

# Multiplying Not-So-Big Integers with FFTs:

## Finding and Pushing the Crossover on Large Modern OoO ARM CPUs

Cesare Huang<sup>1</sup>, Daisy Meng-Wei Liu<sup>1,2</sup>, David Shu-Yu Wu<sup>1,3</sup>,  
Bow-Yaw Wang<sup>1</sup> and Bo-Yin Yang<sup>1</sup>

<sup>1</sup> IIS and CITI, Academia Sinica, Taipei, Taiwan,

[by\(w|y\)ang@iis.sinica.edu.tw](mailto:by(w|y)ang@iis.sinica.edu.tw), [cesarehuang@icloud.com](mailto:cesarehuang@icloud.com), [daisyliu0225@gmail.com](mailto:daisyliu0225@gmail.com)

<sup>2</sup> National Tsing-Hua University, Hsinchu, Taiwan

<sup>3</sup> National Taiwan University, Taipei, Taiwan [b12902036@ntu.edu.tw](mailto:b12902036@ntu.edu.tw)

**Abstract.** Established multiple-precision software generally reserves Fast-Fourier Transform (FFT) multiplication for very large operands; for example, GMP reports full-product FFT thresholds of roughly 3000–10000 limbs [GtGdt23a, GtGdt23b]. Becker et al. [BHK<sup>+</sup>22a] showed that Number-Theoretic Transform (NTT) multiplication can become advantageous much earlier on Cortex-M microcontrollers. It thus becomes an interesting engineering question to check where the crossover actually takes place on a big modern CPU. Our first-generation implementation on Cortex-A76 and Neoverse-N1 shows that the crossover point is around 10k bits when using ARMv8+ with Neon, using optimized and verified code. Motivated by the tighter Barrett bound of Becker [Bec26], we developed the Second implementation, which eliminates additional reductions and pushes the crossover below 5.6k bits. We integrate our multipliers into OpenSSL RSA, resulting in end-to-end speedup for the public-key operations.

**Keywords:** Fast Fourier Transform, Big Integer Multiplications, Unsaturated Limbs

## 1 Introduction

Multi-precision integer multiplication is the arithmetic core of public-key cryptography, directly affecting the efficiency of schemes such as RSA [RSA78], Diffie-Hellman [DH76], and those in isogeny-based cryptography. Take CSIDH [CCSCD<sup>+</sup>24] as an example: its reference implementation directly calls GMP with 2048-9216-bit primes. For RSA, implementations in OpenSSL [Ope26] commonly use iterative multi-precision Montgomery multiplication to perform modular reduction efficiently [Mon85]. Both libraries represent large integers as arrays of machine-word limbs, multiplying through hand-optimized assembly kernels and algorithms tailored to specific operand sizes. Multiplication algorithms such as FFT and NTT have traditionally been considered advantageous only for very large operands. However, recent work on NTT-based multiplication for somewhat small integers [BHK<sup>+</sup>22a] showed that this threshold can be substantially lower on microcontrollers and concluded that NTT-based modular arithmetic should be considered from 2048 bits onward, indicating the practical crossover point varies with the target microarchitecture and implementation.

NIST’s post-quantum cryptography standardization effort [Nat24b, Nat24a] has driven extensive optimization of lattice-based schemes. Because their polynomial arithmetic relies heavily on the NTT, substantial work has optimized NTT variants for different polynomial rings, primes, and even input conditions [BHK<sup>+</sup>22b, CHK<sup>+</sup>21, ACC<sup>+</sup>21, Hwa24, CYW25]. These techniques can also benefit classical large-integer arithmetic, which reduces to polynomial multiplication, provided that coefficient reconstruction, carry handling, and

range constraints are carefully managed. It is therefore natural to ask where this crossover occurs on modern ARMv8-A processors with Neon instructions.

**Related Work** As a highly-optimized multi-precision library, GMP reports full-product FFT thresholds in the thousands of limbs [GtGdt23a, GtGdt23b]. For smaller operands, it typically uses Toom-Cook, Karatsuba, and schoolbook multiplications. Our measurements indicate that transform-based multiplication can become competitive well below GMP’s tuned internal FFT threshold. For the implementation side, the product coefficients may exceed the range that can be uniquely reconstructed from a single NTT prime, motivating the use of multiple NTT moduli [ACC<sup>+</sup>21] in a Residue Number System (RNS). More recently, [Bec26] provides a precise pointwise output bound for Barrett multiplication, improving the traditional bound to inside a single prime in signed arithmetic.

From another perspective, random testing cannot establish algebraic correctness and tight range invariants over all inputs, such as an NTT implementation with aggressive lazy reduction. Therefore, machine-checked formal verification of optimized NTT assembly using CryptoLine was developed in [HLS<sup>+</sup>22], with subsequent work improving algebraic equivalence and range analysis [CLT<sup>+</sup>25] of certain modular multiplications. Furthermore, instead of manually handling instruction scheduling and register allocation to reduce latency in the optimized assembly, SLOTHY [ABKK24, slo] converts the problem into CSPs with instruction latencies and throughput corresponding for specific architectures. Logical-equivalence checking of cryptographic assembly is further studied in [LLS<sup>+</sup>24, Tri26], which enables us to check the bit-exact equivalence between versions before and after SLOTHY.

Our work extends these directions to the crossover finding, OpenSSL integration, and end-to-end verification on out-of-order ARMv8-A processors, which is our first-generation implementation. CryptoLine does not scale well to construct high-level algebraic equivalence with the multiplication, so we use HOL Light [Har09] with the assumed correctness of the lower-level kernels verified by CryptoLine. Moreover, the tighter bound permits additional modular reductions to be eliminated and motivates the **Second** implementation. We refer to the original implementation as **First** and the later revision as **Second** throughout the paper.

Our paper proposes the following three contributions.

1. We provide an empirical assessment of our constant-time NTT-based large-integer multipliers on several out-of-order(OoO) processors such as A72, A76, N1, V1, and V2. Compared with GMP’s variable-time implementation, we identify a crossover point below 5632 bits on Cortex-A76 and Neoverse-N1 in our **Second** generation implementation.
2. To deal with oversights in instruction scheduling, carry propagation, modular reduction, or range assumptions, the **First** generation implementation of this work provides formal assurance that our optimized implementation follows its algebraic specifications under the required range constraints. Moreover, we handle low-level assembly with its algebraic / range contracts and equivalence check with CryptoLine and high-level composition with HOL Light.
3. We integrate our multipliers into OpenSSL RSA and evaluate the resulting end-to-end performance. On Cortex-A76 and Neoverse-N1, the **Second** generation implementation reduces the cycle count of RSA-8192 verification and encryption by about 37%, while the **First** generation RSA-10240 implementation reduces them by about 27–28%. These results demonstrate that the arithmetic improvements are able to translate to application-level gains.

We will open source the code in the future.

**Structure of the Paper** The paper is structured as follows: Section 2 reviews the mathematical and verification background. Section 3 presents the First- and Second-generation implementations and their integration into OpenSSL. Section 4 describes the CryptoLine and HOL Light verification. Section 5 evaluates standalone multiplication, Montgomery multiplication, and RSA integration. Section 6 gives a short conclusion.

## 2 Preliminaries

### 2.1 Integer Multiplication via Polynomial Multiplication

The standard approach for NTT-based large-integer multiplication [BHK<sup>+</sup>22a] represents integers as polynomials and reduces integer multiplication to polynomial arithmetic. Let  $\beta = 2^\ell$  denote the radix base. Given an integer  $a$ , its radix- $\beta$  representation is written as

$$a = \sum_{i=0}^{m-1} a_i \beta^i.$$

This representation is associated with the polynomial

$$A(x) = \sum_{i=0}^{m-1} a_i x^i \in \mathbb{Z}[x],$$

so that evaluating at  $x = \beta$  recovers the original integer, namely  $A(\beta) = a$ . Similarly, an integer  $b$  is represented by a polynomial  $B(x)$  satisfying  $B(\beta) = b$ . We refer to the radix decomposition as chunking. Hence,  $C(x) = A(x)B(x) \Rightarrow C(\beta) = A(\beta)B(\beta) = ab$ . To compute  $C(x)$  efficiently, the polynomial multiplication is performed in NTT-friendly quotient rings. For each NTT prime  $q_j$ , the input polynomials are reduced modulo  $q_j$  and multiplied in  $\mathbb{Z}_{q_j}[x]/\langle x^n - 1 \rangle$ . The NTT machinery used to perform these polynomial products is reviewed next in Section 2.2.

The transform length  $n$  is chosen large enough to avoid degree wrap-around modulo  $x^n - 1$ . Since both input polynomials have degree at most  $m - 1$ , their product  $C(x) = A(x)B(x) = \sum_{t=0}^{2m-2} c_t x^t$  has degree at most  $2m - 2$ . Therefore, the transform length  $n \geq 2m - 1$  used in this work is sufficient to ensure that reduction modulo  $x^n - 1$  introduces no wrap-around.

The coefficients of  $C(x)$  may exceed a single NTT prime, so the computation is performed independently modulo multiple primes and then eventually combined with Chinese Remainder Theorem (CRT). More formally, because  $0 \leq a_i, b_i < \beta$ , each coefficient of  $C(x)$  satisfies

$$0 \leq c_t = \sum_{i+j=t} a_i b_j \leq m(\beta - 1)^2.$$

Let  $q_1, \dots, q_s$  be pairwise coprime NTT moduli and define  $Q = \prod_{j=1}^s q_j$ . Accordingly, the modulus-product requirement stated above is satisfied whenever  $Q > m(\beta - 1)^2$ .

Finally, dechunking evaluates the reconstructed polynomial at  $\beta = 2^\ell$  and propagates carries to obtain the final integer product  $ab$  [BHK<sup>+</sup>22a]. From here, `chk` denotes chunking and `dechk` denotes dechunking in the algorithms below.

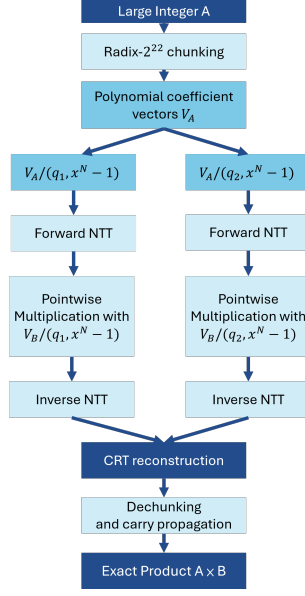


Figure 1: Overview of the multiplication pipeline

## 2.2 Number Theoretic Transform

Number-Theoretic Transform (NTT) is a finite-field analogue of the Discrete Fourier Transform [Pol71, Hwa24, LZ22]. By transforming polynomial multiplication into pointwise products, it reduces computational complexity from quadratic to  $O(n \log n)$  time.

