Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4

Amin Abdulrahman[1,2], Jiun-Peng Chen[3], Yu-Jia Chen[4], Vincent Hwang[3,5], Matthias J. Kannwischer[2,3] and Bo-Yin Yang[3]

[1] Ruhr University Bochum, Germany
amin.abdulrahman@rub.de
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
matthias@kannwischer.eu
[3] Academia Sinica, Taipei, Taiwan
jpchen@citi.sinica.edu.tw, vincentvbh7@gmail.com, by@crypto.tw
[4] IKV Technology, Taipei, Taiwan
yujia@email.ikv-tech.com.tw
[5] National Taiwan University, Taipei, Taiwan

Abstract. The U.S. National Institute of Standards and Technology (NIST) has designated ARM microcontrollers as an important benchmarking platform for its Post-Quantum Cryptography standardization process (NISTPQC). In view of this, we explore the design space of the NISTPQC finalist Saber on the Cortex-M4 and its close relation, the Cortex-M3. In the process, we investigate various optimization strategies and memory-time tradeoffs for number-theoretic transforms (NTTs).

Recent work by Chung et al. has shown that NTT multiplication is superior compared to Toom–Cook multiplication for unprotected Saber implementations on the Cortex-M4 in terms of speed. However, it remains unclear if NTT multiplication can outperform Toom–Cook in masked implementations of Saber. Additionally, it is an open question if Saber with NTTs can outperform Toom–Cook in terms of stack usage. We answer both questions in the affirmative. Additionally, we present a Cortex-M3 implementation of Saber using NTTs outperforming an existing Toom–Cook implementation. Our stack-optimized unprotected M4 implementation uses around the same amount of stack as the most stack-optimized implementation using Toom–Cook while being 33%-41% faster. Our speed-optimized masked M4 implementation is 16% faster than the fastest masked implementation using Toom–Cook. For the Cortex-M3, we outperform existing implementations by 29%-35% in speed.

We conclude that for both stack- and speed-optimization purposes, one should base polynomial multiplications in Saber on the NTT rather than Toom–Cook for the Cortex-M4 and Cortex-M3. In particular, in many cases, composite moduli NTTs perform best.

Keywords: NTT · Saber · Cortex-M4 · Cortex-M3 · NISTPQC

1 Introduction

Shor's algorithm [Sho97] threatens all widely deployed public-key cryptography as it solves the integer factorization and the discrete logarithms on a quantum computer. Therefore, NIST has called for proposals to replace their existing standards for digital signatures and key encapsulation mechanisms (KEMs) [NIS]. We are currently in the third round of the process, where 7 finalist schemes and 8 alternate schemes remain [AASA+20]. Of the 7 finalists, 4 are KEMs: Classic McEliece [ABC+20], a code-based scheme, plus

Kyber [ABD+20b], NTRU [CDH+20], and Saber [DKRV20], which are all lattice-based with similar performance characteristics.

Saber is based on the module learning with rounding (M-LWR) problem. Its arithmetic operates in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ with $q = 2^{13}$ and $n = 256$. One of Saber's distinguishing features, compared to its close relative Kyber [ABD+20b], is the power-of-two modulus $q = 2^{13}$, while Kyber uses the prime modulus 3329. Using a power-of-two modulus has benefits, but also a major disadvantage in being less suitable for the number-theoretic transform (NTT). Recent work by Chung et al. [CHK+21] has shown that Saber can still profit from NTT multiplications by switching to a larger prime modulus allowing NTTs. Indeed, Saber with NTTs can also be significantly faster than Toom–Cook on the major NIST software targets: ARM Cortex-M4 and Haswell with AVX2.

We address three questions in this paper:

1. The Chung et al. [CHK+21] implementation, not optimized for stack usage, has a large memory footprint. *How well can NTT-based Saber perform stack-wise on the Cortex-M4*, and in particular, can we achieve a smaller memory footprint for Saber with NTTs compared to the stack-optimized Toom implementation from [MKV20]?

2. The [CHK+21] implementation relies on one of the multiplicands being small and only computes the correct 25-bit result. This is normally true for the secrets in Saber, but it does not apply to masked implementations in which the secret is arithmetically shared modulo q (e.g., [VBDK+20]). *How does Saber with NTTs perform for masked implementations*, in particular can they outperform masked Toom-based Saber from [VBDK+20] in speed and stack usage?

