\omega$ can be computed as follows: $H = \lfloor b\omega/R \rfloor$ (*multiply, high*), then $L = b\omega' \bmod R$ (multiply, low), then $b\omega/R \equiv H - \lfloor Lq/R \rfloor \pmod q$ (again *multiply, high* and subtract).

Notice that the result of Montgomery reduction and multiplication $\pmod q$ is between $\pm q$, not $\pm q/2$. This is an example of *lazy reductions*. In high-speed implementations, the programmer never does any full reductions unless and until absolutely forced to.

2.3 The Number Theoretic Transform (NTT) and Butterflies

NTTs are critically important for speed in long multiplications. Classic works on integer multiplications [SS71, Für09, HVDH21] use them as basic blocks. NISTPQC 3rd round candidates Dilithium, Falcon, and Kyber [ABD+20b, ABD+20a, FHK+17] wrote NTTs into their specs to squeeze out extra efficiency improvements. NTRU, Saber, and NTRU Prime [DKRV20, CDH+20, BBC+20] can also use NTTs for speed [ACC+21, CHK+21].

2.3.1 Standard Fast Fourier Transform (FFT) and NTT

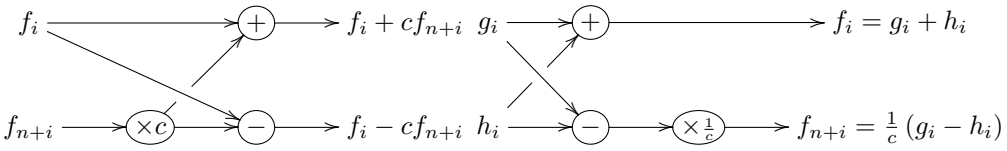
The "usual" radix-2 NTT/FFT means recursively using this ring isomorphism [CT65]:

$$\mathbb{F}[X]/\langle X^{2n} - c^2\rangle \cong \mathbb{F}[X]/\langle X^n - c\rangle \times \mathbb{F}[X]/\langle X^n + c\rangle;$$
$$\sum_{i=0}^{2n-1} f_i X^i \leftrightarrow \left(\sum_{i=0}^{n-1} (f_i + cf_{n+i})\, X^i, \sum_{i=0}^{n-1} (f_i - cf_{n+i})\, X^i\right).$$

which holds if $2c$ is invertible. Considered as an "in-place" operation, starting with a size-$2n$ array of elements of $\mathbb{F}$ representing an element of $\mathbb{F}[X]/\left(X^{2n} - c^2\right)$ and ending with the bottom and top half of that array representing the element of $\mathbb{F}[X]/\left(X^n - c\right)$ and of $\mathbb{F}[X]/\left(X^n + c\right)$ respectively, then with a little change of notation we may depict the map in Figure 1a and its inverse map, up to a factor of 2, in Figure 1b. We refer to c as a *twiddle factor* (of the butterfly). If $2|n$ and $\sqrt{c} \in \mathbb{F}$ can be found we can repeat the process.

As described by Cooley–Tukey, this only stops at linear factors, when we have reduced a polynomial multiplication in $\mathbb{F}[X]/\langle X^{2^k} - c\rangle$ to many independent multiplications in $\mathbb{F}$.

An FFT/NTT as described outputs in a "bit-reversed" order. When the NTT (FFT) is strictly to multiply two polynomials, we can ignore the different output order as long as the inverse NTT takes this into account.



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

Figure 1: The "Butterflies" of Fast Fourier Transforms

2.3.2 "Twisted" FFT and NTT

Gentleman–Sande proposed a slightly different procedure [GS66] in which with the help of a such that $a^n = -1$ we apply recursively the following transformation

$$\frac{\mathbb{F}[X]}{\langle X^{2n} - 1\rangle} \cong \frac{\mathbb{F}[X]}{\langle X^n - 1\rangle} \times \frac{\mathbb{F}[X]}{\langle X^n + 1\rangle} \overset{X=aY}{\cong} \frac{\mathbb{F}[X]}{\langle X^n - 1\rangle} \times \frac{\mathbb{F}[Y]}{\langle Y^n - 1\rangle}.$$

Mapping $X = aY$ from $\mathbb{F}[X]/\langle X^n - c\rangle$ to $\mathbb{F}[Y]/\langle Y^n - 1\rangle$ is called *twisting*. Twisted (Gentleman–Sande) NTTs (FFTs) apply GS butterflies and its inverse apply CT butterflies.

One can see from Figure 1 that if Montgomery Multiplication is used, starting from values bounded by $\pm q/2$, after ℓ layers of CT butterflies, the new values are bounded by $\pm(\ell + \frac{1}{2})q$ whereas GS butterflies return values between $\pm 2^{\ell-1}q$. In general CT butterflies are better for lazy reduction, and as a result *some implementations do normal NTTs going forward and twisted NTTs in inverse so as to be able to use CT butterflies both ways.*

2.3.3 Principal Roots and Incomplete NTTs

To split $\mathbb{F}[X]/\langle X^n - c\rangle$ and repeat it k times requires that there is a $a \in \mathbb{F}$ such that $a^n = c$. Obviously we need $2^k | n$, and when $c = 1$, we need a to be a *principal root* of 1: Let $[n]_q = \sum_{i=0}^{n-1} q^i$ be the q-analog of n. A *principal* n-th root of unity ω is an n-th root of unity satisfying the orthogonality $[n]_{\omega^i} = 0$ for $1 \leq i < n$ [Für09, HVDH21]. The existence of a principal root $\pmod m$ means that $n|(p-1)$ for all primes $p|m$. This definition coincides with *primitive* roots when m is prime.

If we stop short in our sequence of mappings prior to reaching linear factors, we have what is called an "incomplete" NTT/FFT and we are left with modular multiplications of low-degree polynomials. Sometimes we are forced to stop because the appropriate roots do not exist, sometimes because of efficiency considerations.

3 The CryptoLine tool

CryptoLine [TWY17, PTWY18, LST+19, FLS+19] is a tool intended for a programmer to verify his (or her) own arithmetic programs. It was developed with the idea that a programmer need not write within a fixed framework or depend on the whims of the compiler, as in [EPG+19]. Instead, the programmer codes any which way as desired. The main program of CryptoLine is written in OCaml while some subsidiary scripts are in Python.

3.1 The CryptoLine Language

Cryptoline is a domain-specific language for modeling cryptographic assembly programs and their specifications [PTWY18]. Cryptoline is a strongly-typed language. Constants in cryptoline are associated with a type. Variables must have specific types according to *declarations*. Operations can only be between specific types. All casts are explicit.

Let w be a positive integer. The type $\mathsf{uint}\ w$ comprises unsigned integers denoted by bit strings of length w. Similarly, $\mathsf{sint}\ w$ are signed integers denoted by bit strings of length w (2's complement). So $\mathsf{uint}\ w$ denotes integers greater than -1 and less than 2^w and $\mathsf{sint}\ w$ denotes integers greater than $-2^{w-1}-1$ and less than 2^{w-1}. bit is short for uint 1. So in these verifications, we deal with (mostly) uint32, sint32, uint16, sint16, and bit.

$$
\begin{array}{rcl}
\mathit{Num} & ::= & \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{Var} & ::= & \cdots \mid x \mid y \mid z \mid \cdots \\
\mathit{Exp} & ::= & \mathit{Atom} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid \mathit{Exp} \times \mathit{Exp} \\
\mathit{APred} & ::= & \mathit{APred} \wedge \mathit{APred} \mid \mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} \equiv \mathit{Exp} \bmod [\mathit{Exp}, \mathit{Exp}, \ldots, \mathit{Exp}] \\
\mathit{RPred} & ::= & \mathit{Exp} < \mathit{Exp} \mid \mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} \equiv \mathit{Exp} \bmod \mathit{Exp} \\
 & \mid & \mathit{RPred} \wedge \mathit{RPred} \mid \neg \mathit{RPred}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Const} & ::= & \mathit{Num}@\mathit{Type} & \\
\mathit{Atom} & ::= & \mathit{Var} \mid \mathit{Const} &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Inst} & ::= & \mathsf{mov}\ \mathit{Var}\ \mathit{Atom} & \mid \mathsf{cmov}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{add}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} & \mid \mathsf{adds}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{adc}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} & \mid \mathsf{adcs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{sub}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} & \mid \mathsf{subs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{sbb}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} & \mid \mathsf{sbbs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{mul}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} & \mid \mathsf{mull}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{shl}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} & \mid \mathsf{spl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \\
 & \mid & \mathsf{cshl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Num} & \mid \mathsf{join}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
 & \mid & \mathsf{cast}\ \mathit{Var}@\mathit{Type}\ \mathit{Atom} & \mid \mathsf{assert}\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
 & \mid & \mathsf{assume}\ \mathit{APred}\ \&\&\ \mathit{RPred} & \mid \mathsf{ghost}\ \mathit{Var}@\mathit{Type}\ :\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
 & \mid & \mathsf{cut}\ \mathit{APred}\ \&\&\ \mathit{RPred} &
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Decl} & ::= & \mathit{Type}\ \mathit{Var} \\
\mathit{Prog} & ::= & \mathit{Decl}^*\ \mathit{Inst}^*
\end{array}
$$

Figure 2: CryptoLine Syntax

3.1.1 General Instructions

Figure 2 shows the syntax of CryptoLine. An arithmetic instruction retrieves values from *sources* and stores results in *destinations*. For example, $\mathsf{mov}\ v\ a$ copies the source a to the destination v; while depending on the value of c, $\mathsf{cmov}\ v\ c\ a_0\ a_1$ stores either value of sources a_0 or a_1 in the destination v.

To model arithmetic in cryptographic assembly programs, many instructions involve flags. For example, $\mathsf{adds}\ c\ d\ a\ b$ means to take two atomic inputs a, b of (equal) size w, add them into an integer of size $w+1$, then splits the top bit off into the first destination variable (c, the carry) and the second destination variable (d, the destination register of the instruction). "Short" instructions not covering the apparent output range require that overflows did not happen. For example, $\mathsf{add}\ d\ a\ b$ says to add atomic inputs a, b of the same size w and checks that the result fits in the destination variable d of the same size w. Recall that signed and unsigned integers have different bounds when they are of size w. CryptoLine type system infers the types of sources to decide if their sum is representable by the destination variable.

In addition to additions and subtractions, to deal with multi-word arithmetic, CryptoLine also includes multi-word constructs for example, those that split (spl) or join (join) words, as well as multi-word shifts (cshl). Finally, there are long multiplications (mull) as well as their "short" version (mul). When a signed integer of size $w+v$ is split into two integers of size w and v respectively, the more significant destination is signed but the less significant destination is unsigned. CryptoLine type system again infers types of destination variables to ensure all arithmetic computation is within proper bounds.

Finally, the cast instruction casts the source to a designated type. CryptoLine checks whether integers in all executions are within the bounds of the designated type.

Type inference and bound checking are useful in detecting over- and under-flow. They are especially helpful when a signed Montgomery reduction is used. Particularly, both signed and unsigned integers coexist after a signed Montgomery reduction in various NTT implementations. All arithmetic instructions must be without over- or under-flow.

3.1.2 Asserts and Assumes

As a modeling language, CryptoLine also provides special instructions for verification purposes. The assert P && Q instruction checks if both the algebraic predicate P and range predicate Q are true among all executions. An *algebraic predicate* is a conjunction of equations or modular equations. A *range predicate* is an arbitrary Boolean formula over comparisons, equations, or modular equations. In CryptoLine, algebraic predicates are verified by CASs; and range predicates are verified by SMT solvers. When programmers would like to check if their programs compute as expected, they can add assert instructions with intended algebraic or range predicates at suitable locations. CryptoLine will verify these predicates automatically.