**Definition** We work over the ring  $\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}$  for a prime modulus  $q$ . Let  $\omega \in \mathbb{Z}_q$  be a principal  $n$ -th root of unity with  $n \mid q - 1$ , and we denote  $[n] := \{0, 1, \dots, n - 1\}$ . The NTT is defined as the evaluation map over the quotient ring:

$$\text{NTT} : \mathbb{Z}_q[x] / \langle x^n - 1 \rangle \longrightarrow \mathbb{Z}_q^n, \quad \text{NTT}(a) := (a(1), a(\omega), \dots, a(\omega^{n-1})).$$

This map is a ring isomorphism, where multiplication on  $\mathbb{Z}_q^n$  can be performed component-wise.

$$a \cdot b = \text{NTT}^{-1}(\text{NTT}(a) \cdot_{\text{pt}} \text{NTT}(b)),$$

where  $\cdot_{\text{pt}}$  denotes pointwise multiplication.

From the divide-and-conquer perspective, the NTT can be viewed as the recursive factorization of  $x^n - 1 = (x^{n/2} - 1)(x^{n/2} + 1)$  for power-of-two  $n$ :

$$\mathbb{Z}_q[x] / \langle x^n - 1 \rangle \longrightarrow \prod_{i \in [n]} \mathbb{Z}_q[x] / \langle x - \omega^i \rangle,$$

The inverse transform  $\text{NTT}^{-1}$  is then obtained by evaluating at the inverse powers of  $\omega$  and scaling by  $n^{-1} \pmod{q}$ .

**FFT Butterflies and Lazy Reduction** NTT relies on two standard butterfly patterns: Cooley-Tukey (CT) [CT65] and Gentleman-Sande (GS) [GS66]. A common approach uses CT butterflies for the forward NTT and GS butterflies for the inverse NTT. Given an input pair  $(a, b)$  and a twiddle factor  $\omega$ , a CT butterfly computes:

$$t \leftarrow \omega \cdot b \bmod q, \quad a' \leftarrow a + t \bmod q, \quad b' \leftarrow a - t \bmod q.$$

A GS butterfly computes:

$$t \leftarrow a + b \bmod q, \quad u \leftarrow a - b \bmod q, \quad a' \leftarrow t, \quad b' \leftarrow \omega^{-1}u \bmod q.$$

Reductions after every butterfly are expensive, so implementers tend to delay them and allow coefficients to grow beyond the canonical range, namely lazy reduction. This saves instructions but requires proving that subsequent operations remain within safe ranges. Although GS butterflies are the conventional choice for inverse NTT, it can be observed that the range of the  $a + b$  term is doubled every butterfly, growing exponentially. In contrast, CT butterflies effectively control the range by posing a modular multiplication for every  $t \leftarrow \omega b$ , resulting in only linear growth with respect to the output bound of corresponding modular multiplication. Following the explanation in [Yan26], performing NTT from either side using CT and GS butterflies can lead to equivalent results if the twiddle factors are correctly managed. Hence, CT butterflies for both sides of NTT are more advantageous for range management.

**Good’s Trick and Zero-Skipping** When the NTT length is  $n = rs$  with  $\gcd(r, s) = 1$ , Good’s trick maps the transform into an  $r \times s$  two-dimensional array [Goo58, Hwa24].

$$\mathbb{Z}_q[x]/\langle x^n - 1 \rangle \cong \mathbb{Z}_q[x]/\langle x^r - 1 \rangle \otimes_{\mathbb{Z}_q} \mathbb{Z}_q[x]/\langle x^s - 1 \rangle,$$

This reindexing eliminates inter-stage twiddle factors, enabling independent  $r$ -point and  $s$ -point NTTs along each dimension to reduce twiddle loading overhead.

The zero padding required for linear polynomial multiplication leaves roughly half of the transform input zero, and a straightforward Good-Thomas implementation may perform redundant operations on zero entries. To address this, the zero-skipping technique proposed in [CYW25] optimizes the computation by cyclically shifting coefficients within each column to align the zero patterns with the first transform layer. By the time-shifting property of the DFT, this cyclic shift only introduces known twisting factors in the transform domain, which can be absorbed into subsequent computations.

## 2.3 Modular Multiplication

**Overview** Following the notation of [BHK<sup>+</sup>22b], given two integers  $a, b$  and an odd modulus  $q$ , modular multiplication computes a signed representative of  $ab \bmod^\pm q$ . We denote rounding to the nearest integer by  $\lfloor \cdot \rfloor$  and rounding to the nearest even integer by  $\lfloor \cdot \rfloor_2$ . A naive implementation computes the full product and then performs a generic modular reduction, but division by  $q$  is expensive on common processors and is also usually avoided in constant-time cryptographic code. Barrett and Montgomery multiplication replace this division by multiplication with precomputed constants and division by a word-size radix  $R = 2^w$  [Bar87, Mon85].

### 2.3.1 Barrett Multiplication

**Barrett Multiplication and Three-instruction Barrett** Barrett multiplication is convenient when one operand is fixed or known in advance. First, note that  $ab \bmod^\pm q = ab - q \lfloor \frac{ab}{q} \rfloor$ , but the division is still expensive. In the NTT setting, the known factor  $b$  is a twiddle factor  $\omega_n^i$ , where  $i = 0, \dots, n-1$  and  $n$  is the NTT length, and we rewrite  $\frac{ab}{q} = \frac{a}{R} \cdot \frac{bR}{q}$ , and precompute  $v = \lfloor \frac{bR}{q} \rfloor$ . Then we approximate  $ab \bmod^\pm q \approx ab - q \lfloor \frac{av}{R} \rfloor$ . The resulting computation is summarized in Algorithm 1, as in [BHK<sup>+</sup>22b]. For the actual implementation in ARM assembly, the SIMD instruction `sqrddmulh` performs a saturating,

rounding, doubling multiply-high operation [Arm]. This realizes the approximation above and reduces the entire computation to only three instructions.

The output range of Barrett multiplication determines how aggressively NTT computation can use lazy reduction. For the three-instruction Barrett multiplication of Becker et al. [BHK<sup>+</sup>22b], the general output bound is  $(-1.5q, 1.5q)$ . Under the stronger input condition  $|a| < q$ , Corollary 2 of [BHK<sup>+</sup>22b] gives the tighter output bound  $(-q, q)$ . [Bec26] gives a precise pointwise output bound for Barrett multiplication

$$|\text{BarMul}(a, b)| \leq \frac{|a|q\epsilon_b}{R} + \frac{q}{2}.$$

by formal verification in Isabelle/HOL.  $\epsilon_b$  is the pointwise approximation error of the precomputed even-integer approximation for  $bR/q$ , and it is equal or smaller than 1 for round-to-nearest-even approximation. Therefore, in the worst case, when input  $|a| = R/2$  in signed arithmetic, the output bound is at most  $q/2 + q/2 = q$ . Compared to the old result, the output bound improves by half a modulus, and for smaller inputs, it is tighter than the previous coarse bound  $q$ .

---

**Algorithm 1** One Known-Factor Barrett Multiplication

---

**Require:**  $|a| < R/2$ ,  $q < R/3$  and is odd,  $R = 2^{32}$ , and  $\omega$  is a known factor

**Require:**  $v = \lfloor \omega R/q \rfloor$  for the abstract form

**Require:**  $h = \frac{1}{2} \lfloor \omega R/q \rfloor_2$  for the Neon realization

**Ensure:**  $z \equiv a\omega \bmod^\pm q$ ,  $|z| \leq \frac{|a|q\epsilon_b}{R} + \frac{q}{2}$ , where  $\epsilon_b = |\omega R/q - \lfloor \omega R/q \rfloor_2|$

**Abstract arithmetic form**

```

1:  $z \leftarrow a\omega$ 
2:  $t \leftarrow \lfloor \frac{av}{R} \rfloor$ 
3:  $z \leftarrow z - qt$ 
4: return  $z$ 

```

**Neon realization**

```

1: mul  $z, a, \omega$ 
2: sqrdmulh  $t, a, h$ 
3: mls  $z, t, q$ 
4: return  $z$ 

```

---

### 2.3.2 Montgomery Multiplication

**Single-Precision Montgomery Multiplication and Its Doubling Form.** Montgomery multiplication [Mon85] computes a representative of  $abR^{-1}$  modulo  $q$  without division by  $q$ . Following [BHK<sup>+</sup>22b], let  $R = 2^w > q$  be the word-size radix and let  $q' = -q^{-1} \bmod R$ . For word-size operands  $a, b$ , the Montgomery product is defined as

$$\begin{aligned} \text{Mont}_q(a, b) &= \left\lfloor \frac{ab + q \cdot ((q' \cdot (ab \bmod R)) \bmod R)}{R} \right\rfloor \\ &= \text{hi}(ab + q \cdot \text{lo}(q' \cdot \text{lo}(ab))), \end{aligned}$$

where  $\text{lo}$  and  $\text{hi}$  extract the lower and upper  $w$  bits, respectively. The output satisfies  $\text{Mont}_q(a, b) \equiv abR^{-1} \pmod{q}$ . We first consider the single-precision setting used over the NTT moduli; the RSA-scale multi-precision setting follows below.

For the doubling form, we use  $T = q^{-1} \bmod R$ , corresponding to the opposite sign convention from  $q' = -q^{-1} \bmod R$  [BHK<sup>+</sup>22b] used in the definition above. Algorithm 2 shows the abstract doubling form and the Neon realization used in our implementation.

**Algorithm 2** Single-Precision Montgomery Multiplication with Doubling**Require:**  $q$  is odd,  $R = 2^w > q$ , and  $T = q^{-1} \bmod R$ **Require:**  $|a|, |b| < R/2$ **Ensure:**  $z \equiv abR^{-1} \bmod^{\pm} q$ , with  $|z| \leq \frac{|a||b|}{2^w} + \frac{q}{2}$ **Abstract arithmetic form**

```

1:  $z \leftarrow \left\lfloor \frac{2ab}{R} \right\rfloor$ 
2:  $k \leftarrow abT \bmod R$ 
3:  $c \leftarrow \left\lfloor \frac{2kq}{R} \right\rfloor$ 
4:  $z \leftarrow (z - c)/2$ 
5: return  $z$ 

```

**Neon realization**

```

1:  $z \leftarrow \text{sqdmulh}(a, b)$ 
2:  $v \leftarrow \text{mul}(b, T)$ 
3:  $k \leftarrow \text{mul}(a, v)$ 
4:  $c \leftarrow \text{sqdmulh}(k, q)$ 
5:  $z \leftarrow \text{shsub}(z, c)$ 
6: return  $z$ 

```

The Neon realization follows Becker et al.’s doubling Montgomery structure, but computes  $bT \bmod R$  on the fly, resulting in a five-instruction sequence for two data-dependent operands.

Another variant is Algorithm 14 in [BHK<sup>+</sup>22b], its long arithmetic allows products to be accumulated before a single Montgomery reduction.