3. While the Cortex-M4 is the primary microcontroller optimization target of NIST, its cheaper predecessor, the Cortex-M3 remains widely deployed, e.g., in hardware security modules (HSMs) like the STA1385[1]. However, the Cortex-M3 is slightly less powerful than the Cortex-M4 especially in terms of features critical to polynomial multiplication. In particular, long multiplications `smull` and `smlal` are not executed in constant time and, consequently, cannot be safely used when handling secret data. The Cortex-M4 implementation heavily relies on these instructions. So the open question is: *Should Saber implementations targeting the Cortex-M3 use NTTs?*

For Question 1, we first optimized Saber with NTTs for stack usage *without sacrificing speed* on Cortex-M4 and achieved a significantly better memory footprint than speed-optimized Toom–Cook implementations from [MKV20].

Then, as this still uses more memory than the stack-optimized Toom–Cook from [MKV20], we propose an alternate approach of NTTs with a composite modulus $q' = q_1 q_2$, where coprime q_1 and q_2 are chosen such that NTTs are defined modulo q_1 and q_2. In this way we can define an NTT modulo q', which allows a very stack efficient implementation *competitive in memory usage and at least 30% faster compared to the most stack-optimized Toom–Cook implementations.*

We answer Question 2 in the affirmative by doing our polynomial multiplication via an NTT with a composite 36-bit modulus, which is sufficiently large for the masked product. We do this by combining 32-bit NTTs with 16-bit NTTs.

Finally, we answer Question 3 also in the affirmative. Here we have two natural alternatives in NTT-based polynomial multiplication using only 16-bit multiplications. One can use 32-bit NTTs but emulate the long multiplications (used already to implement Dilithium which requires 32-bit NTTs [GKS21]). Or one can adopt the approach of the AVX2 implementation of [CHK+21] and use two 16-bit NTTs which can be efficiently

[1] https://www.st.com/en/automotive-infotainment-and-telematics/sta1385.html

implemented while avoiding long multiplications. The result is then recombined using the Chinese remainder theorem (CRT). We show that both approaches are faster than Toom–Cook and the latter approach is the fastest. Furthermore, we also show that stack optimization on Cortex-M4 can be applied to the 16-bit NTT approach on Cortex-M3.

Contribution. We show that, for the Cortex-M3 and Cortex-M4, Toom–Cook is not useful for implementing Saber, and one should always use NTT multiplications. Firstly, the fastest NTT-based Saber implementations use less memory than the fastest Toom–Cook implementations. Secondly, the most stack-efficient implementations are using NTTs. Thirdly, we exhibit two NTT-based Saber implementation on the Cortex-M3, both outperforming Toom–Cook. Lastly, masked Saber implementations are also best implemented using NTTs regardless of whether we value speed, memory *or both*.

In the process, we point out an overlooked stack optimization with multi-moduli NTTs. The optimization justifies an unconventional use of composite-modulus for unmasked Saber and unequal-size NTTs for masked Saber that are not implemented before. Furthermore, we correct a misunderstanding regarding negacyclic convolutions by providing the actual if-and-only-if condition.

Lastly, we justify the use of CT butterflies for the inverse of negacyclic NTTs

Code. All our implementation are open source and available at `https://github.com/multi-moduli-ntt-saber/multi-moduli-ntt-saber`.

Related work. There is a line of works optimizing Saber for the Cortex-M4 [KRS19, MKV20, CHK$^+$21] using Karatsuba, Toom–Cook, and lately also NTTs. A masked Saber is presented by Van Beirendonck et al in [VBDK$^+$20]. Other NISTPQC third-round candidates have been implemented for the Cortex-M3 and M4. The ones most relevant to us are the constant-time NTTs from Greconici et al. [GKS21] and the stack optimizations by Botros et al. [BKS19]. Composite modulus NTTs were earlier studied in the context of side-channel protections for lattice-based schemes by Heinz and Pöppelmann [HP21].

Structure of the paper. This paper is structured as follows: Section 2 introduces Saber, ARM Cortex-M4 and Cortex-M3, and Montgomery multiplication. In Section 3, we present mathematics for NTTs implemented in this paper. In Section 4, we go through implementation details of `MatrixVectorMul` with different emphases. In Section 5, we present the performance of our implementations, and give some t-test results.

2 Preliminaries

2.1 Saber

Saber [DKRV20] is a NISTPQC finalist candidate lattice-based key encapsulation mechanism. It is based on the Module Learning With Rounding (M-LWR) problem on the ring $R_q = \mathbb{Z}_q[x]/\langle x^{256} + 1\rangle$. For all parameter sets $q = 2^{13}$ and $n = 256$.