The assume P && Q instruction on the other hand imposes the algebraic predicate P and range predicate Q on all executions. Effectively, P and Q become premises after the assume instruction. assume P && Q are used to summarize previously verified predicates in assert P && Q to save verification time. Another frequent use of assume is to pass verified predicates between CASs and SMT solvers. Recall that different techniques are applied to verify algebraic and range predicates in CryptoLine respectively. When a range predicate is verified, the established property is unknown to CASs and vice versa. CASs nevertheless can be informed of verified range predicates with assume. Consider the following sequence of instructions

$$\mathsf{assert}\ \mathsf{true}\ \&\&\ Q \qquad \mathsf{assume}\ Q\ \&\&\ \mathsf{true}.$$

CryptoLine first verifies the predicate Q with SMT solvers in the assert instruction. If Q holds for all executions, the predicate is passed to CASs via the assume instruction. Particularly, the *short* add d a b instruction is implicitly adds c d a b followed by assert true && $c = 0$ and assume $c = 0$ && true. CryptoLine first asserts the carry $c = 0$ with SMT solvers and assumes $c = 0$ in CASs.

3.1.3 Compositional Reasoning with Ghost Variables and Cuts

Compared with programs for field or group operations in elliptic curve cryptography, NTT implementations are significantly larger. Montgomery ladderstep in Curve25519 takes four 255-bit field elements and one 256-bit exponent as its inputs. There are roughly 1.3×10^3 input bits. Kyber768 NTT, on the other hand, takes 256 12-bit coefficients ($\approx 3.0 \times 10^3$ bits) as the inputs. Saber NTT takes 256 13-bit coefficients ($\approx 3.3 \times 10^3$ bits). NTRU2048509 takes 509 11-bit coefficients ($\approx 5.6 \times 10^3$ bits). Since input bits of various NTTs are multiples of those in Montgomery ladderstep, the information to be processed is significantly larger. It is perhaps natural to expect much longer cryptographic programs in post-quantum cryptography.

Lengthy cryptographic programs pose new challenges to formal verification. Since verification aims to establish program correctness for all inputs, an extra input bit can double the number of inputs. Longer computation induced by lengthy programs also increases program states for analysis. The infamous state explosion problem severely limits the applicability of formal verification in practice. To verify various NTT implementations formally, new techniques are added to improve the scalability of CryptoLine significantly.

Ghost Variables Computation in a cryptographic program often runs in clearly demarcated stages. The verifier often needed to specify mid-conditions to summarize the

computation "so far" by stages as well. Sometimes, one would like to specify the mid-condition by relating variable values before and after the stage. When the program computes "in place", variable values prior to the stage would be overwritten. Ghost variables in CryptoLine allow verifiers to store values for later reference.

Consider, for instance, the computation of NTT by levels. For efficiency, cryptographic assembly programs often load data at level 0 and compute in registers for later levels. At the beginning of each level, verifiers can store register values in ghost variables. At the end of the level, verifiers specify the relation between ghost variables and registers in the mid-condition. The computation can then be verified by levels.

Cuts Compositional reasoning is a divide-and-conquer technique widely used for ameliorating the state explosion problem in formal verification. The basic idea is to reduce large verification problems into smaller problems. If small problems can be solved, large problems are verified as well. The question, of course, is how to perform such a reduction soundly to avoid incorrect verification results.

CryptoLine provides a simple mechanism to reason about cryptographic programs compositionally. The $\mathsf{cut}\ P\ \&\&\ Q$ instruction allows CryptoLine to verify a program by parts. Let Π_0 and Π_1 be sequences of instructions. Consider the following CryptoLine program

$$\Pi_0 \quad \mathsf{cut}\ P\ \&\&\ Q \quad \Pi_1.$$

CryptoLine transforms the program into the following two programs

$$\Pi_0 \quad \mathsf{assert}\ P\ \&\&\ Q \qquad \text{and} \qquad \mathsf{assume}\ P\ \&\&\ Q \quad \Pi_1.$$

In other words, CryptoLine first verifies the predicates P and Q at the end of Π_0. If both predicates hold, CryptoLine then uses P and Q as premises to verify Π_1. The program $\Pi_0\ \ \Pi_1$ is divided into two smaller programs: $\Pi_0\ \ \mathsf{assert}\ P\ \&\&\ Q$ and $\mathsf{assume}\ P\ \&\&\ Q\ \ \Pi_1$. If any of them fails to verify, the original program $\Pi_0\ \ \Pi_1$ fails as well. The reduction is clearly sound. Both predicates P and Q are verified before they are assumed as premises. Effectively, P and Q can be seen as a summary of the computation in Π_0. The sub-program Π_1 is in turn verified with respect to the summary. Observe that $\mathsf{cut}\ P\ \&\&\ Q$ divides a program $\Pi_0\ \ \Pi_1$ into two sub-programs Π_0 and Π_1 by locality. Since computation dependency often coincides with code locality, the predicates P and Q suffice to summarize the computation of Π_0 and verify the computation of Π_1. We therefore call the condition $P\ \&\&\ Q$ in the cut instruction as a *mid-condition*.

Despite of its applicability in verification, classical compositional reasoning with cut s is insufficient for verifying NTT implementations for post-quantum cryptosystems effectively. For lattice-based cryptosystems, input polynomials for NTTs have degrees in hundreds or even thousands. Consider the 7-level NTT used in Kyber768 as an example. Since different levels have different patterns of computation, implementations naturally compute Kyber768 NTT by levels. A naïve decomposition for Kyber768 NTT implementations would be as follows.

$$\begin{array}{ll}
\Pi_0 & \text{(* first level *)} \\
\mathsf{cut}\ P_0\ \&\&\ Q_0 & \text{(* summary of first level *)} \\
\vdots & \\
\Pi_5 & \text{(* sixth level *)} \\
\mathsf{cut}\ P_5\ \&\&\ Q_5 & \text{(* summary of sixth level *)} \\
\Pi_6 & \text{(* seventh level *)}
\end{array}$$

Using cut's, Kyber768 NTT is divided into seven sub-programs by levels; each level has 256 12-bit coefficients. Verifying all 256 coefficients are computed correctly at each level is certainly better than verifying seven levels of computation. Yet it is far from ideal. In

```
mov b 0@bit
cut 0 : b = 0 && b = 0
mov b 1@bit
cut 1 : b = 1 && b = 1
cmov x b 3142@uint16 2718@uint16
cut 2 : x = 42 && x = 42 prove with 0, 1
```

(a) Before SSA Transformation

```
mov b_0 0@bit
cut 0 : b_0 = 0 && b_0 = 0
mov b_1 1@bit
cut 1 : b_1 = 1 && b_1 = 1
cmov x_0 b_1 3142@uint16 2718@uint16
cut 2 : x_0 = 42 && x_0 = 42 prove with 0, 1
```

(b) After SSA Transformation

Figure 3: CryptoLine Fragments

Kyber768 NTT, recall that a coefficient at level ℓ depends only on two coefficients at level $\ell - 1$ for $0 < \ell \leq 6$. If Kyber768 NTT implementations could be decomposed by dependencies, it would further reduce the size of verification problems and improve the efficiency of formal verification.

Such decompositions however are not attainable through classical compositional reasoning with cut's. Since Kyber768 NTT implementations compute by levels, a coefficient may be computed long after its dependent coefficients were computed. Code locality is therefore different from computation dependency. The cut instruction on the other hand requires the correspondence between code locality and computation dependency. Classical compositional reasoning with cut's cannot further decompose the Kyber768 NTT computation at each level. More sophisticated compositional reasoning is needed.

To verify NTT implementations in lattice-based post-quantum cryptosystems, we extend the CryptoLine cut instruction to support non-local compositional reasoning. To see how it works, consider a Kyber768 NTT implementation again as follows.

```
                                  Π_{0,0}                          (* 1st pair of coefficients in first level *)
cut   0 : P_{0,0} && Q_{0,0}                                       (* summary of 1st pair *)
                                    ⋮
                                  Π_{0,127}                        (* 128th pair of coefficients in first level *)
cut 127 : P_{0,127} && Q_{0,127}                                   (* summary of 128th pair *)
                                  Π_{1,0}                          (* 1st pair of coefficients in second level *)
cut 128 : P_{1,0} && Q_{1,0} prove with 0, 64                      (* summary of 1st pair *)
                                    ⋮
```

Now the NTT implementation is decomposed by coefficient pairs. Additionally, each cut instruction is assigned to a number for reference. When a cut instruction is verified, our extension allows verifiers to add more premises by cut numbers. In the above example, the first coefficient pair of the second level depends on the first and sixty-fifth coefficient pairs of the first level. We therefore add the corresponding cut numbers as additional premises to verify the coefficients in the second level of Kyber768 NTT. Other coefficient pairs are verified similarly. Our extension admits more refined compositional reasoning. It allows us to verify NTT implementations with several hundreds of input coefficients effectively.

In cryptographic assembly implementations, registers are necessarily reused in computation. Care must be taken to avoid unsound verification results. Consider the fragments in Figure 3. In Figure 3a, the bit variable b is set to 0 and the computation is summarized by cut 0. Then b is set to 1 and summarized by cut 1. The conditional assignment then sets the variable x to either 3142 or 2718 by the value of b. At cut 2, CryptoLine is asked to verify whether x is 42 with premises $b = 0$ (from cut 0) and $b = 1$ (from cut 1). Since the conjunctive premise $b = 0$ and $b = 1$ is always false, cut 2 is verified vacuously. That is, x is 42. This is unsound.

To avoid unsoundness, our extension transforms CryptoLine programs to the static single assignment (SSA) form before formal analysis (Figure 3b). The SSA transformation allows our analysis to identify different versions of the same variable uniquely. After SSA transformation, the premises for `cut` 2 are $b_0 = 0$ and $b_1 = 1$. Their conjunction is not false. CryptoLine fails to verify $x = 42$ and finds a counterexample easily. In fact, x_0 is always 3142, and independent of b_0 as expected.

3.1.4 Techniques of Using CryptoLine

Using `itrace.py` and `to_zdsl.py`, a CryptoLine program can be obtained rather easily. Verifiers need to annotate the CryptoLine program with a proper pre-condition, post-condition, and possibly several mid-conditions. These conditions can be derived with the help of programmers or by inspecting the program. Verifiers may choose to specify these conditions with algebraic or range predicates. Since CryptoLine employs different techniques to verify different predicates, its effectiveness varies by the choice of verifiers' specification.

Range predicates are verified by SMT solvers. Very roughly, CryptoLine translates programs into Boolean circuits whose free inputs are the input parameters of `main`. The pre-condition is an additional constraint on free inputs. The *negation* of the post-condition is another constraint on the Boolean circuits. The verification tool then calls an SMT solver to check if the Boolean circuit with constraints is satisfiable. If the answer is "SAT", the negation of the post-condition holds for certain input values satisfying the pre-condition. The verification fails (and we can output those inputs as counterexamples). If the answer is "unSAT", the SMT solver has determined that there are no input value satisfying the pre-condition but falsifying the post-condition at the end of the program. The verification succeeds.

This is a well-established technique widely used in hardware and bit-accurate software verification. Verifying range predicates requires minimal human guidance, verifiers are recommended to write range predicates in general. SMT solvers however do have limitations. For instance, it is widely known that SMT solvers are ineffective in verifying non-linear computation. Cryptographic programs almost surely perform non-linear computation. A more effective verification technique is needed for such programs.