**Algorithm 3** Montgomery multiplication in long arithmetic**Require:**  $q$  is odd,  $R = 2^w > q$ , and  $q' = -q^{-1} \bmod R$ **Require:**  $|a|, |b| < R/2$ **Ensure:**  $c \equiv abR^{-1} \bmod^{\pm} q$ , with  $|c| \leq \frac{|a||b|}{2^w} + \frac{q}{2}$ **Neon realization**

```

1: smull  $l, a, b$ 
2: smull2  $h, a, b$ 
3: uzp1  $t, l, h$ 
4: mul  $t, t, q'$ 
5: smlal  $l, t, q$ 
6: smlal2  $h, t, q$ 
7: uzp2  $c, l, h$ 
8: return  $c$ 

```

**Multi-Precision Montgomery Multiplication** At the RSA-size arithmetic layer, conventional implementations perform the Montgomery reduction iteratively over limbs, leading to operand-scanning methods such as CIOS and FIOS [KAK96]. In contrast, the NTT-based strategy of [BHK<sup>+</sup>22a] computes the large products appearing in the Montgomery reduction using NTT-based integer multiplication. Let  $N$  be an odd modulus, let  $R_{\text{mp}} = 2^k$  be the multi-precision Montgomery radix, and let  $N^{-1}$  denote the inverse of  $N$  modulo  $R_{\text{mp}}$ . Given large operands  $A$  and  $B$ , the procedure first computes the full product  $t = AB$ . It then determines the Montgomery reduction coefficient  $u = (t \bmod R_{\text{mp}})N^{-1} \bmod R_{\text{mp}}$ , forms the correction term  $r = uN$ , and outputs  $C = \frac{t-r}{R_{\text{mp}}}$ . Since  $uN \equiv t \pmod{R_{\text{mp}}}$ , the numerator  $t - r$  is divisible by  $R_{\text{mp}}$ , and the output satisfies

$$C \equiv AB R_{\text{mp}}^{-1} \pmod{N}.$$

**NTT-based Realization of Multi-Precision Montgomery** In our implementation, the large products required by the Montgomery reduction are computed using the NTT-based large-integer multiplication engine, following [BHK<sup>+</sup>22a]. Because  $N$  and  $N^{-1} \bmod R_{\text{mp}}$  are fixed for a given Montgomery context, their chunked NTT representations can be precomputed. The transform of  $B$  can also be precomputed when one operand is fixed. Algorithm 4 summarizes the resulting procedure with CRT reconstruction being abstracted.

**Algorithm 4** NTT-Based Multi-Precision Montgomery Multiplication

---

**Input:**  $a, b \in [0, N)$ ,  $R_{\text{mp}} = 2^k > N$ ,  
 $A = aR_{\text{mp}} \bmod N$ ,  $B = bR_{\text{mp}} \bmod N$ ,  
 $\hat{N}^{-1} = \text{NTT}(\text{chk}(N^{-1} \bmod R_{\text{mp}}))$ ,  
and  $\hat{N} = \text{NTT}(\text{chk}(N))$ .  
**Output:**  $C \in [0, N)$  such that

$$C \equiv ABR_{\text{mp}}^{-1} \equiv abR_{\text{mp}} \pmod{N}.$$


---

```

1:  $\hat{A} \leftarrow \text{NTT}(\text{chk}(A))$ 
2:  $\hat{B} \leftarrow \text{NTT}(\text{chk}(B))$  ▷ Precomputable if  $B$  is fixed
3:  $t \leftarrow \text{dechck}(\text{NTT}^{-1}(\hat{A} \cdot_{pt} \hat{B}))$ 
4:  $\hat{t} \leftarrow \text{NTT}(\text{chk}(t \bmod R_{\text{mp}}))$ 
5:  $u \leftarrow \text{dechck}(\text{NTT}^{-1}(\hat{t} \cdot_{pt} \hat{N}^{-1})) \bmod R_{\text{mp}}$ 
6:  $\hat{u} \leftarrow \text{NTT}(\text{chk}(u))$ 
7:  $r \leftarrow \text{dechck}(\text{NTT}^{-1}(\hat{u} \cdot_{pt} \hat{N}))$ 
8:  $C \leftarrow (t - r)/R_{\text{mp}}$ 
9: if  $C < 0$  then
10:    $C \leftarrow C + N$ 
11: end if
12: return  $C$ 

```

---

## 2.4 Chinese Remainder Theorem

Following [CHK<sup>+</sup>21, BHK<sup>+</sup>22a], we use a divided-difference form of CRT that accepts signed, not necessarily canonical, representatives. Given  $u \equiv u_0 \pmod{q_1}$  and  $u \equiv u_1 \pmod{q_2}$ , let  $m_1 = q_1^{-1} \bmod q_2$  and compute the representative  $d \equiv (u_1 - u_0)m_1 \bmod q_2$ .

We first form  $y = u_0 + dq_1$ , which implies  $y \equiv u_0 \pmod{q_1}$  and  $y \equiv u_1 \pmod{q_2}$ . Let  $Q = q_1q_2$ . Under the signed input bounds used in our implementation, if the intermediate result satisfies  $-Q < y < Q$ , the canonical representative is obtained by

$$u = \begin{cases} y + Q, & y < 0, \\ y, & y \geq 0. \end{cases}$$

Consequently,  $u \equiv u_0 \pmod{q_1}$ ,  $u \equiv u_1 \pmod{q_2}$ , and  $0 \leq u < Q$ .

However, since  $d$  is not centered, it is not obvious to see  $|y| < Q = q_1q_2$ . Assume the input is  $|u_0| \leq cq_1$  and  $|u_1| \leq cq_2$  for the same constant  $c$ . Then,  $|s| = |u_1 - u_0| \leq c(q_1 + q_2) \leq R/2$ . By the new pointwise output bound for Barrett multiplication in [Bec26],  $|d| \leq (\frac{c(q_1+q_2)}{R} + \frac{1}{2})q_2$  with  $\epsilon_{m_1} = 1$  as the worst case. Hence,  $|y| \leq cq_1 + |d|q_1$ .

$$\frac{|y|}{Q} \leq \frac{c}{q_2} + \frac{1}{2} + \frac{c(q_1 + q_2)}{R}.$$

We require it smaller than 1. Therefore,

$$c < \frac{1/2}{(q_1 + q_2)/R + 1/q_2} \rightarrow |y| < Q$$

The older coarse bound derived in [BHK<sup>+</sup>22b] is insufficient to establish the goal for the uncentered representative.

Under the coefficient bound stated in Section 2.1, this canonical representative uniquely recovers the corresponding integer coefficient after multi-moduli NTT.

## 2.5 CryptoLine and HOL Light

Following the methodology in [HLS<sup>+</sup>22, CLT<sup>+</sup>25, FLS<sup>+</sup>19], the CryptoLine verification in this work consists of two primary obligations. First, *algebraic correctness* ensures

that the program output satisfies the intended mathematical specification. Second, *range safety* ensures that intermediate values remain within valid bounds, which is crucial especially for ruling out unintended overflows in implementations using lazy reduction. The verification additionally relies on *functional correspondence* between the assembly code and its CryptoLine model. This correspondence is established manually by translating the assembly according to CryptoLine’s AArch64 instruction semantics and is included in our trusted computing base. A CryptoLine specification is framed by *preconditions* for input assumptions and *postconditions* for required output properties. For long assembly kernels, intermediate proof points are introduced as *cuts*; these cuts record intermediate program states and decompose large verification tasks into smaller proof obligations. Within these specifications, *algebraic assertions* express arithmetic relations, while *range assertions* bound intermediate values and support overflow-safety reasoning.

HOL Light [Har09] is an interactive theorem prover based on higher-order logic. It follows the LCF discipline: all theorems are constructed through a small trusted kernel implementing the inference rules and axioms of the logic. In this work, CryptoLine is used for low-level kernel verification, while HOL Light is used for high-level composition of the multiplication pipeline. Section 4 explains the interface and trust boundary in detail.

## 3 Implementation

### 3.1 Parameters and Pipelines

We use radix  $\beta = 2^{22}$  for all four parameter sets. The supported operand sizes are 5632, 8448, 11264, and 14080 bits, and the corresponding transform lengths are 512, 768, 1024, and 1280 after zero-padding. We implement these transforms using a radix-2 512-point NTT,  $3 \times 256$  Good’s trick with  $i \rightarrow (i \bmod 3, i \bmod 256)$ , a radix-2 1024-point NTT, and a  $5 \times 256$  Good’s trick with  $i \rightarrow (i \bmod 5, i \bmod 256)$ , respectively.

As discussed in Section 2.1, the chunk width determines a trade-off between transform length and coefficient growth. A larger chunk width reduces the number of coefficients and hence the required transform length, but increases the coefficient bound and may require larger or additional NTT moduli. We choose  $\ell = 22$  to balance these factors while allowing efficient signed 32-bit SIMD arithmetic on AArch64 Neon. We use two primes near 27 bits, leaving several bits of headroom in signed 32-bit Neon lanes for lazy reduction. A single prime doesn’t satisfy the coefficient bound, whereas more moduli introduce more NTT / CRT work without shortening transform length.

Unless qualified as forward or inverse, an  $X$ -point NTT denotes the combined forward-transform, pointwise-multiplication, and inverse-transform kernel. Also, we denote the  $i$ -th forward and inverse layers by  $F_i$  and  $I_i$  respectively. Table 1 summarizes the resulting parameter sets and corresponding pipelines.

For each kernel explained below, we present only its best-performing version.

### 3.2 First Generation Implementation

**Winograd Style Inverse 3-point NTT** Used in the 8448-bit multiplier, the inverse 3-point NTT uses the Winograd-style form [Win78]

$$t = (b - c)\omega_3^2, \quad a' = a + b + c, \quad b' = a - c + t, \quad c' = a - b - t.$$

This reduces the kernel from four Barrett multiplications to one. The inverse scaling is folded into the preceding 256-point stage.

**Pair-sum Pair-difference 5-point NTT and Zero-Skipping** For the 5-point NTT in the 14080-bit multiplier, we start from a pair-sum and pair-difference formulation of the radix-5

**Table 1:** Parameter sets and high-level multiplication pipelines.

| <i>Bits</i> | <i>n</i> | <i>q</i> <sub>1</sub> | <i>q</i> <sub>2</sub> | <i>Q</i> = <i>q</i> <sub>1</sub> <i>q</i> <sub>2</sub> | Pipeline                                                                                              |
|-------------|----------|-----------------------|-----------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| 5632        | 512      | 125399041             | 134203393             | 16828976781146113                                      | Chunking → 512-point NTT → CRT with dechunking                                                        |
| 8448        | 768      | 133199617             | 134203393             | 17875840547700481                                      | Chunking → zero-skipping forward 3-point NTT → 256-point NTT → inverse 3-point NTT → CRT → dechunking |
| 11264       | 1024     | 125399041             | 125417473             | 15727230838843393                                      | Chunking → 1024-point NTT → CRT with dechunking                                                       |
| 14080       | 1280     | 126320641             | 126309121             | 15955449128866561                                      | Chunking → zero-skipping forward 5-point NTT → 256-point NTT → inverse 5-point NTT → CRT → dechunking |

transform. For a full five-input transform, this reduces the 16 constant multiplications to eight, which the inverse 5-point NTT implements directly.