Table 1: Saber Parameter Sets

name	l	$T = 2^{\epsilon_T}$	μ
`Lightsaber`	2	2^3	10
`Saber`	3	2^4	8
`Firesaber`	4	2^6	6

Algorithm 1 Saber Key Generation

Output: $pk = (\text{seed}_A, b), sk = (s)$

$\text{seed}_A \leftarrow \texttt{Sample}_U()$

$A \in R_q^{l \times l} \leftarrow \texttt{Expand}(\text{seed}_A)$

$s \in R_q^l \leftarrow \texttt{Sample}_B()$

$b \leftarrow \texttt{Round}(A^T \cdot s)$

Algorithm 3 Saber CPA Decryption

Input: $ct = (c, b'), sk = (s)$

Output: m

$v \leftarrow b'^T(s \mod p)$

$m \leftarrow \texttt{Round}(v - 2^{\epsilon_p - \epsilon_T} c \mod p)$

Algorithm 2 Saber CPA Encryption

Input: $m, r, pk = (\text{seed}_A, b)$

Output: $ct = (c, b')$

$A \in R_q^{l \times l} \leftarrow \texttt{Expand}(\text{seed}_A)$

$s' \in R_q^l \leftarrow \texttt{Sample}_B(r)$

$b' \leftarrow \texttt{Round}(As')$

$v' \leftarrow b^T(s' \mod p)$

$c \leftarrow \texttt{Round}(v' - 2^{\epsilon-1} m)$

Algorithms 1–3 are the CPA-secure scheme's keygen, encryption, and decryption and follow the submission material [DKRV20]. Here $\texttt{Sample}_U$ samples from the uniform distribution, $\texttt{Sample}_B$ samples from a binomial distribution, and `Expand` expands a seed to a uniform matrix of polynomials.

Saber's most time-consuming operation in key generation and encryption is the matrix-vector multiplication of polynomials $A^T \cdot s$ and As'. In decryption the most expensive operation is the inner product of $b'^T \cdot s$. We do not further discuss Saber's CCA-secure KEM construction, which uses a variant of the Fujisaki-Okamoto (FO) transform due to Hofheinz-Hövelmanns-Kiltz [HHK17]. We do note that Saber does require re-encryption in the decapsulation algorithm, and, therefore, improving the encryption also improves decapsulation.

Parameters. The module dimension l, the rounding parameter T, and the secret distribution parameter μ varies according to the parameter sets `Lightsaber`, `Saber`, and `Firesaber` (respectively targeting the NIST security levels 1, 3, and 5). See Table 1 for a summary. Hence, `MatrixVectorMul` is computing the product of an $l \times l$ matrix and an $l \times 1$ vector, whereas `InnerProd` is computing the inner product of two $l \times 1$ vectors.

2.2 ARM Cortex-M4 and Cortex-M3

The ARM Cortex-M4 is selected by NIST as a standard embedded platform to evaluate candidates (including Saber) in the NISTPQC process. For both scientific curiosity and practical reasons, we also implement Saber on the cheaper and also common Cortex-M3 to explore the variation in performance when some instructions are not supported or can only be used for secret-unrelated computations. The Cortex-M4 implements the ARMv7E-M architecture, some of the most prominent features are as follows:

- **14 General purpose registers.** There are 16 registers, named `r0`–`r15`. Except for the stack pointer (`r13`) and the program counter (`r15`), all other registers are general purpose registers.

- **Floating-point registers.** There are 32 single-precision floating-point registers that can also be used as a low-latency cache (cf. [ACC+21, CHK+21].)
- **Cycles for load and store instructions.** Store instructions are always one cycle. A sequence of h loads with no dependency is always $h+1$ cycles.
- **Single cycle long multiplications.** Long multiplications `{u,s}mull` and their accumulating counterparts `{u, s}mlal` are always one cycle.
- **Barrel shifter.** Shifts and rotates (`asr`, `lsl`, `lsr`, and `ror`), come at no extra cost when used as the "flexible second operand" of a standard data-processing instruction.
- **SIMD instructions.** Arithmetic instructions operating on registers as chunks of 8-bit or 16-bit elements. `{u,s}{add,sub}{8,16}` add up elements as packed 8-bit or 16-bit elements. `smul{b,t}{b,t}` multiply specified halves of registers. `smla{b,t}{b,t}` accumulates products of specified halves of registers into a register. `smlad{x}` accumulates two $16 \times 16 = 32$-bit multiplications into a register. `pkh{bt,tb}` pack two half words into a word.

The ARM Cortex-M3 implements the ARMv7-M architecture. The most important differences between Cortex-M3 and Cortex-M4 regarding constant-time implementation of Saber with NTTs are as follows [ARM10]:

- **No floating-point registers.** There is no FPU, hence, we will experience more overhead when spilling registers.
- **Early-terminating long multiplications.** Long multiplications (and the variants with accumulation) `{u,s}mull`, `{u,s}mlal` are early-terminating instructions that cannot be used for computing on secret data.
- **No SIMD instructions.** There are no operations either treating registers as packed 8-bit or 16-bit elements or operating on specific halves of operands.