CryptoLine employs CASs to verify algebraic predicates. Particularly, non-linear equations and modular equations can be verified easily by the algebraic technique. The verification tool essentially translates every CryptoLine instruction into polynomial equations. For example, adds c d a b is translated to $2^w c + d = a + b$ and $c(1-c) = 0$ when a and b are unsigned integers of size w; cmov d c a b is translated to $d = ca + (1-c)b$. Note that all possible executions of the instructions adds c d a b or cmov d c a b are roots of corresponding polynomial equations. A CryptoLine program is thus translated to a set of polynomial equations. All program executions are also roots of the set of polynomial equations. To verify if all program executions must satisfy the post-condition, it suffices to verify if all roots of polynomial equations for the program are also roots of the polynomial equations in the post-condition. CryptoLine calls a CAS to solve this algebraic problem. Instead of logical techniques, non-linear computation is thus verified by algebraic techniques.

Verifiers are recommended to write algebraic predicates to verify non-linear computation. Algebraic predicates nevertheless are very restrictive. They do not allow comparison and must be conjunctive. It is sometimes necessary to combine both CASs and SMT solvers to verify conditions. Verifiers need to be creative to pass information between the two techniques via assert and assume. Human guidance is still needed during verification.

Consider, for example, the signed Montgomery reduction in Section 2.2. We have $A - Lq = A - ((A \bmod R)Q \bmod R)q \equiv A - AQq \equiv 0 \pmod R$. To compute $A - Lq$ requires a full multiplication, its value is stored in two registers r_H and r_L where the

low-limb register r_L is always zero. Because of non-linear computation, SMT solvers show $r_L = 0$ but require some effort. CASs easily show $r_L \equiv 0 \pmod R$ but not $r_L = 0$ on the other hand. Since the radix R is precisely the word size, $r_L \equiv 0 \pmod R$ is actually $r_L = 0$. Verifiers can safely assume $r_L = 0$ after CASs assert $r_L \equiv 0 \pmod R$.

3.2 Walkthrough: How the AVX2 Kyber768 NTT is Verified

Notations. NTT layers go up from 0, and inverse NTT (iNTT) layers count down to 0.

- $F = \sum_{k=0}^{n-1} f_k X^k \in \mathbb{Z}_q[X]$ is the polynomial we began with. If we central-reduce F first before the NTT, the result is marked with a "hat" ($\hat{F}$, $\hat{f}_k$).
- After NTT level i, the j-th polynomial is $G_{i,j} = \sum_{k=0}^{n/2^{i+1}-1} g_{i,j,k} X^k$, $0 \le j < 2^{i+1}$.
- $\zeta_{i,j}$ is the roots of unity used at the end of level i (counting up).
- $\mathbb{Z}_q[X]/\langle X^{n/2^L} - \zeta_0 \rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^{n/2^L} - \zeta_{2^L-1} \rangle$ contains the NTT result, so $\zeta_{L-1,j} = \zeta_j$, where $0 \ldots (L-1)$ number the L levels.
- The NTT result comprises polynomials $P_j = \sum_{k=0} p_{j,k} X^k$ (we see the array of $p_{j,k}$'s).
- After iNTT level i, the j-th polynomial is $H_{i,j} = \sum_{k=0}^{n/2^i-1} h_{i,j,k} X^k$, $0 \le j < 2^i$.
- $\overline{F}$ is the result of the inverse NTT.

We will first give an overview of what is involved in verifying a high-speed NTT in assembly — handwritten by somebody else — with this walk-through. The Intel AVX2 Kyber768 NTT is chosen because it is simplest and illustrates our points well.

Starting from the executable, a running trace of a subroutine is extracted to be verified, using the script `itrace.py` that calls `gdb`. The extracted trace looks like the following:

```
$ itrace.py test ntt PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas
$ more  PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas

#PQCLEAN_KYBER768_AVX2_polyvec_ntt:
:
# [some bookkeeping information]
:
vmovdqa (%rsi),%ymm0            #! EA = L0x5555556395e0; Value = 0x0d010d010d010d01; PC = 0x55555556eb4f
vpbroadcastq 0x140(%rsi),%ymm15 #! EA = L0x555555639720; Value = 0x7b0a7b0a7b0a7b0a; PC = 0x55555556eb53
vmovdqa 0x100(%rdi),%ymm8       #! EA = L0x7fffffffb080; Value = 0xffff0000ffff0001; PC = 0x55555556eb5c
:
vpbroadcastq 0x148(%rsi),%ymm2  #! EA = L0x555555639728; Value = 0xfd0afd0afd0afd0a; PC = 0x55555556eb7c
vpmullw %ymm15,%ymm8,%ymm12     #! PC = 0x55555556eb85
:
vpmulhw %ymm2,%ymm8,%ymm8       #! PC = 0x55555556eb99
:
vmovdqa (%rdi),%ymm4            #! EA = L0x7fffffffaf80; Value = 0x0000ffff00000000; PC = 0x55555556eba9
:
vpmulhw %ymm0,%ymm12,%ymm12     #! PC = 0x55555556ebbc
:
vpaddw %ymm8,%ymm4,%ymm3        #! PC = 0x55555556ebcc
vpsubw %ymm8,%ymm4,%ymm8        #! PC = 0x55555556ebd1
:
:
```

`test` was a test program compiled to use the routine in question. Most instructions start with `vp` indicating the Intel AVX2 instruction set. We note that the above code loads two sets of 64 coefficients into `%ymm4-7` and `%ymm8-11`, then a set of twiddle factors (in Montgomery form) into `%ymm15` and starts butterflies using Montgomery multiplications. The program actually does 4 butterflies at a time, the snippet above only contains code pertaining to just one butterfly (the dots here as below stand for cut material).

We put a set of translation rules on top of a running trace with a set of translation rules and then run another script, `to_zdsl.py`. The rules to the above program looks like

```
#! $1c(%rsi) = %%EA
#! (%rsi) = %%EA
#! $1c(%rdi) = %%EA
#! (%rdi) = %%EA
#! %ymm$1c = %%ymm$1c
#! vpbroadcastq $1ea, $2v -> mov $2v_0 $1ea;\nmov $2v_1 $1ea[+2];\nmov $2v_2 $1ea[+4];\nmov $2v_3 $1ea[+6]; \
   \nmov $2v_4 $1ea;\nmov $2v_5 $1ea[+2];\nmov $2v_6 $1ea[+4];\nmov $2v_7 $1ea[+6]; ...
#! vmovdqa $1ea, $2v -> mov $2v_0 $1ea;\nmov $2v_1 $1ea[+2];\nmov $2v_2 $1ea[+4];\nmov $2v_3 $1ea[+6]; ...
#! vmovdqa $1v, $2ea -> mov $2ea $1v_0;\nmov $2ea[+2] $1v_1;\nmov $2ea[+4] $1v_2;\nmov $2ea[+6] $1v_3; ...
```

The initial lines specify variables. The lines `#! $1c(reg)=%%EA` and `#! (reg)=%%EA` map register-indirect-offset addressed memory to more memory variables. Each line after that describes an instruction. For example, `vpbroadcastq $1ea, $2v` means to broadcast the 64-bit memory location `$1ea` into each 64-bit limb of the target register `$2v`; `vpmullw` means to multiply each pair of matching 16-bit limbs in the two source registers into the corresponding limb in the target register, etc. Note we have to read the source and understand what is going on to annotate the program appropriately. For example the multiplication instructions require special care:

```
#! vpmullw $1v, $2v, $3v -> smull mulH$2v_0 mulL_0 $1v_0 $2v_0;\n ...;\ncast $3v_0@sint16 mulL_0;\n ...
#! vpmulhw %%ymm0, $1v, $2v -> smull mulH_0 red_0 $1v_0 ymm0_0;\nassert true && red_0 = mulLymm_0; \
   \nassume red_0 = mulLymm_0 && true;\n ... \nmov $2v_0 mulH_0;\n ...
#! vpmulhw $1v, $2v, $3v -> smull mulH_0 mulL$2v_0 $2v_0 $1v_0;\n ...\nmov $3v_0 mulH_0;\n ...
```

In CryptoLine, multi-limb multiplications always return unsigned lower parts, but we are using signed integers throughout, so in the translation rules for `vpmullw`, we need to typecast `@sint16` for each limb. A high-limb multiplication is often troublesome either with the matching lower-limb multiplication somewhere else in the code, or something assumed about the lower limb. Here, in a signed Montgomery multiplication, what is assumed is that particular pairs of unused lower-limbs are equal, and we can translate appropriately as `%ymm0` has always 16 q's. One can find this in the code, captured in the CryptoLine program by `vmovdqa (%rsi),%ymm0 #! EA = L0x5555556395e0;` and (see below) `mov L0x5555556395e0 ( 3329)@sint16; ...`, allowing us to annotate correctly.[5]

At this point, the script `to_zdsl.py` converts each actual CPU instruction into one or more lines in CryptoLine instructions, usually the latter in AVX2 code.

```
$ to_zdsl PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas > PQCLEAN_KYBER768_AVX2_polyvec_ntt.cl

(* #! -> SP = 0x7fffffffa358 *)
:
(* many bookkeeping instructions deleted *)
:
(* vmovdqa (%rsi),%ymm0                  #! EA = L0x5555556395e0; Value = 0x0d010d010d010d01; PC = ...
mov ymm0_0 L0x5555556395e0;
mov ymm0_1 L0x5555556395e2;
:
mov ymm0_f L0x5555556395fe;
(* vpbroadcastq 0x140(%rsi),%ymm15       #! EA = L0x555555639720; Value = 0x7b0a7b0a7b0a7b0a; PC = ...
mov ymm15_0 L0x555555639720;
mov ymm15_1 L0x555555639722;
:
..............
```

After some irrelevant bookkeeping instructions, each vector instruction splits into 16× word-sized (16-bit) actions by our translation rules. Now, we set down what the constants (copied from source code) are, the entering conditions (inputs, assumptions/pre-conditions), and the concluding conditions (outputs, requirements/post-conditions). Again this requires understanding what the code does, and some scripts to generate the annotations.

```
proc main (
sint16 f000, sint16 f001, sint16 f002, sint16 f003,
...
```

[5] This is a benefit of handwritten assembly; equivalent intrinsics compiled code would migrate the q values from register to register over the course of the whole program, making our annotations much harder.

```
sint16 f252, sint16 f253, sint16 f254, sint16 f255
) =
{
true && and [
(-3329)@16 <s f000, f000 <s (3329)@16, (-3329)@16 <s f001, f001 <s (3329)@16,
...
(-3329)@16 <s f254, f254 <s (3329)@16, (-3329)@16 <s f255, f255 <s (3329)@16
]
}
(*********** initialization ***********)
mov L0x7fffffffaf80 f000; mov L0x7fffffffaf82 f001;  mov L0x7fffffffaf84 f002;
...
mov L0x7fffffffb17e f255;
```