The forward transform admits further specialization because only the first 640 coefficients are nonzero before the  $5 \times 256$  Good decomposition. Consequently, the first 128 columns contain three nonzero rows, while the remaining 128 columns contain two. We specialize the radix-5 formulas to these two cases, reducing the core to four and three constant multiplications, respectively.

For the last four outputs  $Y_1, \dots, Y_4$ , we retain a doubled intermediate representation and absorb the factor of two into precomputed half-twist constants,  $(2Y_j) \cdot \text{twist}_j / 2 = Y_j \text{twist}_j$ . The inverse normalization is folded into the last inverse layer of the 256-point NTT.

**1024-point NTT** A 1024-point NTT consists of ten forward layers, pointwise multiplication, and ten inverse layers. A straightforward implementation processes each layer separately by loading the coefficients from memory, applying the butterfly operations, and storing the results back. However, processing each layer separately introduces redundant memory access overhead and underutilizes the vector registers, thereby preventing efficient register-level data reuse and limiting overall performance. To reduce this overhead, we fuse consecutive layers whenever possible. The 10 layers are grouped as  $3 \rightarrow 2 \rightarrow 3 \rightarrow 2$  blocks, and the inverse transform is organized in the reverse order. Moreover, we fuse the last block of forward NTT, pointwise multiplication, and first block of inverse NTT together. The final scaling factor of the inverse transform is folded into the last layer of inverse 1024 NTT.

After dealing with memory traffic by layer fusion, modular multiplication is another major cost. To save multiplications, we take advantage of the padded zeros in the first layer so that the butterflies can be replaced with `mov` instructions. When the twiddle factor is 1 in the first three layers in the forward NTT and first two layers in the inverse NTT, we can save modular multiplications and use additions and subtractions instead at the cost of range being doubled.

As mentioned in Section 2.2, we use CT butterflies for both forward and inverse NTT to strike a balance between aggressive lazy reduction and range control. Also, twiddle-factor multiplications inside butterfly operations use Barrett multiplication, whereas pointwise multiplications use Montgomery multiplication.

**Twiddle factor layout** For forward NTT, each forward layer’s twiddle sequence is a prefix of the next layer’s sequence. Therefore, we store the forward twiddle factors in the layout,

where  $\text{br}_k(i)$  denotes the  $k$ -bit reversal of  $i$ .

$$\text{tw}_{\text{fwd}}[i] = \omega^{\text{br}_{k-1}(i)}, \quad 0 \leq i < 2^{k-1}.$$

For the inverse NTT, however, we store the twiddle factors in natural order per layer for layer-specific inverse twists designed for CT-style inverse NTT, which requires a dedicated, separate twiddle table due to this distinct ordering.

$$\text{tw}_{\text{inv}}[i][j] = \omega^{-j2^{k-1-i}}, \quad 0 \leq i < k, \quad 0 \leq j < 2^i.$$

### 3.3 Second Generation Implementation

Motivated by [Bec26], **Second** applies the tighter Barrett bound to re-optimize the following kernels.

#### 3.3.1 256- and 512-point NTT

Similar to the 1024-point NTT, 256-point NTT consists of 8 layers, which is grouped as  $4 \rightarrow 4$ . The aggressive fusion strategy follows [BHK<sup>+</sup>22b]’s negacyclic 256-point implementation for Dilithium, resulting in high register pressure, but it brings only three load / store phases ( $F_1 \rightarrow F_4, F_5 \rightarrow I_4, I_5 \rightarrow I_8$ ).

Since 256-point NTT is after zero-skipping 3-point NTT, there is no zero-padding structure anymore. However, due to the new bound provided in [Bec26], **Second** removes the reduction between  $I_1$  and  $I_2$  present in **First** generation while retaining the omission of Barrett multiplication by 1 in  $F_1-F_3$  and  $I_1-I_2$ . For the pointwise multiplication, we find that Montgomery with long arithmetic is faster by 17% on our Cortex-A76 than the doubling variant.

The implementation of 512-point NTT is trickier because the 9 layers aren’t separated as 3 blocks. Since the first and last layer can be absorbed across kernel boundaries, we decide to move the first zero-padding layer to the former data transfer between chunking and NTT, and the last inverse layer to the later one, CRT with dechunking. Thus, the remaining NTT is  $F_2-F_9$ , pointwise multiplication, and  $I_1-I_8$ , which is equivalent to a cyclic 256-point NTT for the first half of the elements, and a negacyclic 256-point NTT for the second half of the elements. This decomposition allows the 512-point implementation to reuse most of the optimized 256-point kernel while preserving the same load/store structure.

**Twiddle factor layout** Unlike the 1024-point implementation, the fused 256-point kernel physically repacks the twiddles according to the access pattern of each fused block. Early layers interleave each twiddle with its precomputed Barrett constants, whereas the last layers use layouts matching their vector load order. The cyclic half of the 512-point kernel reuses this layout; the negacyclic half follows the same packing strategy with its  $\psi$ -based twiddles, where  $\psi$  is a primitive 512-th root with  $\omega = \psi^2$ .

#### 3.3.2 Chunking and Dechunking

Section 2.1 defines chunking and dechunking abstractly. Our implementation exploits the identity  $704 = 11 \cdot 64 = 32 \cdot 22$ . Thus, every 11 radix-2<sup>64</sup> limbs correspond exactly to 32 radix-2<sup>22</sup> coefficients. The chunking routine uses scalar instructions such as `ubfx` and `extr`, which are convenient for bit extraction. Because  $64 \equiv -2 \pmod{22}$ , the insertion offsets follow the fixed sequence  $0, 20, 18, \dots, 2$ . The extraction schedule within each iteration is fully unrolled, with all shift constants hard-coded.

Dechunking contains a serial dependency through carry propagation, but its memory-access and bit-packing patterns remain regular. Conceptually, it streams the radix-2<sup>22</sup>

coefficients, normalizes each coefficient together with the incoming carry, and packs every 32 normalized digits into 11 radix-2<sup>64</sup> limbs. To reduce the dependency, we separate the chain to two subchains for 5632-bit and three subchains for 8448-bit multiplier, and propagate the carry from the end of the first chain.

For the  $3 \times 256$  Good's trick layout, the dechunking routine maintains three row pointers and accesses the coefficients in their logical one-dimensional order. This traversal performs the inverse Good permutation implicitly. For the 512-point multiplier, no such permutation is required; we instead fuse CRT reconstruction with dechunking and use SLOTHY [ABKK24, slo] to interleave the vectorized CRT instructions with scalar carry propagation.

### 3.3.3 Zero-Skipping 3-point Forward NTT

The  $3 \times 256$  Good decomposition exposes two zero patterns in the forward radix-3 transform. For the first 128 columns, the inputs have the form  $(a, b, 0)$ , with base transform

$$y_0 = a + b, \quad y_1 = a + b\omega_3, \quad y_2 = a + b\omega_3^2.$$

After the Good permutation, the two outputs are multiplied by  $\tau$  and  $\tau^2$ , respectively, where  $\tau \in \{1, \omega_3, \omega_3^2\}$ .

Using  $1 + \omega_3 + \omega_3^2 = 0$ , we specialize the three twist cases:

$$\begin{aligned} \tau = 1 : \quad & t = b\omega_3, \quad y_1 = a + t, \quad y_2 = a - b - t, \\ \tau = \omega_3 : \quad & t = (a - b)\omega_3, \quad \tau y_1 = t - b, \quad \tau^2 y_2 = -a - t, \\ \tau = \omega_3^2 : \quad & t = a\omega_3, \quad \tau y_1 = b - a - t, \quad \tau^2 y_2 = b + t. \end{aligned}$$

Thus, each twist class requires only one constant multiplication.

For the remaining 128 columns, the inputs have the form  $(a, 0, 0)$ . The same identity allows the two powers of  $\omega_3$  to be obtained from one multiplication and additions/subtractions. Among the three twist classes, the class with twist one requires no multiplication, leaving only two vector Barrett multiplications.

The Good-twist phases repeat every three columns. We use `ld3` to deinterleave consecutive columns by phase, so that all four SIMD lanes share the same twist. This allows a vector whose twist is one to bypass the twist multiplication entirely.

### 3.3.4 Range Analysis for 8448-bit Multiplication

We derive the range bound for **Second** 8448-bit pipeline due to the omission of multiplication by 1 and the aggressive use of lazy reduction. The 5632-bit one can be checked analogously. Our primes are approximately 27 bits, putting the parameters into the formula in Section 2.4, the bound is approximately  $8q_i$ ,  $i \in \{1, 2\}$ . Also,

$$2^{31}/q_i \approx 16,$$

so lazy-reduction intermediate values have roughly  $16q_i$  of headroom in signed 32-bit arithmetic before overflow.

Table 2 shows that the bound is insufficient to omit multiplication by 1 in  $F_4$  for 256-point NTT. However, we can shorten 3-instruction Barrett multiplication to 2 without `mul`. In contrast,  $F_4$ 's multiplication by 1 is saved for 512-point NTT.