2.3 Montgomery multiplication

We employ Montgomery multiplication for computing $\mathtt{mMul}(a, b\mathtt{R} \bmod {}^{\pm}\mathtt{Q}) = ab \bmod {}^{\pm}\mathtt{Q}$ [Mon85] where b is a known constant, $\mathtt{R}$ is architecture-friendly and coprime to $\mathtt{Q}$, and $\mathrm{mod}^{\pm}$ is signed modular reduction giving values in $\left[-\frac{\mathtt{Q}}{2}, \frac{\mathtt{Q}}{2}\right)$. The computation of $ab \bmod {}^{\pm}\mathtt{Q}$ is

$$ab \bmod {}^{\pm}\mathtt{Q} = \mathrm{hi}\left(a \cdot (b\mathtt{R} \bmod {}^{\pm}\mathtt{Q}) + \mathtt{Q} \cdot \mathrm{lo}\left(\mathtt{Qprime} \cdot \mathrm{lo}\left(a \cdot (b\mathtt{R} \bmod {}^{\pm}\mathtt{Q})\right)\right)\right)$$

where $\mathtt{Qprime} = -\mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$, and lo and hi are extraction of the lower $\log_2 \mathtt{R}$ bits and upper $\log_2 \mathtt{R}$ bits, respectively. In our implementations, we use either $\mathtt{R} = 2^{16}$ or $\mathtt{R} = 2^{32}$.

3 Number-Theoretic Transform

Number-theoretic transforms (NTTs) are critically important for efficient long multiplications. The most important works on integer multiplication [SS71, Für09, HVDH21] use NTTs as basic building blocks. NTTs are so critical to the performance of polynomial multiplications that the NISTPQC 3rd round candidates Dilithium, Falcon, and Kyber wrote NTTs into their specs [ABD+20b, ABD+20a, FHK+17]. In addition, the candidates NTRU, NTRU Prime, and Saber [DKRV20, CDH+20, BBC+20] can be sped up using NTTs [ACC+21, CHK+21].

In this section, we go over the mathematics for NTTs in their abstract form while maintaining the consistency of notations in our implementation details in Section 4. We provide the definitions as they are only required to be and do not attach unnecessary restrictions on them. All the formulations are known in the literature with various abstractions. But we give a clear explanation why NTT with a composite modulus is defined in such a way by relating the principal n-th roots of unity to CRT in Section 3.2.1. The justification trivially follows from the definitions without expanding the double summation. Furthermore, we point out an overlooked implementation aspect with multi-moduli NTTs in Sections 3.2.4 and 3.2.5.

An invertible NTT over $\mathbb{Z}_m[x]/\langle x^n - \zeta^n\rangle$ is defined if and only if the following conditions are satisfied:

1. Divisibility: Suppose m admits the prime factorization $m = p_0^{d_0} p_1^{d_1} \cdots p_{k-1}^{d_{k-1}}$, then n must divide $\mathbf{0}(m) \coloneqq \gcd(p_0 - 1, p_1 - 1, \ldots, p_{k-1} - 1)$ [AB74, Theorem 1.].

2. Invertibility: ζ must be invertible [CF94].

Condition 1. enables NTTs over $\mathbb{Z}_m[x]/\langle x^n - 1\rangle$ and Condition 2. allows the extension of the definition to $\mathbb{Z}_m[x]/\langle x^n - \zeta^n\rangle$. With only size-1 NTTs possible, Saber's coefficient ring is unfriendly for NTTs.

A closer look at the Chinese Remainder Theorem (CRT). Let R be a commutative ring, I_i be ideals of R so that $I_i + I_j = R$ for $i \neq j$, and δ be Kronecker delta. Section 3 is all about the CRT in the abstract sense that the formulae are various instantiations of the isomorphism:

$$\phi : R/\left(\bigcap_{i=0}^{n-1} I_i\right) \to \prod_{i=0}^{n-1} R/I_i, \quad \phi : a + \left(\bigcap_{i=0}^{n-1} I_i\right) \mapsto (a + I_0, a + I_1, \ldots, a + I_{n-1}) \tag{1}$$

[Für09, Theorem 2.4]. The inverse can be written as

$$\phi^{-1} : \prod_{i=0}^{n-1} R/I_i \to R/\left(\bigcap_{i=0}^{n-1} I_i\right), \quad \phi^{-1} : (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}) \mapsto \sum_{i=0}^{n-1} r_i \hat{a}_i \tag{2}$$

where the unique $(r_0, r_1, \ldots, r_{n-1})$ satisfies $r_i \bmod I_j = \delta_{ij}$ and $\sum_{i=0}^{n-1} r_i = 1$ [Bou89, Proposition 10 - (b), Section 8.11, Chapter I]. We will then review how the divisibility and invertibility conditions translate into ϕ and ϕ^{-1}.