We declare the entering conditions. Each "condition" actually comprises two specifications: an algebraic part, to be checked with a Computer Algebra System (CAS; defaults to Singular but can be Magma, Mathematica, or maple), and a range part, to be checked using an SMT (Satisfiability Module Theory) solver, here via Boolector. In the preamble above, the first `true` specifies that there are no restrictions algebraically on the input array, but the second portion restricts each entering polynomial coefficient to be between $\pm q$. Then "initialization" assign each starting 16-bit limb in memory, represented by `L(hex address)`, to an input variable `f###`.

```
(*********** constants ***********)
mov L0x5555556395e0 (  3329)@sint16; mov L0x5555556395e2 (  3329)@sint16;
...
mov L0x555555639600 ( -3327)@sint16; mov L0x555555639602 ( -3327)@sint16;
...
mov L0x555555639620 ( 20159)@sint16; mov L0x555555639622 ( 20159)@sint16;
...
mov L0x555555639adc (    32)@sint16; mov L0x555555639ade (    32)@sint16;
(*********** ghost polynomial ***********)
ghost x@bit, inp_poly@bit : inp_poly**2 =
f000*(x**0) + f001*(x**1) + f002*(x**2) + f003*(x**3) +
...
f252*(x**252) + f253*(x**253) + f254*(x**254) + f255*(x**255)
 && true;

(* main body of program goes here ... *)
```

Each PQClean NTT uses an array of twiddle factors that already resides in memory, and we copy the numbers (as in the snippet) directly from the source, inserted using a `python` script. The `ghost` polynomial is a compositional reasoning gadget that combines the entering coefficients into one entity (cf. Section 3.1.3). After level 0 is completed, we fill in the conditions according to the description of the NTT in Section 2.3.1.

```
:
(*********** SUMMARY OF LEVEL 0 ***********)

cut and [
eqmod (inp_poly**2)
(L0x7fffffffaf80*(x**0) + L0x7fffffffaf82*(x**1) + L0x7fffffffaf84*(x**2) +
:
 L0x7fffffffb07c*(x**126) + L0x7fffffffb07e*(x**127))
[3329, x**128 - (1729)],
eqmod (inp_poly**2)
(L0x7fffffffb080*(x**0) + L0x7fffffffb082*(x**1) + L0x7fffffffb084*(x**2) +
:
 L0x7fffffffb17c*(x**126) + L0x7fffffffb17e*(x**127))
[3329, x**128 - (1600)]]
&&
and [
(-6658)@16 <s L0x7fffffffaf80, L0x7fffffffaf80 <s (6658)@16,
:
(-6658)@16 <s L0x7fffffffb17e, L0x7fffffffb17e <s (6658)@16];

(*********** LEVELS 1..6, explained below ***********)
:
```

The incomplete NTT in the Intel AVX2 implementation from PQClean [PQC21] does

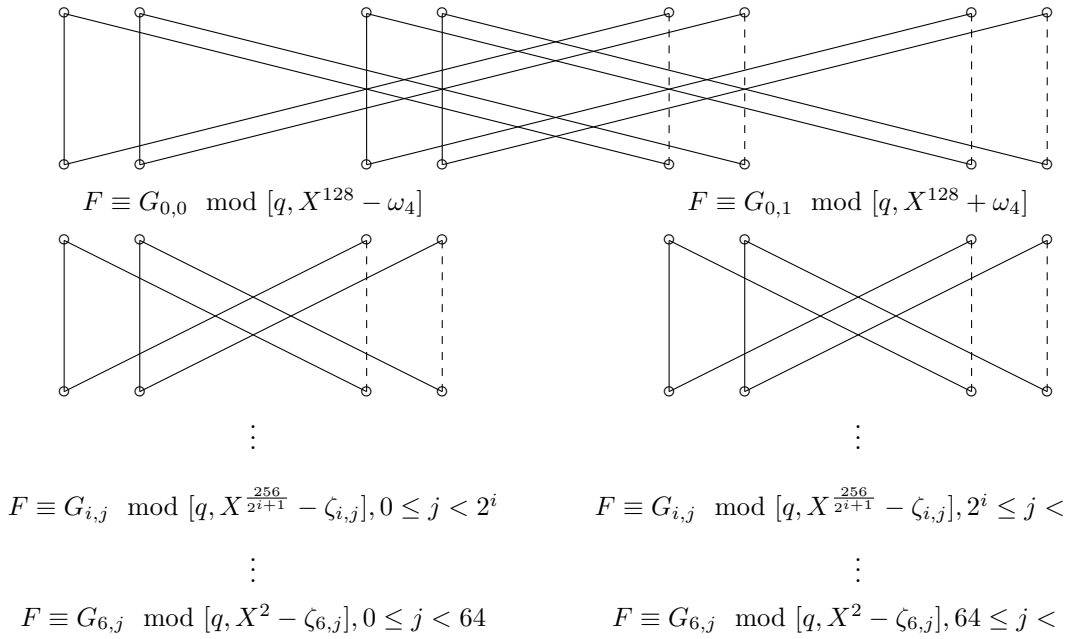


Figure 4: Workflow of verifying AVX2 implementation for Kyber NTT.

the following map (where ζ_i denote all the primitive 256-th roots of unity in $\mathbb{Z}_q$):

$$\begin{aligned} \mathbb{Z}_q[X]/\langle X^{256}+1\rangle &\rightarrow \mathbb{Z}_q[X]/\langle X^{128}-\omega_4\rangle \times \mathbb{Z}_q[X]/\langle X^{128}+\omega_4\rangle \\ \rightarrow \cdots &\rightarrow \mathbb{Z}_q[X]/\langle X^2-\zeta_0\rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^2-\zeta_{127}\rangle \end{aligned}$$

In this AVX implementation, a 256-bit vector register contains 16 16-bit signed integer coefficients. NTT multipliers (roots of unity) moreover are in Montgomery form. Each multiplication is hence always combined with a signed Montgomery reduction. Because of Montgomery reductions and the small magnitude of q, all coefficients are representable in 16 bits at all seven levels. There is no overflow. No extra reduction is needed.

However, one notes that the Intel AVX2 implementation does not compute NTT strictly by levels. There is level 0, in which all 256 coefficients are used together. Then from level 1 onward, at most 128 coefficients are needed at a time. The implementation therefore uses eight 256-bit vector registers to hold the coefficients of the NTT at each level. After level 6 for the first 128 coefficients is done, the last 128 coefficients are loaded and then the NTT levels 1 through 6 for these coefficients are performed.

We follow the same strategy in verification. The mid-conditions that we see at the end of level 0 specify that

$$F \equiv G_{0,j} \mod [q, X^{128} - \zeta_{0,j}] \text{ for all } 0 \le j < 2$$

and

$$-2q < g_{0,j,k} < 2q \text{ for all } 0 \le j < 2, 0 \le k < 128.$$

Here the $\zeta_{0,j}$ are 1729 and 1600, the principal 4th roots of unity (denoted as $\pm\omega_4$ in the map above). At level $i > 1$, we specify these mid-conditions for the first 128 coefficients

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}] \text{ for all } 0 \le j < 2^i$$

and

$$-(2+i)q < g_{i,j,k} < (2+i)q \text{ for all } 0 \le j < 2^i, 0 \le k < 256/2^{i+1}.$$

We show the cut at level 1 as an example, first half of coefficients:

```
:
(*********** SUMMARY OF LEVEL 1 0 ***********)

cut
and [
eqmod (inp_poly**2)
(ymm3_0*(x**0) + ymm3_1*(x**1) + ymm3_2*(x**2) + ymm3_3*(x**3) +
:
 ymm6_c*(x**60) + ymm6_d*(x**61) + ymm6_e*(x**62) + ymm6_f*(x**63))
[3329, x**64 - (2580)],
eqmod (inp_poly**2)
(ymm8_0*(x**0) + ymm8_1*(x**1) + ymm8_2*(x**2) + ymm8_3*(x**3) +
:
 ymm11_c*(x**60) + ymm11_d*(x**61) + ymm11_e*(x**62) + ymm11_f*(x**63))
[3329, x**64 - (749)]]
&&
and [
(-9987)@16 <s ymm3_0, ymm3_0 <s (9987)@16,
:
(-9987)@16 <s ymm11_f, ymm11_f <s (9987)@16];
```

The 128 coefficients at the end of the first half level 1 form two degree-63 polynomials related to the input polynomial by modular equivalence. At the same time, each coefficient is guaranteed to be less than $3q$ in magnitude. As above, at level i, the polynomials are split further with equivalence modulo various $X^{256/2^{i+1}} - \zeta_{i,j}$ and bound by $\pm(2+i)q$. Similarly, the following mid-conditions are used for the last 128 coefficients at level $i > 1$:

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}] \text{ for all } 2^i \le j < 2^{i+1}$$

with ranges

$$-(2+i)q < g_{i,j,k} < (2+i)q \text{ for all } 2^i \le j < 2^{i+1}, 0 \le k < 256/2^{i+1}.$$

Figure 4 is an illustration. We probably need not repeat ourselves. At the end of the second half level 6, we fill in the following concluding conditions:

```
#retq                                        #! 0x55555556f751 = 0x55555556f751;

{
and [
eqmod (inp_poly**2)
(L0x7fffffffaf80 + L0x7fffffffafa0*x) [3329, x**2 - (17)],
eqmod (inp_poly**2)
(L0x7fffffffafc0 + L0x7fffffffafe0*x) [3329, x**2 - (3312)],
...
(L0x7fffffffb15e + L0x7fffffffb17e*x) [3329, x**2 - (1175)]]
prove with 6 &&
and [
(-26632)@16 <s L0x7fffffffaf80, L0x7fffffffaf80 <s (26632)@16,
...
(-26632)@16 <s L0x7fffffffb17e, L0x7fffffffb17e <s (26632)@16]
prove with 6
}
```

The range portion of the ending condition says that every output limb is supposed to be between $\pm 8q (= 26632)$. The algebraic portion of the ending condition says that every two output coefficients make up a linear polynomial equal to the remainder of the entering polynomial modulo $X^2 - \zeta_i$, with each ζ_i an appropriate root of unity. The "`prove with`" is another compositional reasoning gadget (also see Section 3.1.3). We do not need any of the shorthands that express integers formed of multiple words and their arithmetic operations and algebraic relations in CryptoLine as they are not used here.

Finally we can run CryptoLine. It will obtain from the starting conditions and each CryptoLine instruction corresponding algebraic relations, then verify each *safety condition* using SMT solvers, then attempt to deduce the conclusions from the premises. It

does so by expressing each algebraic relation as an element in a polynomial ring (one which should be zero when the relation holds), then the algebraic part of the conclusions is also converted to polynomial ring elements, and a CAS reduces the ring element representing the conclusion using the ideal spanned by our collection of relations. If the reduction results in zero, then the verification is successful.

```
$ cv -v -isafety -jobs 24 -slicing -no_carry_constraint PQCLEAN_KYBER768_AVX2_polyvec_ntt.cl
Parsing Cryptoline file:                [OK]            0.089273 seconds
Checking well-formedness:               [OK]            0.031599 seconds
Transforming to SSA form:               [OK]            0.019121 seconds
Rewriting assignments:                  [OK]            0.020577 seconds
Verifying program safety:               [OK]            183.994889 seconds
Verifying range assertions:             [OK]            42.385435 seconds
Verifying range specification:          [OK]            200.594131 seconds
Rewriting value-preserved casting:      [OK]            0.001421 seconds
Verifying algebraic assertions:         [OK]            0.007455 seconds
Verifying algebraic specification:      [OK]            26.648724 seconds
Verification result:                    [OK]            453.802915 seconds
```

As shown in the depiction above, the verification has succeeded.

The Inverse NTT The inverse NTT Intel AVX2 implementation for Kyber is symmetric to the description above. The first 128 coefficients are first computed in inverse levels 6 to 1. The computation for the last 128 coefficients then follows. Finally, all 256 coefficients are computed in the inverse level 0. In Kyber inverse NTT, extra Montgomery reductions are needed to make coefficients representable in 16 bits to avoid over- or under-flow. Let $P_j = p_{j,0} + p_{j,1}X$ for $0 \leq j < 128$ be the 128 input polynomials for the inverse NTT.