## 3.4 RSA-size Montgomery Multiplication and OpenSSL Integration

**Truncated Low-Product Optimization** For RSA-8192, since the 8448-bit multiplication routine integrates the inverse Good permutation into dechunking, our current implementation still performs CRT reconstruction for the complete coefficient array due to the

**Table 2:** Each NTT entry  $c$  denotes the maximum absolute coefficient bound  $|x| \leq cq_i$ .  $D(c) = 2c$  for addition/subtraction-only layers,  $A_i(c) = c + \frac{1}{2} + \frac{cq_i}{R}$  for ordinary CT layers, and  $M_i(c) = \frac{1}{2} + \frac{c^2 q_i}{R}$  for pointwise Montgomery multiplication.

| Stage                        | $q_1 = 133199617$ | $q_2 = 134203393$ | Note                                      |
|------------------------------|-------------------|-------------------|-------------------------------------------|
| Input to forward 3-point NTT | $0.03149q_1$      | $0.03126q_2$      | radix-2 <sup>22</sup> coefficients        |
| Forward 3-point NTT          | $0.56396q_1$      | $0.56349q_2$      | zero-skipping radix-3                     |
| $F_1$                        | $1.128q_1$        | $1.127q_2$        | $D(c)$                                    |
| $F_2$                        | $2.256q_1$        | $2.254q_2$        | $D(c)$                                    |
| $F_3$                        | $4.512q_1$        | $4.508q_2$        | $D(c)$ ; multiplication by 1 omitted      |
| $F_4$                        | $5.152q_1$        | $5.149q_2$        | $A_i(c)$ ; reduction retained             |
| $F_5$                        | $5.811q_1$        | $5.810q_2$        | $A_i(c)$                                  |
| $F_6$                        | $6.492q_1$        | $6.491q_2$        | $A_i(c)$                                  |
| $F_7$                        | $7.193q_1$        | $7.194q_2$        | $A_i(c)$                                  |
| $F_8$                        | $7.916q_1$        | $7.919q_2$        | $A_i(c)$                                  |
| Pointwise multiplication     | $2.443q_1$        | $2.459q_2$        | $M_i(c)$                                  |
| $I_1$                        | $4.887q_1$        | $4.919q_2$        | $D(c)$                                    |
| $I_2$                        | $9.773q_1$        | $9.838q_2$        | $D(c)$                                    |
| $I_3$                        | $10.577q_1$       | $10.645q_2$       | $A_i(c)$                                  |
| $I_4$                        | $11.405q_1$       | $11.478q_2$       | $A_i(c)$                                  |
| $I_5$                        | $12.258q_1$       | $12.336q_2$       | $A_i(c)$                                  |
| $I_6$                        | $13.138q_1$       | $13.222q_2$       | $A_i(c)$                                  |
| $I_7$                        | $14.046q_1$       | $14.135q_2$       | $A_i(c)$                                  |
| $I_8$                        | $14.981q_1$       | $15.077q_2$       |                                           |
| Inverse scaling              | $0.96462q_1$      | $0.97109q_2$      |                                           |
| Inverse 3-point NTT          | $2.89385q_1$      | $2.91326q_2$      | Both below the CRT admissible input bound |

memory layout, while dechunking terminates after producing the required lower output limbs.

For the RSA-10240 parameter set, the computation of the Montgomery reduction coefficient requires only the lower  $k$  bits of the product used to derive  $u = (t \bmod R_{\text{mp}})N^{-1} \bmod R_{\text{mp}}$ . Consequently, the corresponding inverse NTT does not need to materialize the complete integer product. We omit 512 final scaling operations and the lower half of the butterflies in the last inverse-NTT layer. The subsequent CRT reconstruction and dechunking stages are truncated accordingly, reducing the materialized output to 468 CRT coefficients and 160 radix-2<sup>64</sup> limbs.

**Integration into OpenSSL** The modulus-dependent NTT representations are precomputed during RSA key generation, following the fixed-operand strategy of [BHK<sup>+</sup>22a]. OpenSSL’s RSA implementation contains two relevant one-known-input Montgomery operations. The conversion into the Montgomery domain multiplies by  $R_{\text{mp}}^2 \bmod N$ , whereas the conversion back to the standard representation multiplies by  $1 \bmod N$  [Ope26]. Since these operands are fixed, their NTT representations can be precomputed and reused.

The private signing and decryption paths in OpenSSL use CRT [Ope26, MKJR16] over two approximately 5120-bit factors for RSA-10240. Because our 11264-bit kernel is poorly matched to these half-size products, we apply the NTT-based Montgomery path only to the full-modulus encryption and verification operations.

## 4 Verification

### 4.1 Overview

All the kernels verified are in our **First** generation. We provide the example of range analysis for **Second** in Section 3.3.4. CryptoLine focuses on the algebraic and range contracts of low-level assembly kernels. After verifying the original routine, we use CryptoLine’s `cv_cec` [FLS<sup>+</sup>19, fml] to check the equivalence between the original implementation and the SLOTHY-optimized version [ABKK24, slo], validating that the scheduling preserves the program semantics. For end-to-end correctness, HOL Light composes the corresponding high-level contracts, manually translated from the CryptoLine specifications, into the integer-multiplication theorem.

### 4.2 Low-level Kernels with CryptoLine

At the low level, each CryptoLine procedure is specified by algebraic and range pre/post-conditions. CryptoLine discharges these obligations over the assembly-derived model using a Satisfiability Modulo Theories (SMT) solver for range constraints, a Computer Algebra System (CAS) for nonlinear algebraic conditions, and the Integer Set Library (ISL) for linear arithmetic. To make verified kernels reusable, we use `call` to encapsulate each proved kernel behind its pre/postcondition, following the pattern

$$\text{proc } f(\text{inputs}; \text{outputs}) = \{A_{\text{pre}} \&\& R_{\text{pre}}\} \cdots \{A_{\text{post}} \&\& R_{\text{post}}\},$$

where  $A$  and  $R$  denote the algebraic and range assertions, respectively.

#### 4.2.1 Modular Multiplication

Our assembly implementation utilizes Barrett and Montgomery multiplication. We verify these routines for algebraic correctness and range safety. For Barrett multiplication, we prove that  $z \equiv aw \pmod{q_i}$ , and  $|z| < q_i$  when  $|a| < q_i$ . Otherwise  $|z| < 1.5q_i$ . For Montgomery Multiplication, the postconditions are  $z \equiv abR^{-1} \pmod{q_i}$  and  $|z| < \frac{|a||b|}{R} + \frac{q_i}{2}$ .

Certain Neon instructions hide information needed by the algebraic proof. In the doubling Montgomery multiplication, `sqdmulh` exposes only the signed high part of each doubled product. We therefore introduce ghost variables satisfying

$$ab = 2^{31}h_{ab} + \ell_{ab}, \quad kq = 2^{31}h_{kq} + \ell_{kq}.$$

The final `shsub` computes

$$r = \left\lfloor \frac{h_{ab} - h_{kq}}{2} \right\rfloor.$$

Writing  $h_{ab} - h_{kq} = 2r + b$  for the discarded bit  $b \in \{0, 1\}$  gives

$$ab - kq = 2^{32}r + (\ell_{ab} + b2^{31} - \ell_{kq}).$$

CryptoLine proves that the residual term satisfies

$$\ell_{ab} + b2^{31} - \ell_{kq} \equiv 0 \pmod{2^{32}}$$

and

$$-(2^{31} - 1) \leq \ell_{ab} + b2^{31} - \ell_{kq} \leq 2^{32} - 1.$$

Hence the residual is zero, establishing the cancellation required for the Montgomery congruence.

Montgomery multiplication involves products of two variable operands, introducing non-linear polynomial constraints with intermediate terms. We use Singular [DGPS24]

to compute Gröbner bases, which eliminate these intermediate terms and establish the final modular equivalence. Following the approach in [CLT<sup>+</sup>25], Barrett multiplication in our implementation multiplies symbolic values by precomputed constants, so the resulting obligations remain linear and can be efficiently proved by the ISL solver [Ver10].

#### 4.2.2 256 NTT

**Overview** The 8448-bit multiplier applies the same fused 256-point row kernel to the three rows of the  $3 \times 256$  Good decomposition. Rather than verify the fully unrolled three-row implementation as one procedure, we verify one row kernel and reuse its contract three times. Equivalence checking then establishes that these three calls match the concrete unrolled implementation. The row kernel itself is divided into the five stages shown in Table 3.

**Table 3:** Stage decomposition used in the CryptoLine verification of the fused 256-point NTT multiplication kernel

| Stage   | CryptoLine routine      | Fwd.    | Inv.    | State                  |
|---------|-------------------------|---------|---------|------------------------|
| Stage 1 | <code>stageOne</code>   | (1)–(3) | –       | (A, B)                 |
| Stage 2 | <code>stageTwo</code>   | (4)–(6) | –       | (A, B)                 |
| Stage 3 | <code>stageThree</code> | (7)–(8) | (1)–(2) | (A, B $\rightarrow$ A) |
| Stage 4 | <code>stageFour</code>  | –       | (3)–(5) | (A)                    |
| Stage 5 | <code>stageFive</code>  | –       | (6)–(8) | (A)                    |
| Main    | <code>main</code>       | (1)–(8) | (1)–(8) | (A, B $\rightarrow$ A) |

Each stage is verified independently, where the range postconditions serve as the preconditions for the subsequent stage. For instance, Stage 3 begins from the postcondition  $\Phi_2$  established by Stage 2, executes forward layers  $F_7, F_8$ , the pointwise multiplication, and inverse layers  $I_1, I_2$ , and establishes the postcondition  $\Phi_3$  consumed by Stage 4. The main procedure composes these verified stage specifications to establish the final polynomial congruence of the kernel. We use similar methodology for the verification of 1024-point NTT.

**Algebraic conditions** Let  $a(x)$  and  $b(x)$  be the two input polynomials, and let  $r(x)$  be the output polynomial produced by the kernel. The final postcondition is expressed as the row-wise convolution

$$r(x) \equiv \kappa \cdot a(x)b(x) \pmod{q, x^{256} - 1},$$

where  $\kappa$  denotes the remaining normalization factor after the inverse transform. In the 8448-bit multiplication, this factor is  $\kappa = 3^{-1} \bmod q$ , coming from the inverse 3-point NTT.

Each stage-level postcondition records the state produced by the corresponding fused code block. Since such a block is only a partial transform rather than a standalone mathematical primitive, its contract describes an intermediate composition. The composed `main` procedure can instead state the final correctness theorem in convolution form.

**Range Conditions** The range assertions propagate coefficient bounds through the same stage boundaries used by the algebraic proof. For butterfly additions and subtractions, the bounds are propagated by interval addition. If  $a$  is bounded by  $|\alpha|q$  and  $b$  is bounded by  $|\beta|q$ , then both  $a + b$  and  $a - b$  are bounded by  $(|\alpha| + |\beta|)q$ . This mechanism accounts for range growth across butterfly layers when reductions are delayed. The range postcondition for each stage is derived from the final bounds of its live output vectors. The purpose

of the range proof is not only to prevent signed integer overflow, but also to justify the omitted reductions in the fused implementation. Since  $q < 2^{27}$ , values bounded by  $16q$  still fit within the signed 32-bit range. The CryptoLine range assertions show that the full fused kernel remains within the required bounds while using lazy reduction.

#### 4.2.3 Chunking and Dechunking with Implicit Inverse Permutation

We verify chunking and dechunking as changes of radix. For chunking, let the 132 unsigned 64-bit input limbs  $a_0, \dots, a_{131}$  represent the 8448-bit integer

$$A = \sum_{i=0}^{131} a_i 2^{64i}.$$

Because the input limbs are unsigned 64-bit variables, their bounds are already enforced by their types and no additional range precondition is required.

The chunking routine produces 384 radix- $2^{22}$  coefficients together with the zero-padding condition.

$$A = \sum_{j=0}^{383} c_j 2^{22j}, \quad 0 \leq c_j < 2^{22}$$

Thus, the proof establishes both the exact radix conversion and the input layout required by the subsequent 768-point NTT.