3.1 Explicit Chinese remainder theorem computations

Explicitly computing a number from its remainders modulo a small number of coprime moduli q_i is an "Explicit Chinese Remainder Theorem" computation. There are basically two known algorithms: [MS90, Theorem 23] which resembles Lagrangian interpolation, and [CHK+21, Theorem 1] which resembles more divided-difference interpolation.

We follow the latter here. Let q, q_0, q_1 be pairwise co-prime and $m_1 := q_0^{-1} \bmod^{\pm} q_1$. For the system $u \equiv u_0 \pmod{q_0}$, $u \equiv u_1 \pmod{q_1}$, where $|u_0| < q_0/2$, $|u_1| < q_1/2$, $|u| < q_0 q_1/2$, solutions of u and $u \bmod^{\pm} q$, are explicitly given by:

$$\begin{aligned} u &= u_0 + \big((u_1 - u_0) m_1 \bmod^{\pm} q_1\big)\, q_0 \\ u \bmod^{\pm} q &= \big(u_0 + \big(\big((u_1 - u_0) m_1 \bmod^{\pm} q_1\big) \bmod^{\pm} q\big) \cdot q_0\big) \bmod^{\pm} q. \end{aligned}$$

3.2 NTT over an integer ring

3.2.1 Explicit formulations for NTTs

In [AB74], the divisibility condition $n|\mathbf{0}(m)$ was established for NTTs over arbitrary $\mathbb{Z}_m$. Let $[n]_q = \sum_{i=0}^{n-1} q^i$ be the q-analog[2] of n so $[n]_x = \sum_{i=0}^{n-1} x^i \in \mathbb{Z}_m[x]$. To arrive at a definition more constructively, if $n|\mathbf{0}(m)$ then n is invertible in $\mathbb{Z}_m$ and we can always choose a principal n-th root of unity ω giving $\mathtt{NTT}_{n:1:\omega}$ as follows

$$\mathtt{NTT}_{n:1:\omega} : \begin{cases} \mathbb{Z}_m[x]/\langle x^n - 1\rangle & \to \prod_{i=0}^{n-1} \left(\mathbb{Z}_m[x]/\langle x - \omega^i\rangle\right) \\ \boldsymbol{a}(x) & \mapsto \left(\boldsymbol{a}(1), \boldsymbol{a}(\omega), \ldots, \boldsymbol{a}(\omega^{n-1})\right) \end{cases} \tag{3}$$

[Für09] along with its inverse $\mathtt{NTT}^{-1}_{n:1:\omega}$ defined as below (where $\boldsymbol{r}_i = \frac{1}{n}[n]_{\omega^{-i}x}$).

$$\mathtt{NTT}^{-1}_{n:1:\omega} : \begin{cases} \prod_{i=0}^{n-1} \left(\mathbb{Z}_m[x]/\langle x - \omega^i\rangle\right) & \to \mathbb{Z}_m[x]/\langle x^n - 1\rangle \\ (\hat{a_0}, \hat{a_1}, \ldots, \hat{a_{n-1}}) & \mapsto \sum_{i=0}^{n-1} \boldsymbol{r}_i \hat{a_i} \end{cases} \tag{4}$$

A principal n-th root of unity ω is an n-th root of unity satisfying the orthogonality

$$\frac{1}{n}[n]_{\omega^i} = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{for } 1 \le i < n \end{cases} \text{[AB74, Equation (13)].}$$

Since $\boldsymbol{a}(x) \bmod (x - \omega^i) = \boldsymbol{a}(\omega^i)$, $\boldsymbol{r}_i \bmod (x - \omega^j) = \delta_{ij}$, and $\sum_{i=0}^{n-1} \boldsymbol{r}_i = 1$, we see that $\mathtt{NTT}_{n:1:\omega}$ and $\mathtt{NTT}^{-1}_{n:1:\omega}$ are just the polynomial formulation of ϕ and ϕ^{-1}.

We also note that $\mathtt{NTT}_{n:1:\omega}$ and $\mathtt{NTT}^{-1}_{n:1:\omega}$ directly carry over to finite commutative rings with identity 1 given the invertibility of $n = [n]_1 = \underbrace{1 + 1 + \cdots + 1}_{n}$ as a ring element. We suggest interested readers to refer to [DV78, Theorem 4.] on how the condition $n|\mathbf{0}(m)$ can be generalized to arbitrary finite rings.