We have the following

$$-q < p_{j,k} < q \text{ for all } 0 \leq j < 128, 0 \leq k < 2.$$

We specify the following mid-conditions at inverse level i for $6 \geq i > 0$, $0 \leq j < 127$:

$$H_{i,\lfloor j/2^{7-i} \rfloor} \equiv 2^{16-i} P_j \mod [q, X - \zeta_j]$$

Similarly, the mid-conditions for the last 128 coefficients the same but for $128 \leq j < 256$. Finally, Kyber inverse NTT has the following post-conditions

$$\overline{F} = H_{0,0} \equiv 2^{16} P_j \mod [q, X - \zeta_j]$$

and $-8q < h_{0,0,k} < 8q$ for $0 \leq k < 256$. *Note that the output polynomial has an extra factor of* 2^{16} *after the Kyber inverse NTT because the point multiplication is Montgomery, introducing an extra factor of* 2^{-16} *that needs balancing out.*

3.3 Differences on the Cortex-M4

The ARM Cortex-M4 is a microcontroller with usually no OS to run, so we must use a PC connected to a development kit (here, a STM32F429I-disc1). The `itrace.py` has support for Cortex-M4 that uses `gdb-multiarch`, the multi-architectural gdb. The translation rules differ, but it is otherwise the same process.

4 Verifying the Implementations

In each implementation of NTT multiplication, we set out what transformations/mappings are done and what butterflies are used, as well as pointing out any potential pitfalls. Section 3.2 already detailed the procedure to verify the NTT for Intel AVX2 Kyber.

4.1 Saber, Intel AVX2 implementation

Recall that Saber uses a module of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{Z}_q[x]/\langle X^n + 1\rangle$, with $q = 2^{13}$ and $n = 256$. For performing only a single polynomial multiplication it is usually advantageous to use an incomplete NTT but for Saber wherein the matrix-vector product the vector of polynomials only needs to be transformed once and the inner products can be computed in the NTT basis, a complete NTT is preferable.

The Intel AVX2 implementation uses prime moduli $q_0 = 7681$ and $q_1 = 10753$ for the NTTs of length 256 and maps:

$$R_{q_s} = \mathbb{Z}_{q_s}[X]/\langle X^{256} + 1\rangle \rightarrow \mathbb{Z}_{q_s}[X]/\langle X - \zeta_0\rangle \times \cdots \times \mathbb{Z}_{q_s}[X]/\langle X - \zeta_{255}\rangle$$

where $q_s \in \{q_0, q_1\}$. A polynomial multiplication over R_q is performed by the following steps. First, the implementation applies two complete 256-NTTs over R_{q_0} to the input polynomials, performs coefficient-wise multiplication, and then applies an inverse NTT over q_{q_0}. Second, the first step is repeated once but this time all operations are over R_{q_1}. Finally, the Chinese remainder theorem (CRT) is applied to obtain the multiplication result over R_q.

4.1.1 Forward NTT

The input of the NTT routine is a degree-256 polynomial with each coefficient ranging between $\pm q/2$. Let $F(X) = \sum_{k=0}^{255} f_k X^k \in \mathbb{Z}_q[X]$ be the input polynomial. We specify the following range preconditions

$$-q/2 \leq f_k < q/2, \text{ for all } 0 \leq k < 256$$

where the algebraic precondition is simply `true`.

The NTT routine first performs three levels (levels 0, 1, and 2) of CT butterflies, and then twists all the polynomials. This is followed by another three levels (levels 3, 4, and 5) of CT butterflies, and then all polynomials are twisted again. Finally, two additional CT butterflies (levels 6 and 7) are performed. Extra Montgomery reductions are applied when needed to make coefficients representable in 16 bits to avoid over- or under-flow. We detailed the postcondition and the mid-conditions in the following paragraphs.

Let $G_{i,j}(X) = \sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k} X^k \in \mathbb{Z}_{q_s}[X]$ be the polynomials at the end of level i for $0 \leq i \leq 7$ and $0 \leq j < 2^{i+1}$. Let $\zeta_{i,j}$ be the roots of unity in R_{q_s} at level i with $0 \leq i \leq 7$ and $0 \leq j < 2^{i+1}$. The first three levels map

$$\begin{aligned}
\mathbb{Z}_{q_s}[X]/\langle X^{256} + 1\rangle &\rightarrow \prod_{j=0}^{1} \mathbb{Z}_{q_s}[X]/\langle X^{128} - \zeta_{0,j}\rangle \\
&\rightarrow \prod_{j=0}^{3} \mathbb{Z}_{q_s}[X]/\langle X^{64} - \zeta_{1,j}\rangle \\
&\rightarrow \prod_{j=0}^{7} \mathbb{Z}_{q_s}[X]/\langle X^{32} - \zeta_{2,j}\rangle.
\end{aligned}$$

At the end of level i for $0 \leq i \leq 2$, we specify the algebraic mid-conditions for $0 \leq j < 2^{i+1}$:

$$F(X) \equiv G_{i,j}(X) \mod [q_s, X^{256/2^{i+1}} - \zeta_{i,j}].$$

Polynomials $G_{2,j}(X)$ are then twisted before the next three levels of CT butterflies.

Consider a polynomial $G_{2,j}(X) \in \mathbb{Z}_{q_s}[X]/\langle X^{32} - \zeta_{2,j}\rangle$ at the end of level 2. Let α_j be a 32-th root of $\zeta_{2,j}$. The polynomial is twisted by multiplying each coefficient $g_{2,j,k}$ with α_j^k based on the following mapping:

$$\mathbb{Z}_{q_s}[X]/\langle X^{32} - \zeta_{2,j}\rangle \rightarrow \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{32} - 1\rangle$$

with $X = \alpha_j Y_j$. Define

$$\zeta'_{i,j} = \begin{cases} 1 & \text{if } j = 0 \\ -1 & \text{if } j = 1 \\ \zeta_{i-1,j-2} & \text{if } i \geq 1 \text{ and } j \geq 2 \end{cases}$$

The one level CT butterfly in level 3 after twisting is based on the following mappings:

$$\mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{32} - 1\rangle \quad \rightarrow \quad \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle$$

where $0 \leq j < 8$. Thus we have

$$F(X) \equiv G_{3,j}(Y) \mod [q_s, X - \alpha_j Y_j, Y_j^{16} - \zeta'_{0,j \bmod 2}].$$

for $0 \leq j < 16$ at the end of level 3. Since polynomials over Y can be rewritten as polynomials over X based on the following mappings:

$$\begin{array}{rl} & \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle \\ \rightarrow & \mathbb{Z}_{q_s}[X]/\langle X^{16} - \alpha_j^{16}\zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[X]/\langle X^{16} - \alpha_j^{16}\zeta'_{0,1}\rangle. \end{array}$$

We have the algebraic mid-conditions at the end of level 3 for $0 \leq j < 16$:

$$F(X) \equiv G_{3,j}(\alpha^{-1}_{\lfloor j/2 \rfloor} X) \mod [q_s, X^{16} - \alpha^{16}_{\lfloor j/2 \rfloor}\zeta'_{0,j \bmod 2}].$$

Level 4 is based on the following mappings:

$$\mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle \quad \rightarrow \quad \prod_{t=0}^{3} \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{8} - \zeta'_{1,t}\rangle$$

where $0 \leq j < 8$. Again, polynomials over Y at the end of level 4 can be rewritten as polynomials over X based on the following mappings:

$$\prod_{t=0}^{3} \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{8} - \zeta'_{1,t}\rangle \quad \rightarrow \quad \prod_{t=0}^{3} \mathbb{Z}_{q_s}[X]/\langle X^{8} - \alpha_j^{8}\zeta'_{1,t}\rangle$$

We have the algebraic mid-conditions at the end of level 4 for $0 \leq j < 32$:

$$F(X) \equiv G_{4,j}(\alpha^{-1}_{\lfloor j/4 \rfloor} X) \mod [q_s, X^{8} - \alpha^{8}_{\lfloor j/4 \rfloor}\zeta'_{1,j \bmod 4}].$$

The CT butterfly in Level 5 is applied in the same way. In general, at the end of level i for $3 \leq i \leq 5$, we specify the algebraic mid-conditions for $0 \leq j < 2^{i+1}$:

$$F(X) \equiv G_{i,j}(\alpha^{-1}_{\lfloor j/2^{i-2} \rfloor} X) \mod [q_s, X^{256/2^{i+1}} - \alpha^{256/2^{i+1}}_{\lfloor j/2^{i-2} \rfloor}\zeta'_{i-3,j \bmod (2^{i-2})}].$$

Polynomials after level 5 are twisted again before the last two levels of CT butterflies. Let β_j be the 4-th root of $\zeta'_{2,j \bmod 8}$ for $0 \leq j < 64$. Similar to the twisting after level 2, each polynomial $G_{5,j}$ for $0 \leq j < 64$ is twisted by multiplying $g_{5,j,k}$ with β_j^k. For $6 \leq i \leq 7$ and $0 \leq j < 2^{i+1}$, we specify the algebraic mid-conditions:

$$F(X) \equiv G_{i,j}(\alpha^{-1}_{\lfloor j/2^{i-2} \rfloor}\beta^{-1}_{\lfloor j/2^{i-5} \rfloor} X) \mod [q_s, X^{256/2^{i+1}} - \alpha^{256/2^{i+1}}_{\lfloor j/2^{i-2} \rfloor}\beta^{256/2^{i+1}}_{\lfloor j/2^{i-5} \rfloor}\zeta'_{i-6,j \bmod 2^{i-5}}].$$

Specifically the algebraic mid-conditions after level 7 are

$$F(X) \equiv G_{7,j}(\alpha^{-1}_{\lfloor j/32 \rfloor}\beta^{-1}_{\lfloor j/4 \rfloor} X) \mod [q_s, X - \alpha_{\lfloor j/32 \rfloor}\beta_{\lfloor j/4 \rfloor}\zeta'_{1,j \bmod 4}]$$

for $0 \leq j < 256$. Define $\zeta_j = \alpha_{\lfloor j/32 \rfloor} \beta_{\lfloor j/4 \rfloor} \zeta'_{1,j \bmod 4}$. The algebraic postconditions specified are:

$$F(X) \equiv G_{7,j}(X) \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$.

The ranges of coefficients do not simply increase by q after each CT butterfly because of twisting polynomials after levels 2 and 5 and extra Montgomery reductions. Instead, ranges of coefficients are computed by the program `test_range256n` from the `ntt-polymul` repo (see footnote 1) and are asserted in the mid-conditions after each CT butterfly.

CryptoLine successfully verifies all the mid-conditions and the postconditions we specify for the NTT routine.