Conversely, dechunking verifies the conversion from the 768 reconstructed radix- $2^{22}$  coefficients back to 264 radix- $2^{64}$  output limbs. Let  $\pi$  denote the map from a logical coefficient index to its stored position in the Good-Thomas layout. Given reconstructed coefficients  $d_0, \dots, d_{767}$ , the postcondition proves that the output limbs  $r_0, \dots, r_{263}$  satisfy

$$\sum_{i=0}^{263} r_i 2^{64i} = \sum_{j=0}^{767} d_{\pi(j)} 2^{22j}, \quad 0 \leq r_i < 2^{64}.$$

#### 4.2.4 Remaining Kernels

We verify the zero-skipping forward and Winograd-style inverse radix-3 kernels, the specialized radix-5 kernels described in Section 3.2, and the CRT reconstruction using the same algebraic/range contract structure as above. Table 4 summarizes the verified contracts.

**Table 4:** Verified algebraic and range contracts for the remaining kernels.

| Kernel     | Verified algebraic contract                 | Range / special point |
|------------|---------------------------------------------|-----------------------|
| Fwd. 3-NTT | $y = \text{NTT}_3(x)$ with Good Permutation | $ y  < 1.5q_i$        |
| Inv. 3-NTT | $x' = \text{iNTT}_3(y)$                     | $ x'  < 4.5q_i$       |
| Fwd. 5-NTT | $y = \text{NTT}_5(x)$ with Good Permutation | $ y  < 1.5q_i$        |
| Inv. 5-NTT | $x' = \text{iNTT}_5(y)$                     | $ x'  < 7.5q_i$       |
| CRT        | $u \equiv u_i \pmod{q_i}, i = 1, 2$         | $0 \leq u < Q$        |

### 4.3 High-level Composition and HOL Light

CryptoLine isn't able to prove the end-to-end equivalence due to Gröbner elimination over radix-expanded integers being intractable. Based on the proved conditions in CryptoLine, we manually translate the specifications into HOL Light semantics as assumptions. Define

$$L_b(x_0, \dots, x_{n-1}) = \sum_{i=0}^{n-1} x_i b^i.$$

. We have four assumptions,

$$\begin{aligned} L_{2^{22}}(a_{22}) &= L_{2^{64}}(a_{64}), \\ L_{2^{22}}(b_{22}) &= L_{2^{64}}(b_{64}), \\ L_{2^{22}}(\text{GoodDiag}(c_{\text{CRT}})) &= L_{2^{22}}(\text{Conv}(a_{22}, b_{22})), \\ L_{2^{64}}(r_{64}) &= L_{2^{22}}(\text{GoodDiag}(c_{\text{CRT}})). \end{aligned}$$

. Here,  $\text{Conv}(a_{22}, b_{22})$  denotes ordinary coefficient convolution, while  $\text{GoodDiag}$  maps the Good-layout CRT output back to logical coefficient order. The first two assumptions instantiate the chunking contracts, the third captures the high-level NTT/CRT pipeline contract corresponding to the low-level proofs, and the fourth instantiates dechunking. HOL Light then composes these relations to recover integer multiplication. We use the assumptions to prove the theorem  $L_{2^{64}}(r_{64}) = L_{2^{64}}(a_{64})L_{2^{64}}(b_{64})$  for 8448-bit pipeline, and the 11264- and 14080-bit high-level pipeline are proved by the similar method.

## 4.4 Discussion

All verification were on an ARM Neoverse-N1 server with 80 cores.

**Table 5:** CryptoLine verification time for the 256-point NTT pipeline.

| Component                                    | Time          |
|----------------------------------------------|---------------|
| Doubling Montgomery pointwise multiplication | 2 m 26 s      |
| StageOne                                     | 44 m          |
| StageTwo                                     | 1 h 3 m       |
| StageThree                                   | 1 h 25 m      |
| StageFour                                    | 22 m          |
| StageFive                                    | 24 m          |
| Main procedure                               | 59 m          |
| Total                                        | $\approx 5$ h |

The proof artifacts comprise 72 CryptoLine files (4983045 mostly generated lines; 342 MB) and 12 HOL files (1677 lines).

The trusted computing base includes CryptoLine’s AArch64 semantics and proof backends, the HOL Light kernel and OCaml runtime, the functional correspondence between the assembly and its CryptoLine model, and the manual translation from CryptoLine contracts to HOL Light assumptions.

The two verification tools operate at different abstraction levels. CryptoLine is better suited to instruction-level algebraic and range reasoning, whereas HOL Light makes the high-level algebraic composition substantially easier to express and prove. This division avoids applying Gröbner elimination directly to the radix-expanded end-to-end computation. The downside is that the CryptoLine-to-HOL interface is maintained manually: there is no certificate exchange, and HOL Light does not re-prove the assembly semantics, roots, ranges, CRT uniqueness, or no-wrap properties established at the lower level.

## 5 Results

### 5.1 Benchmark Settings

To study architecture-dependent performance, we evaluate our implementations on Cortex-A72, Cortex-A76 [Arm18], Neoverse-N1, Neoverse-V1, and Neoverse-V2. The Cortex-A72

and Cortex-A76 measurements are obtained on Raspberry Pi 4 and Raspberry Pi 5, respectively, while the Neoverse measurements use AWS Graviton2, Graviton3, and Graviton4 instances. For cycle counting, we use the AArch64 benchmarking framework [Kan25] with Linux `perf` and report median cycle counts. For standalone big-integer multiplication, we use GMP 6.2.1 tuned separately for each target as the primary variable-time baseline and also benchmark its constant-time `mpn_sec_mul`. The tuned multiplication thresholds differ substantially across microarchitectures. In particular, the `MUL_FFT_THRESHOLD` values are 2688, 2176, 2176, 2752, and 2752 limbs on Cortex-A72, Cortex-A76, Neoverse-N1, Neoverse-V1, and Neoverse-V2, respectively. We also benchmark FLINT 3.6.0. On Cortex-A72 and Neoverse-N1, its performance is generally close to tuned GMP over the evaluated sizes, while on Cortex-A76 it is slower. We therefore use tuned GMP as the baseline here. On Neoverse-V1 and V2, FLINT is substantially faster than GMP, so we report these results in Appendix A. For Montgomery multiplication and RSA, we use OpenSSL 4.0.0-dev [Ope26]. On Cortex-A72, Cortex-A76, and Neoverse-N1, the Neon Montgomery path is the relevant optimized baseline. OpenSSL’s AArch64 dispatch enables its Neon RSA Montgomery path by default on Cortex-A72 and Neoverse-N1, but not on Cortex-A76. Since the Neon path is also faster than the scalar path on Cortex-A76 in our measurements, we explicitly enable it there and use it as the baseline. On Neoverse-V1 and Neoverse-V2, however, OpenSSL’s scalar path is faster than its Neon path, so we use the scalar implementation as the primary baseline and report the Neon results separately. All reported numbers for our implementation use SLOTHY-optimized assembly kernels [ABKK24, slo]. **F** and **S** denote the First- and Second-generation implementations, respectively. **F** denotes the **F**irst generation and **S** denotes the **S**econd.

## 5.2 Crossover Point with Big Number Multiplication over GMP

Table 6 presents the performance of our NTT-based big-integer multipliers compared with GMP for its variable-time `mpn_mul_n` and constant-time `mpn_sec_mul`. Here, “Ours (2 unknown)” denotes the general case where both operands are unknown at runtime, whereas “Ours (1 known)” allows one operand’s NTT representation to be precomputed.

**Table 6:** Performance for big-integer multiplication

| Target | Bits  | GMP (var.-time) | Ours (2 unk. / 1 known)           |                   | GMP (const.-time) |
|--------|-------|-----------------|-----------------------------------|-------------------|-------------------|
| A76    | 5632  | 22903           | <b>21472</b> / 15738 ( <b>S</b> ) | +6.25% / +31.28%  | 54221             |
|        | 8448  | 41051           | 43589 / 33734 ( <b>F</b> )        | −6.18% / +17.82%  | 122258            |
|        |       |                 | 34212 / 25781 ( <b>S</b> )        | +16.66% / +37.20% |                   |
|        | 11264 | 63750           | 59775 / 46619 ( <b>F</b> )        | +6.24% / +26.87%  | 217069            |
|        | 14080 | 88863           | 83234 / — ( <b>F</b> )            | +6.33% / —        | 339176            |
| A72    | 5632  | 28373           | <b>26111</b> / 19179 ( <b>S</b> ) | +7.97% / +32.40%  | 54293             |
|        | 8448  | 49517           | 54237 / 41407 ( <b>F</b> )        | −9.53% / +16.38%  | 123681            |
|        |       |                 | 45095 / 32326 ( <b>S</b> )        | +8.93% / +34.72%  |                   |
|        | 11264 | 74823           | 72726 / 59416 ( <b>F</b> )        | +2.80% / +20.59%  | 219315            |
|        | 14080 | 99443           | 100220 / — ( <b>F</b> )           | −0.78% / —        | 341618            |
| N1     | 5632  | 23268           | <b>21536</b> / 15768 ( <b>S</b> ) | +7.44% / +32.23%  | 54224             |
|        | 8448  | 41600           | 43083 / 32775 ( <b>F</b> )        | −3.56% / +21.21%  | 122051            |
|        |       |                 | 34331 / 25833 ( <b>S</b> )        | +17.47% / +37.90% |                   |
|        | 11264 | 62992           | 58847 / 45948 ( <b>F</b> )        | +6.58% / +27.06%  | 216860            |
|        | 14080 | 88699           | 81605 / — ( <b>F</b> )            | +8.00% / —        | 338921            |

The results show that the crossover against tuned GMP depends on both the microarchitecture and the implementation generation. At 5632 bits, the **Second** generation two-unknown-input multiplier reduces the cycle count by 6.25%, 7.97%, and 7.44% on

Cortex-A76, Cortex-A72, and Neoverse-N1, respectively. At 8448 bits, **First** is slower than GMP on all three processors, whereas **Second** improves the margin to 16.66%, 8.93%, and 17.47%, respectively. Thus, among the tested sizes, **Second** places the observed crossover below 5632 bits on all three processors.

Against GMP’s constant-time `mpn_sec_mul`, which remains on the schoolbook for all operand sizes, our two-unknown-input implementations require approximately 52–76% fewer cycles across the measured sizes.

### 5.3 Modular Multiplication over OpenSSL

Table 7 reports the performance of Montgomery multiplication at common RSA sizes on Cortex-A76, A72 and Neoverse-N1. We compare our approach against OpenSSL’s vectorized baseline `bn_mul_mont` [Ope26]. During benchmarking, the Neon-optimized path is explicitly selected by setting the control variable `ossl_OPENSSL_armv8_rsa_neonized` to 1. For the two-unknown-input case, the reported cost includes the additional forward transform required to preprocess the second runtime operand.