3.2.2 Differentiating between principal and primitive n-th roots of unity

A primitive n-th root of unity is a ρ such that for every $0 \le i < n$, $\rho^i \ne 1$ and $\rho^n = 1$. In $\mathbb{Z}_m$ for a prime m, primitive and principal n-th roots of unity coincide. On the other hand, there are easy counterexamples for composite m. First consider $m = p^r$ a prime power, say for $\mathbb{Z}_9$. 4 is a primitive but not principal third root of unity. Moreover, 45 is a primitive but not principal 2048-th root of unity in $\mathbb{Z}_{8192}$. Then take $m = p_0 p_1$ for distinct primes p_0 and p_1, say $\mathbb{Z}_{15}$. Here 7 is a primitive but not principal fourth root of unity. One can construct more via the definition of Carmichael's lambda function [Car14, Chapter 5.8].

3.2.3 NTTs in composite coefficient rings via CRT

Suppose $m = q_0 q_1$ with coprime q_0 and q_1. Clearly n^{-1} exists in $\mathbb{Z}_{q_0 q_1}$ iff n^{-1} exists in both $\mathbb{Z}_{q_0}$ and $\mathbb{Z}_{q_1}$. Also from the definition of $\mathbf{0}$, $n|\mathbf{0}(m)$ implies $n|\mathbf{0}(q_0)$ and $n|\mathbf{0}(q_1)$. This means that ω being a principal n-th root of unity in $\mathbb{Z}_{q_0 q_1}$ is equivalent to ($\omega \bmod q_0$) and ($\omega \bmod q_1$) being principal roots in $\mathbb{Z}_{q_0}$ and $\mathbb{Z}_{q_1}$, respectively. The converse is also true that if $\omega_0 \in \mathbb{Z}_{q_0}$ and an $\omega_1 \in \mathbb{Z}_{q_1}$ are principal roots, we can find a principal root $\omega \in \mathbb{Z}_{q_0 q_1}$ via an explicit CRT computation from $\omega \equiv \omega_0 \pmod{q_0}$, $\omega \equiv \omega_1 \pmod{q_1}$.

3.2.4 Multi-moduli to save memory

There is an often overlooked implementation aspect of multi-moduli NTTs on the ARM Cortex-M4: Let q_0 and q_1 be coprime moduli for 16-bit NTTs, then we can compute an

[2] q-analog is frequently used in Combinatorics. In some sense, it is a symbolic generalization of n – we start by seeing $n = \underbrace{1 + 1 + \cdots + 1}_{n}$ and replacing each 1 with q^i in a symbolic fashion. q-factorials and q-binomial coefficients naturally have some combinatorial interpretations.

NTT over $\mathbb{Z}_{q_0q_1}$. Due to M4's powerful 1-cycle long multiplications, a 32-bit NTT over q_0q_1 easily outpaces $2 \times$ 16-bit NTTs. Indeed 16-bit NTT $<$ 32-bit NTT $\ll 2 \times$ 16-bit NTTs in cycle counts. We can, thus, reduce stack usage without sacrificing performance.

Suppose we want to multiply two size-n polynomials where each coefficient of the result is smaller than the product of k 16-bit primes, then the following approach only needs $16(k+1) \times n/8 = 2n(k+1)$ bytes of storage. We first note that this memory usage can be achieved with k distinct 16-bit NTTs by interleaving the computation. However, the fact that one 32-bit NTT being significantly faster than two 16-bit NTTs means we should replace every two 16-bit NTTs with a 32-bit NTT. If k is odd, then we can process the multiplicands by computing $\frac{k-1}{2}$ 32-bit NTTs and one 16-bit NTT for each. If k is even, for the first multiplicand, we compute $\frac{k}{2}$ 32-bit NTTs and transform the last one into the result of two 16-bit NTTs, while for the second multiplicand, we compute $\frac{k}{2} - 1$ 32-bit NTTs and two 16-bit NTTs.

3.2.5 Prior uses of multi-moduli

RNS (residue number system) is used in the context of homomorphic encryption for computing NTTs over many primes $p_0, p_1, \dots, p_{k-1}$ and the result in $\mathbb{Z}_{p_0p_1\cdots p_{k-1}}$ for speed. To use the Explicit CRT *a la* [MS90, Theorem 23], the representation is usually redundant. Here we use only two 16-bit prime moduli (non-redundantly) for reducing stack usage and jumping between the rings as shown in Section 4. In [HP21], the authors essentially used RNS to protect linear computation from side-channel attacks. They lift $\mathbb{Z}_{p_0}$ to $\mathbb{Z}_{p_0p_1}$, and compute NTT over $\mathbb{Z}_{p_0p_1}$ for fault protection. Our approach is to switch to $\mathbb{Z}_{p_0p_1}$ for speed and to $\mathbb{Z}_{p_0}$ and $\mathbb{Z}_{p_1}$ for saving memory. We will detail when to switch which way later.