4.1.2 Inverse NTT

The inverse NTT Intel AVX2 implementation for Saber is symmetric. It first computes two layers of GS butterflies in inverse levels 7 to 6 followed by a twisting (at the end of level 6), and then three layers of GS butterflies in inverse levels 5 to 3 followed by another twisting (at the end of level 3). Finally, three layers of GS butterflies are computed in levels 2 to 0. Let $P_j = p_j$ for $0 \leq j < 256$ be the 256 input polynomials for the inverse NTT. The algebraic precondition for the inverse NTT routine is simply `true`. Let $H_{i,j}(X) = \sum_{k=0}^{256/2^i - 1} h_{i,j,k} X^k$ be the polynomials obtained at the end of inverse level i for $7 \geq i \geq 0$. We specify the mid-conditions at the end of inverse level 7:

$$2P_j \equiv H_{7,j}(\alpha^{-1}_{\lfloor j/32 \rfloor} \beta^{-1}_{\lfloor j/4 \rfloor} X) \mod [q_s, X - \zeta_j].$$

for $0 \leq j < 256$. Inverse level 6 contains one layer GS butterfly followed a twisting. At the end of inverse level i for $6 \geq i \geq 4$, the mid-conditions are:

$$2^{8-i} P_j \equiv H_{i,j}(\alpha^{-1}_{\lfloor j/32 \rfloor} X) \mod [q_s, X - \zeta_j].$$

for $0 \leq j < 256$. Inverse level 3 contains one layer GS butterfly followed by another twisting. At the end of level i for $3 \geq i \geq 0$, the mid-conditions are:

$$2^{8-i} P_j \equiv H_{i,j}(X) \mod [q_s, X - \zeta_j].$$

$0 \leq j < 256$. Define $\overline{F} = H_{0,0}$. The algebraic postconditions of the inverse NTT routine are then:

$$\overline{F} \equiv 2^{8-i} P_j \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$.

To speed up verification, algebraic mid-conditions at the ends of inverse levels 7 to 4 are actually removed because the algebraic mid-conditions at the end of inverse level 3 can be easily verified without any preceding mid-condition. Moreover, we apply the non-local compositional reasoning technique in inverse levels 3 to 0. Consider for example an inverse level i for $2 \geq i \geq 0$. Every modular equation at the end of inverse level i is only related to one modular equation at the end of inverse level $i + 1$. However, the mid-conditions at the end of inverse level $i + 1$ are specified in a single cut and thus, all of them are taken into account by computer algebra systems when proving a modular equation at the end of inverse level i. To improve performance, following the mid-conditions at the end of inverse level $i + 1$, we add one `cut` for each modular equation appearing in the mid-conditions. We are then able add one `prove with` to refer to the only one related modular equation in inverse level $i + 1$ for each modular equation to be verified in the mid-conditions at the end of inverse level i. Therefore hundreds of modular equations are

eliminated from the problems submitted to computer algebra systems. Such application of non-local compositional reasoning drastically reduces the verification time.

The ranges of coefficients again are computed by the program `test_range256n` and are asserted in the range mid-conditions after each GS butterfly.

CryptoLine successfully verifies all the mid-conditions and the postconditions except the range mid-conditions at the end of level 6. This failure is due to a mismatch of the extra reduction in level 6. The implementation of inverse NTT applies one extra reduction at the end of level 6 while the programmer's own range computation program `test_range256n` applies the extra reduction at the beginning of level 7. After the fix of the range computation program, we specify new ranges at the end of level 6 and then CryptoLine successfully verifies all the mid-conditions and the postconditions. Note that the program was correct; it was the programmer's range-checker that was wrong. As a result of our work, the range-checking tool was fixed in commit https://github.com/ntt-polymul/ntt-polymul/commit/7d88aa6b051bd076cc054eafd257c4ae8c10617c.

4.2 NTRU, ARM Cortex-M4 implementation (`ntruhps2048509`)

The `ntruhps2048509` M4 implementation leverages the following mapping, where ζ_i denote all the 256-th roots of unity in $\mathbb{Z}_{q'}$ with $q' = 1043969$:

$$\mathbb{Z}_{q'}[X]/\langle X^{1024}+1\rangle \to \mathbb{Z}_{q'}[X]/\langle X^4-\zeta_0\rangle \times \cdots \times \mathbb{Z}_{q'}[X]/\langle X^4-\zeta_{255}\rangle.$$

The implementation first transforms the polynomial via incomplete size-1024 NTT comprising 2 sets of 4-layer NTTs (CT butterflies), and performs each 4-coefficient multiplication (modulo a degree-3 polynomial) with schoolbook. Then it does 2 sets of 3-layer inverse NTTs (GS butterflies), followed by a final stage. The final stage consists of the following operations: 2 layers of inverse NTTs, taking $\mathrm{mod}(X^{509}-1)$, Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1}$ mod q', and reducing coefficients to the ring $\mathbb{Z}_q$ [CHK+21]. The constants `R` and $\mathtt{NTT}_\mathtt{N}$ are 2^{32} and 256, respectively. Coefficients in the implementation use the signed 32-bit representation.

4.2.1 Forward NTT

The input of the NTT routine is a degree-508 polynomial with each coefficient ranging between $\pm q$. Let $F = \sum_{k=0}^{508} f_k X^k$ be the input polynomial. The pre-conditions are

$$-q \le f_k < q, \text{ for all } 0 \le k \le 508.$$

The implementation first performs central reduction for each coefficient to normalize the range between $\pm q/2$ before NTT. The 8-level NTT is then computed in two phases: 4-layer NTTs from level 0 to level 3 are calculated first, followed by 4-layer NTTs from level 4 to level 7.

Define polynomial $\hat{F} = \sum_{k=0}^{508} \hat{f}_k X^k$ to be the result of the central reduction. Let $G_{i,j} = \sum_{k=0}^{1024/2^{i+1}-1} g_{i,j,k} X^k$ denote the polynomials obtained at the end of level i, where $0 \le i \le 7$, $0 \le j < 2^{i+1}$, and $\zeta_{i,j}$ with $0 \le j < 2^{i+1}$ the roots of unity at level i. The output polynomials of the NTT routine are therefore $G_{7,j} = \sum_{k=0}^{3} g_{7,j,k} X^k$ with $0 \le j < 256$. The post-conditions to be verified are

$$F \equiv \hat{F} \mod q \qquad \text{and} \qquad \hat{F} \equiv G_{7,j} \mod [q', X^4-\zeta_{7,j}], \text{ for all } 0 \le j < 256$$

with ranges

$$-128q' < g_{7,j,k} < 128q', \text{ for all } 0 \le j < 256, 0 \le k < 4.$$

The first part of the algebraic post-conditions represents the correctness of the central reduction, while the second part specifies the correctness of the 8-level NTT. CryptoLine

successfully verifies the correctness of the NTT routine with respect to the aforementioned pre- and post-conditions.

In order to improve the verification efficiency, we further utilize the compositional reasoning mechanism provided by the `cut` instruction. As the 8-level NTT is clearly demarcated into two phases, we specify the following mid-conditions at the end of the first phase (level 3) to split the whole verification problem into two smaller sub-problems:

$$F \equiv \hat{F} \mod q \qquad \text{and} \qquad \hat{F} \equiv G_{3,j} \mod [q', X^{64} - \zeta_{3,j}], \text{ for all } 0 \le j < 16 \tag{1}$$

with ranges

$$-5q' < g_{3,j,k} < 5q', \text{ for all } 0 \le j < 16, 0 \le k < 64.$$

In fact, we divide the sub-problems into even smaller pieces to achieve more efficiency, thanks to the non-local compositional reasoning feature supported by the `cut` instruction. For example in the first phase, the 4-layer NTTs are performed iteratively. Each iteration only transforms 16 coefficients. We thus insert the following mid-conditions at the end of the e-th iteration for $0 \le e < 64$ to specify the computation of that iteration:

$$\begin{aligned} f_{64k+e} &\equiv \hat{f}_{64k+e} \mod q, \text{ for all } 0 \le k < 16 \\ \text{and} \qquad g_{3,j,e} X^e &\equiv \sum_{k=0}^{15} \hat{f}_{64k+e} X^{64k+e} \mod [q', X^{64} - \zeta_{3,j}], \text{ for all } 0 \le j < 16, \end{aligned} \tag{2}$$

where we assume $f_k = \hat{f}_k = 0$ when $k > 508$. Accordingly, we use the "`prove with`" extension of `cut` in the mid-conditions (1) to add mid-conditions (2) as extra premises to ease the verification.

4.2.2 Inverse NTT

The inverse NTT routine consists of three phases. Phases I and II transform all 1024 coefficients by 3-layer GS butterflies from inverse levels 7 to 5 and from inverse levels 4 to 2, respectively. In phase III, 2-layer inverse NTTs from inverse levels 1 to 0 are performed iteratively, with 4 coefficients at each iteration. In the same iteration, for each resulting coefficient, the mapping $\mathbb{Z}_{q'}[X]/\langle X^{1024} + 1\rangle \to \mathbb{Z}_{q'}[X]/\langle X^{509} - 1\rangle$ is calculated immediately, followed by Montgomery multiplication by the factor $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1} \bmod q'$ and finally reduction to the ring $\mathbb{Z}_q$.

To formalize appropriately the post-conditions, we denote several polynomials by the following notations:

- $P_j = \sum_{k=0}^{3} p_{j,k} X^k$ with $0 \le j < 256$, the input polynomials for the inverse NTT routine;
- $\overline{F} = \sum_{k=0}^{1023} \overline{f}_k X^k$, the polynomial obtained at the end of 8-level inverse NTT;
- $F^* = \sum_{k=0}^{508} f_k^* X^k$, the remainder polynomial after taking $\bmod (X^{509} - 1)$ and Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1} \bmod q'$ in phase III;
- $\tilde{F} = \sum_{k=0}^{508} \tilde{f}_k X^k$, the output polynomial of the inverse NTT routine.

The post-conditions to be verified are therefore specified as follows:

$$\overline{F} \equiv 256 P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \le j < 256 \tag{3}$$

$$\mathtt{NTT}_\mathtt{N} F^* \equiv \mathtt{R}\overline{F} \mod [q', X^{509} - 1] \tag{4}$$

$$\tilde{F} \equiv F^* \mod q \tag{5}$$

with appropriate ranges. Conditions (3) constitute the correctness of the 8-level inverse NTT, while condition (4) ensures the correctness of both the modulo operation by $X^{509}-1$ and Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1} \bmod q'$ in phase III. Condition (5) means the final reduction is correct.

Similarly, we construct mid-conditions to make the verification more efficient, thanks to the clear three-phase structure of the implementation. Let $H_{i,j} = \sum_{k=0}^{1024/2^i-1} h_{i,j,k}X^k$ be the polynomials obtained at the end of inverse level i with $7 \geq i \geq 0$ and $0 \leq j < 2^i$. Then we have the following mid-conditions at the end of phase I (inverse level 5)

$$H_{5,\lfloor j/8 \rfloor} \equiv 2^3 P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256;$$

and the following mid-conditions at the end of phase II (inverse level 2)

$$H_{2,\lfloor j/64 \rfloor} \equiv 2^6 P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256.$$

Moreover, we define more refined mid-conditions in a similar way to the verification of the NTT routine, since the phases in the inverse NTT routine are also implemented by iterations. We refer the interested readers to the supplementary material for the detailed mid-conditions that have been used.

Interestingly, when verifying the post-condition (4), CryptoLine reports "failed". This post-condition corresponds to the correctness of both the modulo operation by $X^{509}-1$ and Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1} \bmod q'$ in phase III. The failure indicates either that these computations are flawed, or that the computations are correct yet CryptoLine is not able to verify with existing premises. After inspecting the error and related code, we found that the following modular equation can be verified with CryptoLine:

$$\mathtt{NTT}_\mathtt{N} f_2^* \equiv \mathtt{R}(\overline{f}_2 + \overline{f}_{511} + \overline{f}_{1017}) \mod [q'].$$

Nevertheless, note that condition (4) requires the following modular equation:

$$\mathtt{NTT}_\mathtt{N} f_2^* X^2 \equiv \mathtt{R}(\overline{f}_2 X^2 + \overline{f}_{511} X^{511} + \overline{f}_{1020} X^{1020}) \mod [q', X^{509}-1].$$

Since $X^{1017} \not\equiv X^2 \mod [X^{509}-1]$, the code does not appear to calculate the coefficient f_2^* correctly. Yet the problem evades all test inputs. There must be a simple explanation.