**Table 7:** Performance for Montgomery multiplication

| Target | Multiplier | Bits  | OpenSSL | Ours (2 unknown / 1 known)          |                   |
|--------|------------|-------|---------|-------------------------------------|-------------------|
| A76    | 5632-bit   | 5120  | 56597   | 65602 / 47190 ( <b>S</b> )          | −15.91% / +16.62% |
|        |            | 6144  | 80227   | 102707 / 76061 ( <b>S</b> )         | −28.02% / +5.19%  |
|        | 8448-bit   | 8192  | 139720  | 136410 / 99295 ( <b>F</b> )         | +2.37% / +28.94%  |
|        |            |       |         | <b>104329</b> / 77683 ( <b>S</b> )  | +25.33% / +44.40% |
|        | 11264-bit  | 10240 | 215609  | 186876 / 133390 ( <b>F</b> )        | +13.33% / +38.13% |
| A72    | 5632-bit   | 5120  | 60911   | 81796 / 59045 ( <b>S</b> )          | −34.29% / +3.06%  |
|        |            | 6144  | 85171   | 135434 / 100643 ( <b>S</b> )        | −59.02% / −18.17% |
|        | 8448-bit   | 8192  | 146178  | 170537 / 123704 ( <b>F</b> )        | −16.66% / +15.37% |
|        |            |       |         | <b>137550</b> / 102819 ( <b>S</b> ) | +5.90% / +29.66%  |
|        | 11264-bit  | 10240 | 223919  | 242817 / 176600 ( <b>F</b> )        | −8.44% / +21.13%  |
| N1     | 5632-bit   | 5120  | 56655   | 65814 / 47350 ( <b>S</b> )          | −16.17% / +16.42% |
|        |            | 6144  | 80260   | 103173 / 76429 ( <b>S</b> )         | −28.55% / +4.77%  |
|        | 8448-bit   | 8192  | 139742  | 134932 / 99591 ( <b>F</b> )         | +3.44% / +28.73%  |
|        |            |       |         | <b>104704</b> / 77964 ( <b>S</b> )  | +25.07% / +44.21% |
|        | 11264-bit  | 10240 | 215516  | 184094 / 133766 ( <b>F</b> )        | +14.58% / +37.93% |

The benefit of our Montgomery multiplication depends strongly on whether one operand can be precomputed and on how closely the fixed NTT kernel matches the operand size. The one-known-input variant performs particularly well when the kernel is well matched to the modulus. At 8192 bits, **Second** reduces the cycle count relative to OpenSSL Neon by 44.40%, 29.66%, and 44.21% on Cortex-A76, Cortex-A72, and Neoverse-N1, respectively. At 10240 bits, the **First**-generation 11264-bit kernel achieves reductions of 38.13%, 21.13%, and 37.93%.

The smaller operand sizes expose the cost of length utilization. For example, the 8448-bit kernel applied to 6144-bit operands remains 5.19% and 4.77% faster in the one-known-input case on Cortex-A76 and Neoverse-N1, respectively, but is 18.17% slower on Cortex-A72. The two-unknown-input case is more sensitive to this mismatch. At 8192 bits, replacing **First** with **Second** changes the margin over OpenSSL from 2.37% to 25.33% on Cortex-A76 and from 3.44% to 25.07% on Neoverse-N1. On Cortex-A72, the same transition changes a 16.66% slowdown into a 5.90% cycle reduction.

## 5.4 Integration into OpenSSL RSA

Table 8 reports the end-to-end performance of OpenSSL RSA verification and encryption on Cortex-A76, Cortex-A72, and Neoverse-N1 after integrating our NTT-based Montgomery multiplication. We focus on public-key operations only because these operations use full-modulus arithmetic. OpenSSL signing and decryption instead use CRT over approximately half-size factors, for which our fixed NTT kernels provide little benefit.

**Table 8:** Performance for different RSA sizes

| Target | Multiplier | Size  | Operation | Ours (F / S)             | OpenSSL (Neon / Scalar) | Improvement against OpenSSL Neon (F / S) |
|--------|------------|-------|-----------|--------------------------|-------------------------|------------------------------------------|
| A76    | 5632-bit   | 5120  | Verify    | — / 1070065              | 1131589 / 1413966       | — / +5.44%                               |
|        |            |       | Encrypt   | — / 1076692              | 1136681 / 1419155       | — / +5.28%                               |
|        | 8448-bit   | 6144  | Verify    | — / 1703906              | 1601669 / 2029822       | — / -6.38%                               |
|        |            |       | Encrypt   | — / 1710650              | 1607768 / 2035934       | — / -6.40%                               |
|        |            | 8192  | Verify    | 2275665 / <b>1741234</b> | 2788623 / 3596094       | +18.39% / +37.56%                        |
|        |            |       | Encrypt   | 2301371 / 1750415        | 2797371 / 3602484       | +17.73% / +37.43%                        |
|        | 11264-bit  | 10240 | Verify    | 3101060 / —              | 4299864 / 5605696       | +27.88% / —                              |
|        |            |       | Encrypt   | 3131536 / —              | 4308865 / 5615556       | +27.32% / —                              |
| A72    | 5632-bit   | 5120  | Verify    | — / 1363257              | 1213916 / 1425773       | — / -12.30%                              |
|        |            |       | Encrypt   | — / 1398264              | 1248442 / 1461142       | — / -12.00%                              |
|        | 8448-bit   | 6144  | Verify    | — / 2245255              | 1700662 / 2044146       | — / -32.02%                              |
|        |            |       | Encrypt   | — / 2306050              | 1741865 / 2086273       | — / -32.39%                              |
|        |            | 8192  | Verify    | 3228034 / <b>2289678</b> | 2918243 / 3614003       | -10.62% / +21.54%                        |
|        |            |       | Encrypt   | 3298764 / 2352440        | 2968606 / 3666342       | -11.12% / +20.76%                        |
|        | 11264-bit  | 10240 | Verify    | 4062384 / —              | 4463382 / 5630184       | +8.98% / —                               |
|        |            |       | Encrypt   | 4132804 / —              | 4525045 / 5692746       | +8.67% / —                               |
| N1     | 5632-bit   | 5120  | Verify    | — / 1070970              | 1132446 / 1414722       | — / +5.43%                               |
|        |            |       | Encrypt   | — / 1077670              | 1138700 / 1420806       | — / +5.36%                               |
|        | 8448-bit   | 6144  | Verify    | — / 1704725              | 1602582 / 2030856       | — / -6.37%                               |
|        |            |       | Encrypt   | — / 1712107              | 1609947 / 2037855       | — / -6.35%                               |
|        |            | 8192  | Verify    | 2458209 / <b>1742117</b> | 2788791 / 3595776       | +11.85% / +37.53%                        |
|        |            |       | Encrypt   | 2487498 / 1751388        | 2797490 / 3604822       | +11.08% / +37.39%                        |
|        | 11264-bit  | 10240 | Verify    | 3099824 / —              | 4300789 / 5607025       | +27.92% / —                              |
|        |            |       | Encrypt   | 3130438 / —              | 4311000 / 5618335       | +27.39% / —                              |

At RSA-8192, the **Second** generation implementation reduces verification and encryption cycle counts by about 37.5% on Cortex-A76 and Neoverse-N1, and by about 21% on Cortex-A72. Compared with **First**, **Second** changes the RSA-8192 margin from 18.39% to 37.56% on Cortex-A76, from a 10.62% slowdown to a 21.54% reduction on Cortex-A72, and from 11.85% to 37.53% on Neoverse-N1 for verification.

The benefit is smaller when the fixed NTT kernel is poorly matched to the RSA size. At RSA-6144, for example, the 8448-bit **Second** kernel remains slower than OpenSSL Neon on all three processors. At RSA-10240, the **First**-generation implementation reduces verification and encryption cycle counts by about 27–28% on Cortex-A76 and Neoverse-N1, but by only about 9% on Cortex-A72.

## 5.5 Discussion

The crossover observed here is higher than that reported by Becker et al. [BHK<sup>+</sup>22a]. Their implementation targets Cortex-M3/M55, where the balance between scalar multi-precision arithmetic and SIMD/NTT arithmetic differs substantially from the out-of-order Cortex-A and Neoverse cores considered here. Our comparison also uses target-tuned GMP but not the embedded baselines considered in that work. We therefore regard the crossover as architecture- and baseline-dependent.

SLOTHY is used in all of our kernels to reduce the burden of manual instruction scheduling. The evaluated kernels improve by 7–9% on Cortex-A76, more than 20% on Cortex-A72, about 10–11% on Neoverse-N1, 11–13% on Neoverse-V1, and 5–8% on Neoverse-V2.

The fixed-size specialization also incurs code and memory costs. Across the evaluated implementations, the combined executable code and constant tables occupy 40–84 KB, while peak stack usage together with the tables ranges from 27–53 KB. For the 8448-bit multiplier, **Second** reduces the combined code-and-table footprint from 58,396 to 51,896 bytes and the stack-plus-table footprint from 38,392 to 27,264 bytes. This reduction comes primarily from the smaller constant tables, since the executable code itself grows from 38,756 to 43,288 bytes. These costs are acceptable on the tested application-class cores.

## 6 Conclusion

Our study shows that the crossover of NTT-based large-integer multiplication on modern out-of-order Arm processors is not a fixed operand size, but depends on the microarchitecture, baseline implementation, and degree of fixed-size specialization. The **Second** generation implementation demonstrates that tighter arithmetic bounds can translate directly into lower implementation cost by eliminating reductions, while the OpenSSL experiments show that these gains can survive at the application level. Conversely, the Neoverse-V1/V2 and FLINT results expose the limits of the approach against stronger scalar baselines.

For the **First** generation implementation, CryptoLine and HOL Light provide complementary low- and high-level assurance of the optimized arithmetic pipeline. Overall, NTT multiplication is practical well below the regime in which general-purpose libraries normally switch to FFT methods.

## A Additional Results on Neoverse-V1 and Neoverse-V2

For multiplication, we additionally compare against FLINT 3.6.0, which provides a stronger baseline than tuned GMP on both processors.