3.3 Polynomial multiplication

Let $\psi \in \mathbb{Z}_m$. Polynomial multiplication modulo $x^n - \psi$ means computing $\boldsymbol{a}(x)\boldsymbol{b}(x)$ with the agreement that $x^n = \psi$ so $\boldsymbol{a}(x)\boldsymbol{b}(x) \bmod (x^n - \psi)$ is $\sum_{i=0}^{n-1} c_i x^i$ where

$$c_i = \left(\sum_{j=0}^{i} a_j b_{i-j} + \psi \sum_{j=i+1}^{n-1} a_j b_{i-j+n}\right).$$

If $\psi = 1$ then it is called cyclic convolution, and if $\psi = -1$ then it is called negacyclic convolution. In Saber, we are computing negacyclic convolutions with $n = 256$.

3.4 Discrete weighted transform

We review how to apply the discrete weighted transform (DWT) to negacyclic convolutions, and in general, polynomial multiplication modulo $x^n - \zeta^n$ for an invertible ζ. In [CF94], DWT is given as "introducing a weight signal to compute weighted convolution". In our context, the weight signal are powers $\left(1, \zeta, \dots, \zeta^{n-1}\right)$ of a scalar ζ [CF94, Equation (2.13)]. So we will use the notation of NTT subscripted both with ζ and ω for this DWT.

An implementation of $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathbb{Z}_m[x]/\langle x^n - \zeta^n\rangle$ when $n|\mathbf{0}(m)$ and ζ^{-1} exists, is $\mathtt{NTT}^{-1}_{n:\zeta:\omega}(\mathtt{NTT}_{n:\zeta:\omega}(a)(\cdot)_n\mathtt{NTT}_{n:\zeta:\omega}(b))$ [CF94, Equation (2.15)] where $(\cdot)_n$ is n-long point-wise multiplication and:

$$\mathtt{NTT}_{n:\zeta:\omega} : \begin{cases} \mathbb{Z}_m[x]/\langle x^n - \zeta^n\rangle & \to \prod_{i=0}^{n-1} \left(\mathbb{Z}_m[x]/\langle x - \zeta\omega^i\rangle\right) \\ \boldsymbol{a}(x) & \mapsto \left(\boldsymbol{a}(\zeta), \boldsymbol{a}(\zeta\omega) \dots, \boldsymbol{a}(\zeta\omega^{n-1})\right) \end{cases} \tag{5}$$

$$\mathtt{NTT}^{-1}_{n:\zeta:\omega} : \begin{cases} \prod_{i=0}^{n-1} \left(\mathbb{Z}_m[x]/\langle x - \zeta\omega^i\rangle\right) & \to \mathbb{Z}_m[x]/\langle x^n - \zeta^n\rangle \\ (\hat{a_0}, \hat{a_1}, \dots, \hat{a_{n-1}}) & \mapsto \sum_{i=0}^{n-1} \boldsymbol{r}_i \hat{a_i} \end{cases} \tag{6}$$

[CF94, Equations (2.5) – (2.6)], where $\boldsymbol{r}_i = \frac{1}{n}[n]_{\zeta^{-1}\omega^{-i}x}$. Furthermore, $\mathtt{NTT}_{n:\zeta:\omega}$ and $\mathtt{NTT}^{-1}_{n:\zeta:\omega}$ are also valid if we replace $\mathbb{Z}_m$ by a finite commutative ring.

If $n = 2^k$ and $\zeta^{2^k} = -1$, then ζ^2 is a principal 2^k-th root of unity. By setting $\omega = \zeta^2$, the negacyclic NTTs of Kyber and Dilithium, which are exactly the upper halves of standard NTTs, are special cases of $\mathtt{NTT}_{n:\zeta:\omega}$ and $\mathtt{NTT}^{-1}_{n:\zeta:\omega}$. But notice that our definitions are generic as in [CF94] because we simply aim to compute negacyclic convolutions, and there is no fundamental reason for ζ to be tied with ω. E.g., size-8 NTTs over $\mathbb{Z}_{17}[x]/\langle x^8+1\rangle$ defined by any combinations of (ζ, ω) from $\{3,5,6,7,10,11,12,14\} \times \{2,8,9,15\}$ fulfill the need for negacyclic convolution. Additionally, by setting $\zeta = 1$, one can obtain the cyclic versions $\mathtt{NTT}_{n:1:\omega}$ and $\mathtt{NTT}^{-1}_{n:1:\omega}$.

Let $\circ$ denote the composition so $(f \circ g)(x) = f(g(x))$, then $\mathtt{NTT}_{n:\zeta:\omega} = \mathtt{NTT}_{n:1:\omega} \circ (x \mapsto \zeta y)$ where $x \mapsto \zeta y$, termed "twisting" [Ber], transforms $\pmod{x^n - \zeta^n}$ to $\pmod{y^n - 1}$ and has the obvious inverse $y \mapsto \zeta^{-1} x$.