It turns out that additional premises are needed to verify post-condition (4). Recall that the inverse NTT routine is only for `ntruhps2048509`. As a part of NTT multiplication between two degree-508 polynomials, the modulo operation by $X^{509}-1$ in phase III will take as input a polynomial of degree less than 1017. Thus $\overline{f}_k = 0$ for $1017 \leq k \leq 1023$ in this context. Since $\overline{f}_{1020} = \overline{f}_{1017} = 0$, we have

$$\overline{f}_2 X^2 + \overline{f}_{511} X^{511} + \overline{f}_{1020} X^{1020} \equiv \overline{f}_2 X^2 + \overline{f}_{511} X^{511} + \overline{f}_{1017} X^{1017} \mod [q', X^{509}-1].$$

The routine is correct only if it is used in `ntruhps2048509`! With the observation, we add these assumptions with the following instructions:

$$\mathsf{assume}\ \overline{f}_k = 0 \ \&\&\ \mathsf{true}, \text{ for all } 1017 \leq k \leq 1023.$$

Then CryptoLine successfully verifies all the post-conditions. These assume's illustrate that the inverse NTT routine in question, in particular the modulo operation by $X^{509}-1$ in phase III, is not generally correct. It is correct when being a part of NTT multiplication between two degree-508 polynomials. CryptoLine forces the verifier to specify precisely all the premises required to show correctness, hence helps the programmer and the users to better understand both the generality and limitations of the code.

4.3 Other implementations

As aforementioned in Section 1.1, verification has been carried out on six chosen NTT implementations. We have explained the details on how to verify the three of them, including the AVX2 NTT implementations for KYBERand SABER, and the Cortex-M4 NTT implementation for NTRU. Although the remaining implementations are optimized differently, they are built with basic blocks such as CT/GS butterflies, twisting and Montgomery reductions that we have seen already. The techniques to construct the verification conditions are similar. We demonstrate briefly the primary verification conditions for them in the following. The details of all the conditions employed can be found in our supplementary material.

4.3.1 KYBER, ARM Cortex-M4 implementation (KYBER768)

The KYBER M4 implementation from `pqm4`[2] maps

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

with ζ_j for the primitive 256-th roots of unity. The 7-level NTT is structured with 2 sets of 3-layer CT butterflies and then a set of 1-layer CT butterflies followed by Barrett reductions. The inverse NTT is symmetric with GS butterflies.

For the forward NTT routine, let $F = \sum_{k=0}^{255} f_k X^k$ be the input polynomial, $G_{i,j} = \sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k} X^k$ the polynomials at the end of level i with $0 \le i \le 6$ and $0 \le j < 2^{i+1}$, and $\zeta_{i,j}$ be the roots of the unity at the end of level i. As for the inverse, $P_j = p_{j,0} + p_{j,1}X$ $(0 \le j < 128)$ denote the 128 input polynomials, and $H_{i,j} = \sum_{k=0}^{256/2^i-1} h_{i,j,k} X^k$ for the polynomials at the end of inverse level i with $6 \ge i \ge 0$ and $0 \le j < 2^{i+1}$, where the output polynomial $\overline{F} = H_{0,0}$.

The range pre-condition $-q \le f_k < q$ is used for each coefficient f_k with $0 \le k < 256$ when verifying the NTT routine. We specify the following algebraic mid-conditions at the end of levels $i = 2$ and $i = 5$:

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}], \text{ for all } 0 \le j < 2^i.$$

The above equations with $i = 6$ are the algebraic post-conditions to be verified. On the other hand, the range post-conditions are $0 \le g_{6,j,k} \le q$ due to Barrett reductions.

For the inverse NTT, the algebraic mid-conditions inserted at the end of inverse levels $i = 6$ and $i = 3$ become

$$H_{i,\lfloor j/2^{7-i} \rfloor} \equiv 2^{7-i} P_j \mod [q, X^2 - \zeta_j], \text{ for all } 0 \le j < 128.$$

Finally, the algebraic post-conditions at the end of inverse level 0 are

$$\overline{F} \equiv 2^{16} P_j \mod [q, X^2 - \zeta_j], \text{ for all } 0 \le j < 128$$

with an extra factor 2^9 being the effect of Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_\mathtt{N}^{-1}$. The range mid-conditions and post-conditions are all $-q \le h_{i,j,k} < q$ for each coefficient $h_{i,j,k}$.

4.3.2 SABER, ARM Cortex-M4 implementation

The implementation from [ACC+22] maps

$$\mathbb{Z}_q[X]/\langle X^{256}+1\rangle \to \mathbb{Z}_q[X]/\langle X^4-\zeta_0\rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^4-\zeta_{63}\rangle.$$

Thus the NTT routine performs 2 sets of 3-layer NTTs via CT butterflies. To use CT butterflies as well in the inverse NTT, the mapping is rewritten as follows:

$$\begin{aligned}
\mathbb{Z}_q[X]/\langle X^{256}+1\rangle &\to \mathbb{Z}_q[X,Y]/\langle X^4-Y\rangle\langle Y^{64}+1\rangle \\
&\overset{Y=\zeta Y_{0,0}}{\to} \mathbb{Z}_q[X,Y_{0,0}]/\langle X^4-\zeta Y_{0,0}\rangle\langle Y_{0,0}^{64}-1\rangle \\
&\ \ \vdots \\
&\to \prod_{j=0}^{63}\mathbb{Z}_q[X,Y_{6,j}]/\langle X^4-\zeta_j Y_{6,j}\rangle\langle Y_{6,j}-1\rangle \\
&\to \prod_{j=0}^{63}\mathbb{Z}_q[X]/\langle X^4-\zeta_j\rangle
\end{aligned}$$

where $Y_{i,j}$ are the fresh variables introduced by the i-th twisting. The twisted inverse NTT routine therefore consists of 2 sets of 3-layer CT butterflies, followed by a twisting mixed with Montgomery multiplication by $\mathtt{R}^2\mathtt{NTT}_{\mathtt{N}}^{-1}$, and a central reduction at last.

For the forward NTT, let $F=\sum_{k=0}^{255} f_k X^k$ again be the input polynomial, $G_{i,j}=\sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k}X^k$ the polynomials at the end of level i for $0\le i\le 5$ and $0\le j<2^{i+1}$. For the inverse routine, define $P_j=\sum_{k=0}^{3} p_{j,k}X^k$ $(0\le j<64)$ as the 64 input polynomials, $H_{i,j}=\sum_{k=0}^{256/2^i-1} h_{i,j,k}X^{(k \bmod 4)}Y_{i,j}^{\lfloor k/4\rfloor}$ with $0\le j<2^i$ the polynomials obtained at the end of inverse level i $(5\ge i\ge 0)$, and $\overline{F}=\sum_{k=0}^{255}\overline{f}_k X^k$ the output polynomial of the inverse NTT routine.

As standard NTTs, the NTT routine should satisfy

$$F\equiv G_{i,j} \mod [q, X^{256/2^{i+1}}-\zeta_{i,j}], \text{ for } 0\le j<2^{i+1}, \text{ and } -(i+2)q\le g_{i,j,k}<(i+2)q, \quad (6)$$

at the end of level i. The post-conditions are condition (6) with $i=5$, and the instances when $i=2$ are inserted as mid-conditions at the end of level 2 for verification efficiency.

As for the inverse routine, the following mid-conditions are specified at the end of inverse level 3:

$$H_{3,\lfloor j/8\rfloor}\equiv 2^3 P_j \mod [q, X^4-\zeta_j Y_{6,j}, Y_{6,j}-1], \text{ for } 0\le j<64, \text{ and } -8q\le h_{3,j,k}<8q.$$

Before post-conditions, the following algebraic mid-conditions are inserted:

$$\overline{F}\equiv \mathtt{R}P_j \mod [q, X^4-\zeta_j Y_{6,j}, Y_{6,j}-1], \text{ for all } 0\le j<64.$$

Note that, unlike Section 4.1, we did not eliminate the variables Y's in the above mid-conditions when dealing the twisting. CryptoLine allows both ways of formulating the conditions. Finally, the algebraic post-conditions are verified:

$$\overline{F}\equiv \mathtt{R}P_j \mod [q, X^4-\zeta_j], \text{ for all } 0\le j<64,$$

with explicitly instantiating $Y_{6,j}$ with 1 using `assume`'s for $0\le j<64$ before to prove the algebraic post-conditions. Because of central reductions at the end, the range post-conditions are $-q/2\le\overline{f}_k<q/2$ for all output coefficients $\overline{f}_k$.

4.3.3 NTRU, Intel AVX2 implementation (`ntruhps2048509`)

The `ntruhps2048509` AVX2 implementation from [CHK+21] maps

$$\mathbb{Z}_q[X]/\langle X^{1024}-1\rangle\to\mathbb{Z}_q[X]/\langle X^2-\zeta_0\rangle\times\cdots\times\mathbb{Z}_q[X]/\langle X^2-\zeta_{511}\rangle,$$

with ζ_i again ranging over all the primitive 512-th roots of unity. Both the 9-level NTT and inverse NTT are implemented layer by layer.

As usual, for the forward NTT routine, we use $F = \sum_{k=0}^{511} f_k X^k$ to denote the input polynomial of degree 511, $G_{i,j} = \sum_{k=0}^{1024/2^{i+1}-1} g_{i,j,k} X^k$ for the polynomial obtained at the end of level i with $0 \leq i \leq 8$ and $0 \leq j < 2^{i+1}$, and $\zeta_{i,j}$ with $0 \leq j < 2^{i+1}$ for the roots of unity at level i. As for the inverse NTT routine, $P_j = p_{j,0} + p_{j,1} X$ $(0 \leq j < 512)$ represent the input polynomials, $H_{i,j} = \sum_{k=0}^{1024/2^{i}-1} h_{i,j,k} X^k$ the resulting polynomials at the end of inverse level i with $8 \geq i \geq 0$ and $0 \leq j < 2^i$, and $\overline{F} = \sum_{k=0}^{1023} \bar{f}_k X^k$ for the output polynomial. In this implementation, $\overline{F} = H_{0,0}$.

As standard NTTs, the NTT routine should satisfy

$$F \equiv G_{i,j} \mod [q, X^{1024/2^{i+1}} - \zeta_{i,j}], \text{ for all } 0 \leq j < 2^{i+1} \tag{7}$$

at the end of level i. Thus the algebraic post-conditions to be verified are

$$F \equiv G_{8,j} \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512.$$

On the other hand, the inverse NTT routine should satisfy

$$H_{i,\lfloor j/2^{9-i} \rfloor} \equiv 2^{9-i} P_j \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512 \tag{8}$$

at the end of inverse level i. The algebraic post-conditions of the inverse NTT routine are

$$\overline{F} \equiv 2^9 P_j \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512.$$

Range conditions of the coefficients are computed by the program `test_range1024` from the `ntt-polymul` repository (see footnote 1).

For efficiency, mid-conditions (7) are only added at the end of level 2 when verifying the NTT routine, while mid-conditions (8) are only inserted at the end of inverse level 3 for the inverse NTT routine. Moreover, since the implementation divides the 1024 coefficients into 8 parts and calculates coefficients in each part separately, more refined mid-conditions are also used to further reduce verification time.

5 Results

We use CryptoLine to verify the Intel AVX2 and Cortex M4 assembly implementations for the NTTs for Kyber, NTRU, and Saber. All experiments are running on an Ubuntu 20.04.3 server with 3.2GHz Intel Xeon and 1TB RAM. Table 3 shows the verification time for each instance in seconds. In the table, the column *algebra* shows the time for verifying algebraic properties; *overflow* gives the time for checking over- and under-flow; *range* contains the time for range checks; and *total* is the total running time for the instance. All time is in seconds.