**Table 9:** Performance for big-integer multiplication on Neoverse-V1 and V2

| Target | Bits  | Ours<br>(2 unk. / 1 known) | GMP (var.-time) |                   | FLINT  |                   | GMP<br>(const.-time) |
|--------|-------|----------------------------|-----------------|-------------------|--------|-------------------|----------------------|
|        |       |                            | Cycles          | Speedup           | Cycles | Speedup           |                      |
| V1     | 5632  | 10085 / 7890 ( <b>S</b> )  | 9464            | −6.56% / +16.63%  | 7234   | −39.41% / −9.07%  | 18076                |
|        | 8448  | 21998 / 17630 ( <b>F</b> ) | 16828           | −30.72% / −4.77%  | 14432  | −52.42% / −22.16% | 41739                |
|        | 11264 | 16762 / 13378 ( <b>S</b> ) | 26318           | +0.39% / +20.50%  | 22503  | −16.14% / +7.30%  | 75309                |
|        | 14080 | 28918 / 22923 ( <b>F</b> ) | 36854           | −9.88% / +12.90%  | 33433  | −28.51% / −1.87%  | 118539               |
|        | 14080 | 39422 / — ( <b>F</b> )     | 36854           | −6.97% / —        | 33433  | −17.91% / —       | 118539               |
| V2     | 5632  | 9820 / 7718 ( <b>S</b> )   | 8340            | −17.75% / +7.46%  | 6827   | −43.84% / −13.05% | 16957                |
|        | 8448  | 20463 / 16071 ( <b>F</b> ) | 15003           | −36.39% / −7.12%  | 13299  | −53.87% / −20.84% | 39390                |
|        | 11264 | 16042 / 12822 ( <b>S</b> ) | 23621           | −6.93% / +14.54%  | 21027  | −20.63% / +3.59%  | 71197                |
|        | 14080 | 27342 / 21115 ( <b>F</b> ) | 33416           | −15.75% / +10.61% | 31348  | −30.03% / −0.42%  | 112330               |
|        | 14080 | 36354 / — ( <b>F</b> )     | 33416           | −8.79% / —        | 31348  | −15.97% / —       | 112330               |

**Table 10:** Performance for Montgomery multiplication on Neoverse-V1 and V2

| Target | Multiplier | Bits      | OpenSSL<br>(Scalar) | Ours (2 unk. / 1 known)           | Speedup           | OpenSSL<br>(Neon) |
|--------|------------|-----------|---------------------|-----------------------------------|-------------------|-------------------|
| V1     | 5632-bit   | 5120      | 23152               | 32657 / 23632 ( <b>S</b> )        | −41.05% / −2.07%  | 34326             |
|        |            | 6144      | 33288               | 53109 / 39117 ( <b>S</b> )        | −59.54% / −17.51% | 47973             |
|        | 8448-bit   | 8192      | 58990               | 73152 / 53857 ( <b>F</b> )        | −24.01% / +8.70%  | 82105             |
|        |            | 11264-bit | 92673               | <b>53641</b> / 40230 ( <b>S</b> ) | +9.07% / +31.80%  | 125351            |
| V2     | 5632-bit   | 5120      | 22675               | 31048 / 22886 ( <b>S</b> )        | −36.93% / −0.93%  | 33963             |
|        |            | 6144      | 32618               | 50241 / 37487 ( <b>S</b> )        | −54.03% / −14.93% | 47595             |
|        | 8448-bit   | 8192      | 57849               | 66725 / 49144 ( <b>F</b> )        | −15.34% / +15.05% | 81481             |
|        |            | 11264-bit | 90333               | <b>51287</b> / 38532 ( <b>S</b> ) | +11.34% / +33.39% | 124501            |
|        |            |           |                     | 84643 / 59823 ( <b>F</b> )        | +6.30% / +33.78%  |                   |

**Table 11:** Performance for different RSA sizes on Neoverse-V1 and V2

| Target | Multiplier | Size  | Operation | Ours (F / S)     | OpenSSL<br>(Scalar) | Speedup          | OpenSSL<br>(Neon) |
|--------|------------|-------|-----------|------------------|---------------------|------------------|-------------------|
| V1     | 5632-bit   | 5120  | Verify    | — / 535406       | 363816              | — / −47.16%      | 685018            |
|        |            |       | Encrypt   | — / 537396       | 367836              | — / −46.10%      | 689468            |
|        | 8448-bit   | 6144  | Verify    | — / 875909       | 520022              | — / −68.44%      | 956897            |
|        |            |       | Encrypt   | — / 882592       | 524342              | — / −68.32%      | 963294            |
|        |            | 8192  | Verify    | 1223485 / 904388 | 915481              | −33.64% / +1.21% | 1634476           |
|        |            |       | Encrypt   | 1257430 / 911931 | 920885              | −36.55% / +0.97% | 1641914           |
|        | 11264-bit  | 10240 | Verify    | 1573053 / —      | 1424908             | −10.40% / —      | 2501421           |
|        |            |       | Encrypt   | 1616844 / —      | 1433709             | −12.77% / —      | 2520795           |
| V2     | 5632-bit   | 5120  | Verify    | — / 515165       | 352119              | — / −46.30%      | 676229            |
|        |            |       | Encrypt   | — / 519914       | 355666              | — / −46.18%      | 679489            |
|        | 8448-bit   | 6144  | Verify    | — / 837753       | 504208              | — / −66.15%      | 946280            |
|        |            |       | Encrypt   | — / 841752       | 507362              | — / −65.91%      | 950087            |
|        |            | 8192  | Verify    | 1120914 / 862223 | 889071              | −26.08% / +3.02% | 1621587           |
|        |            |       | Encrypt   | 1147164 / 867197 | 892651              | −28.51% / +2.85% | 1626356           |
|        | 11264-bit  | 10240 | Verify    | 1425086 / —      | 1381852             | −3.13% / —       | 2476623           |
|        |            |       | Encrypt   | 1455674 / —      | 1387525             | −4.91% / —       | 2482639           |

The V1/V2 results show a different regime from Cortex-A72, Cortex-A76, and Neoverse-N1. For two-unknown operands in multiplication, **Second** is approximately at parity with tuned GMP at 8448 bits on V1, but remains slower on V2. FLINT provides a stronger baseline on both processors and remains faster than our two-unknown implementations over all tested sizes. In the one-known-input case, however, the 8448-bit **Second** multiplier outperforms FLINT on both V1 and V2.

The Montgomery results again show the importance of operand-size matching and precomputation. At 8192 bits, **Second** outperforms OpenSSL’s scalar Montgomery implementation in both the two-unknown and one-known settings. These kernel-level gains translate much less strongly to end-to-end RSA: at RSA-8192, **Second** provides only marginal improvements over the scalar OpenSSL baseline, while the other evaluated RSA sizes remain slower.

## References

- [ABKK24] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. Fast and clean: Auditable high-performance assembly via constraint

- solving. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):87–132, 2024.
- [ACC<sup>+</sup>21] 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 Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, Nov. 2021.
- [Arm] Arm Ltd. *Arm Architecture Reference Manual for A-profile Architecture*, M.b edition. Document DDI 0487 M.b.
- [Arm18] Arm Ltd. Cortex-A76: Laptop-class performance with mobile efficiency, May 2018. Accessed: 2026-03-27.
- [Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology—CRYPTO ’86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1987.
- [Bec26] Hanno Becker. Neon NTT – (auto)formalised. IACR Cryptology ePrint Archive, Paper 2026/1223, 2026.
- [BHK<sup>+</sup>22a] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient multiplication of somewhat small integers using number-theoretic transforms. In Chen-Mou Cheng and Mitsuaki Akiyama, editors, *Advances in Information and Computer Security*, volume 13504 of *Lecture Notes in Computer Science*, pages 3–23, Cham, 2022. Springer.
- [BHK<sup>+</sup>22b] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022.
- [CCSCD<sup>+</sup>24] Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. Optimizations and practicality of high-security CSIDH. *IACR Communications in Cryptology*, 1(1), 2024.
- [CHK<sup>+</sup>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.
- [CLT<sup>+</sup>25] Chun-Ming Chiu, Jiaxiang Liu, Ming-Hsien Tsai, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Algebraic linear analysis for Number Theoretic Transform in lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):668–692, 2025.
- [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.
- [CYW25] Chun-Ming Chiu, Bo-Yin Yang, and Bow-Yaw Wang. A new trick for polynomial multiplication: A verified CRT polymul utilizing a monomial factor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):795–816, 2025.

- [DGPS24] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. *Singular 4-4-0—A Computer Algebra System for Polynomial Computations*, 2024. Version 4.4.0p7 in the verification container.
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
- [FLS<sup>+</sup>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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’19, pages 1591–1606, New York, NY, USA, 2019. Association for Computing Machinery.
- [fml] fmlab-iis. CryptoLine. <https://github.com/fmlab-iis/cryptoline>. GitHub repository, commit [8bba2e3c6425](#) (2026-02-03).
- [Goo58] I. J. Good. The interaction algorithm and practical fourier analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958.
- [GS66] W. Morven Gentleman and Gordon Sande. Fast fourier transforms—for fun and profit. In *Proceedings of the November 7–10, 1966, Fall Joint Computer Conference*, pages 563–578. Association for Computing Machinery, 1966.
- [GtGdt23a] Torbjörn Granlund and the GMP development team. *GNU MP 6.3.0 Manual: FFT Multiplication*, 6.3.0 edition, 2023.
- [GtGdt23b] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.3.0 edition, 2023. Current stable release 6.3.0, accessed 2026-04-03.
- [Har09] John Harrison. HOL Light: An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs ’09, pages 60–66, Berlin, Heidelberg, 2009. Springer-Verlag.
- [HLS<sup>+</sup>22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NIST-PQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022.
- [Hwa24] Vincent Hwang. A survey of polynomial multiplications for lattice-based cryptosystems. *IACR Communications in Cryptology*, 1(2), 2024.
- [KAK96] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.
- [Kan25] Matthias J. Kannwischer. aarch64-bench: Portable cycle-counting. <https://github.com/mkannwischer/aarch64-bench>, 2025. GitHub repository, latest visible commit e83d2d1 (2025-04-23), accessed 2026-04-03.
- [LLS<sup>+</sup>24] Li-Chang Lai, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Automatic verification of cryptographic block function implementations with logical equivalence checking. In *Computer Security – ESORICS 2024*, volume 14985 of *Lecture Notes in Computer Science*, pages 377–395. Springer, 2024.

- [LZ22] Zhichuang Liang and Yunlei Zhao. Number theoretic transform and its applications in lattice-based cryptosystems: A survey. *CoRR*, abs/2211.13546, 2022.
- [MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA cryptography specifications version 2.2. Technical Report RFC 8017, Internet Engineering Task Force, 2016.
- [Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
- [Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report FIPS 204, NIST, 2024.
- [Nat24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report FIPS 203, NIST, 2024.
- [Ope26] OpenSSL Project. OpenSSL 4.0.0-dev source tree. <https://github.com/openssl/openssl>, 2026. Version 4.0.0-dev vendored in the artifact at parent commit d0fb04b.
- [Pol71] J. M. Pollard. The fast fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [slo] slothy-optimizer. SLOTHY: Assembly optimization via constraint solving. <https://github.com/slothy-optimizer/slothy>. GitHub repository, accessed 2026-04-03.
- [Tri26] Alix Trieu. Formally verified number-theoretic transform. *IACR Communications in Cryptology*, 2(4), 2026.
- [Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software—ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.
- [Win78] Shmuel Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.
- [Yan26] Bo-Yin Yang. Note on number-theoretic transforms for implementers – butterflies, twisting, incompleteness, and good’s trick. IACR Cryptology ePrint Archive, Paper 2026/1546, 2026.