Verification time varies drastically among the experiments. Consider, for instance, the experiments of the Intel AVX2 implementation for Kyber NTT. The total verification time for inverse NTT is about 16.7 times slower than those for NTT. From Table 3, we see that the time for overflow and range checking is drastically different in both instances. In our verification, coefficient ranges are specified and verified for each level in NTT. On the other hand, coefficient ranges are only specified and verified for the inverse levels 1 and 0 in inverse NTT. Range checking is thus divided into 7 subtasks in NTT whereas it is divided into two in inverse NTT. Compositional reasoning divides large verification tasks into smaller tasks. In Intel AVX2 Kyber NTT and inverse NTT implementations, we observe significant differences in their verification time.

To evaluate the effectiveness of compositional reasoning with cuts, we compare verification time of Intel AVX2 Kyber NTT by numbers of cuts (Figure 5). The NTT

Table 3: Verification Results (in seconds)

KEM	*architecture*	*direction*	*algebra*	*overflow*	*range*	*total*
Kyber768	AVX2	normal	26.6	183.9	242.8	453.8
		inverse	761.7	781.0	6050.0	7593.5
	Cortex M4	normal	134.3	173.7	191.0	499.4
		inverse	1481.0	348.6	184.1	2014.3
`ntru2048509`	AVX2	normal	478.4	1229.8	1738.6	3447.8
		inverse	3868.6	1545.3	12170.3	17585.7
	Cortex M4	normal	1353.0	5970.7	4810.2	12135.2
		inverse	11315.1	3019.6	7813.7	22150.9
Saber	AVX2	normal	60.1	207.7	271.7	539.9
		inverse	436.2	443.8	859.4	1741.0
	Cortex M4	normal	110.2	2731.9	2196.7	5039.3
		inverse	3250.5	2754.0	853.4	6858.8

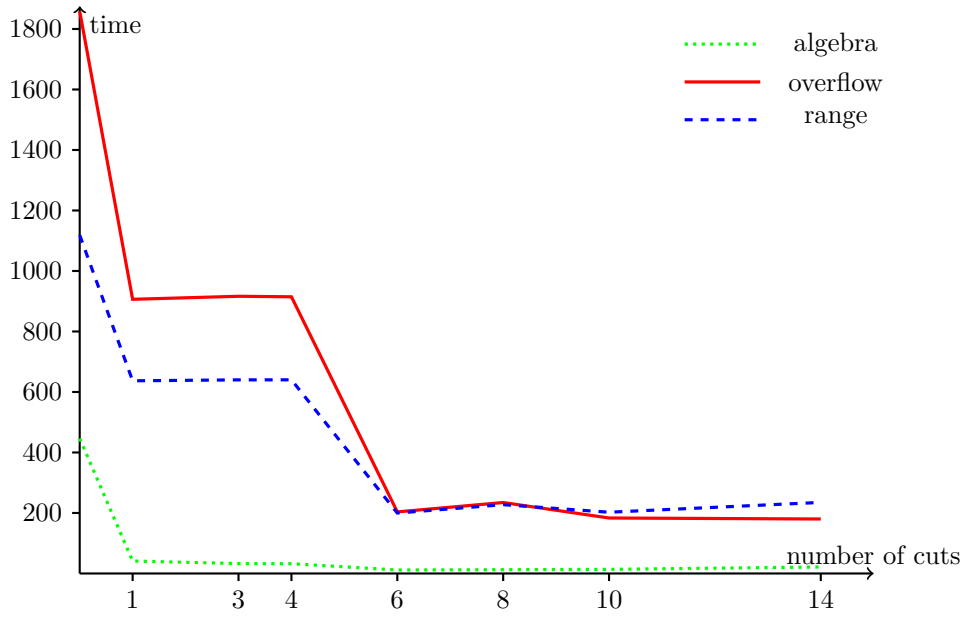


Figure 5: Effectiveness of Cuts in Intel AVX2 Kyber NTT

implementation is divided by different numbers of cuts in the figure. The verification time for algebraic properties is drawn in the dotted green line. The time for overflow checking is in the solid red line. And the time for range checks is the dashed blue line. The Intel AVX2 Kyber NTT implementation in Table 3 uses 14 cuts. It corresponds to the rightmost values in Figure 5. From the figure, we see the monolithic verification time without cuts is the worst. Adding one cut improves the verification time in all categories significantly. The verification time is similar to one, three, or four cuts in our experiments. However, it improves significantly again with six or more cuts. Most interestingly, the best verificaiton time is with 10 cuts. Adding more cuts in fact increases the verification time slightly.

Figure 5 shows that compositional reasoning is better than monolithic verification. The verification time can be reduced by 50% with a single cut. Our experiments also point out limitations of compositional reasoning. First, not all decompositions are effective for verification. Adding more cuts may not improve verification time significantly. Verifiers still need to decide how to divide verification tasks more effectively. Second, extreme decompositions may be harmful. Compositional reasoning necessarily induces overhead. In the extreme case, benefits of compositional reasoning can be nullified by its overhead. Compositional reasoning does not always improve verification time.

Among the three KEM lattice finalists, the verification time for `ntru2048509` is much longer than the others. This is because it considers input polynomials of degree 1024 and performs 10-level NTT. Kyber and Saber have input polynomials of degree 256 with 7- and 8-level NTT respectively. In all cases, inverse NTT implementations always take more time to verify. Recall that NTT always has input polynomials of (very) high degrees and output polynomials of degrees 0 or 1. Subsequently, mid-conditions become simpler at each level of NTT computation. At the last level, computer algebra systems only need to verify modular equations over linear or constant polynomials. Inverse NTT however has the opposite pattern. At each level of inverse NTT computation, mid-conditions become more complex. In the end, computer algebra systems need to verify modular equations over high-degree polynomials. The verification time for algebraic properties is much longer in inverse NTT than those in NTT. Differences in overflow or range checking between NTT and inverse NTT are not so pronounced. Rather, they depend more on the number of cuts. For well-decomposed inverse NTT implementations, their overflow or range checking time can be less than corresponding NTT implementations.

We should also mention the effects of our modifications to CryptoLine. Without non-local cutting, it is not possible to cut Kyber at each level because of the structure of the NTT, in which half of the coefficients are used for most layers; as a result variables move in and out of registers. Without ghosts variables (which enable non-local cuts), one can only relate to the last cut. So effectively the only possibility of cutting is somewhere in the code where all variables are written out to memory, which only happens after layers 0 and 6 for Kyber (this is program-dependent).Initially, without the non-local compositional reasoning, we tried verifying Kyber (the smallest of the programs verified) and it took 8× as much time as with the new extension. In NTRU, with its larger state, Singular choked due to the size of the ideal — on a server with 1TB of RAM.

Human time: Perhaps more important than computer clock time is *human time*. Each of our verifications took less than a week of calendar time, and the majority of it was really communication with the programmer of the code, and secondly reading and gaining a basic understanding of the program at hand. We take this opportunity to note that in no case was the verifier the programmer of the code, although in all cases the programmer either provided very good annotations or was cooperative in resolving any questions that arose.

Conclusion: We demonstrate the feasibility for a programmer to verify his or her high-speed assembly code for PQC, as well as for a verification specialist to verify someone else's high-speed PQC software in assembly code, with some cooperation from the programmer.

Many algorithms in cryptography have clearly demarcated stages. One clear take-away point is that in order to verify such algorithms, enhanced compositional reasoning techniques that take full advantage of such structures is needed. We try to provide this requisite enhanced compositional reasoning with new cuts and Ghost variables functionality.

Future Work: The six instances in this work are just the beginning. The same technique applies to also any implementation of small ideal-lattice-based cryptosystems that also has NTT-based arithmetic, e.g., the KEMs NTRU Prime, LAC, or NewHope [LLZ+18, PAA+19, BBC+20] and the signatures Dilithium and Falcon [ABD+20a, FHK+17]. There are also a myriad of other architectures and other parameter sets to consider.

We could also envision extending CryptoLine to other PQCs such as Rainbow/UOV and Classic McEliece. Ideally, We would hope that CryptoLine and similar tools would make it safe to deploy high-speed custom-made assembly for PQC in production scenarios.

Acknowledgments

The authors in Taiwan are partially funded from the Ministry of Science and Technology grants MOST108-2221-E-001-010-MY3, MOST110-2221-E-001-008-MY3, the Sinica Investigator Award AS-IA-109-M01, the Data Safety and Talent Cultivation Project AS-KPQ-109-DSTCP, the Intel Fast Verified Postquantum Software Project, and the Cybersecurity Center of Excellence Project at National Applied Research Labs. The authors in Shenzhen University are funded by the National Natural Science Foundation of China (62002228), the Natural Science Foundation of Guangdong Province (2022A1515010880), and Shenzhen Science and Technology Innovation Commission (JCYJ20210324094202008, 20200810045225001).

References

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. NISTIR8309 – status report on the second round of the nist post-quantum cryptography standardization process, July 2020. https://doi.org/10.6028/NIST.IR.8309.

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823. ACM, 2017.

[ABD+20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/.

[ABD+20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien

Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/kyber/.

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in NTRU prime comparison of optimization strategies on Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):217–238, 2021. https://doi.org/10.46586/tches.v2021.i1.217-238.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J Kannwischer, and Bo-Yin Yang. Multi-moduli ntts for saber on cortex-m3 and cortex-m4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):127–151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292.

[Aff13] 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.

[AM07] 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.

[ANY12] 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.

[App15] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, 2015.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntruprime.cr.yp.to/.

[BBF+21] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. Easypqc: Verifying post-quantum cryptography. Cryptology ePrint Archive, Report 2021/1253, 2021. https://ia.cr/2021/1253.

[BCS08] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In D. Wagner, editor, *Advances in Cryptology (CRYPTO)*, LNCS, pages 221–240. Springer, 2008.

[BCS16] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. *J. Cryptol.*, 29(4):775–805, 2016.

[BPYA15] 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.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://ntru.org/.

[CHK+21] 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.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[EPG+19] 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.

[FHK+17] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2017. https://pq-crystals.org/dilithium/.

[FLS+19] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1606. ACM, 2019.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO '99*, volume 1666, pages 537–554, 1999. http://dx.doi.org/10.1007/3-540-48405-1_34.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009. https://doi.org/10.1137/070711761.

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: For Fun and Profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. Association for Computing Machinery. https://doi.org/10.1145/1464291.1464352.

[HHK17] 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.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory – ANTS-III*, pages 267–288, 1998. http://dx.doi.org/10.1007/BFb0054868.

[HVDH21] David Harvey and Joris Van Der Hoeven. Integer multiplication in time o (n log n). *Annals of Mathematics*, 193(2):563–617, 2021.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical ring-lwe based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, 2018. https://eprint.iacr.org/2018/1009.

[LST+19] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic in cryptographic c programs. In Julia Lawall and Darko Marinov, editors, *IEEE/ACM International Conference on Automated Software Engineering*, pages 552–564. IEEE, 2019.

[MC13] 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.

[MG07] 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.

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

[PAA+19] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[PQC21] PQClean. The PQClean project. https://github.com/PQClean/PQClean, 2021.

[PTWY18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives. In Sven Schewe and Lijun Zhang, editors, *International Conference on Concurrency Theory*, LIPIcs, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[SXY18] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 520–551. Springer, 2018.

[TWY17] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1973–1987. ACM, 2017.

[YGS+17] 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.

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017.