3.5 Cooley–Tukey and Gentleman–Sande FFTs

Two main algorithms to compute radix-2 NTTs are Cooley–Tukey and Gentleman–Sande FFTs. Cooley–Tukey FFT refers to computing with Cooley–Tukey butterfly (CT butterfly): for a pair (a_0, a_1) and a constant c, map $((a_0, a_1), c)$ to $(a_0 + ca_1, a_0 - ca_1)$ [CT65]. Gentleman–Sande FFT refers to computing using the Gentleman–Sande butterfly (GS butterfly): map $((a_0, a_1), c)$ to $(a_0 + a_1, (a_0 - a_1)c)$ [GS66].

Obviously, $\mathrm{GS}\left(\mathrm{CT}(a_0, a_1, c), c^{-1}\right) = 2(a_0, a_1) = \mathrm{CT}\left(\mathrm{GS}(a_0, a_1, c), c^{-1}\right)$. This observation suggests that any computation composed of CT and GS butterflies can be inverted by inverting the CT and GS butterflies and then canceling the scaling by a power of 2. There are at least two ways of implementing both $\mathtt{NTT}_{n:\zeta:\omega} = \mathtt{NTT}_{n:1:\omega} \circ (x \mapsto \zeta y)$ and $\mathtt{NTT}^{-1}_{n:\zeta:\omega} = (y \mapsto \zeta^{-1} x) \circ \mathtt{NTT}^{-1}_{n:1:\omega}$ described in the previous section.

In this section, we fix $n = 2^k$, $2^k | \mathbf{0}(m)$, and ω a principal 2^k-th root of unity. We describe the case where ζ only needs to be invertible.

3.5.1 CT for NTT and GS for iNTT

Computing $\mathtt{NTT}_{2^k:\zeta:\omega}$ with CT butterflies is mapping

$$\mathbb{Z}_m[x]/\left\langle x^{2^i} - \zeta^{2^i} \right\rangle \text{ to } \mathbb{Z}_m[x]/\left\langle x^{2^{i-1}} - \zeta^{2^{i-1}} \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{i-1}} - \zeta^{2^{i-1}} \omega^{2^{k-1}} \right\rangle,$$

which, when applied recursively, results in the bit-reversal of

$$\mathbb{Z}_m[x]/\langle x - \zeta \rangle \times \mathbb{Z}_m[x]/\langle x - \zeta\omega \rangle \times \cdots \times \mathbb{Z}_m[x]/\left\langle x - \zeta\omega^{2^k - 1} \right\rangle$$

(cf. Appendix E).

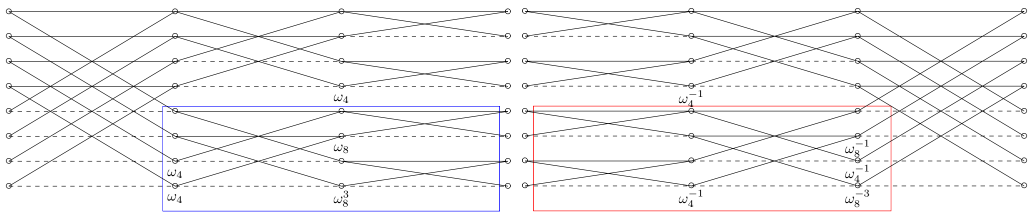
By setting $\zeta = 1$, we have the most commonly seen CT algorithm for $\mathtt{NTT}_{2^k:1:\omega}$ with $k2^{k-1} - 2^k + 1$ multiplications. And by setting $\zeta^{2^k} = -1$ and $\omega = \zeta^2$, we obtain the CT algorithm for NTTs used in Kyber [ABD+20b] and Dilithium [ABD+20a] with $k2^{k-1}$ multiplications.

If we invert all the computations with GS butterflies, then we have the GS algorithm for $\mathtt{NTT}^{-1}_{n:\zeta:\omega}$. If $\zeta^{-2^{k-1}} \neq \pm 1$, we can absorb 2^{k-1} multiplications by 2^{-k} at the end of $\mathtt{NTT}^{-1}_{n:\zeta:\omega}$as shown in Figure 1. This approach is widely used in optimized implementations on Cortex-M4. In particular, NewHope and NewHope-Compact by [ABCG20], Kyber by [ABCG20, GKS21], Dilithium by [GKS21], and Saber by [CHK+21]. But we can absorb more multiplications using the CT FFT algorithm for $\mathtt{NTT}^{-1}_{n:\zeta:\omega}$ as shown in the next section.

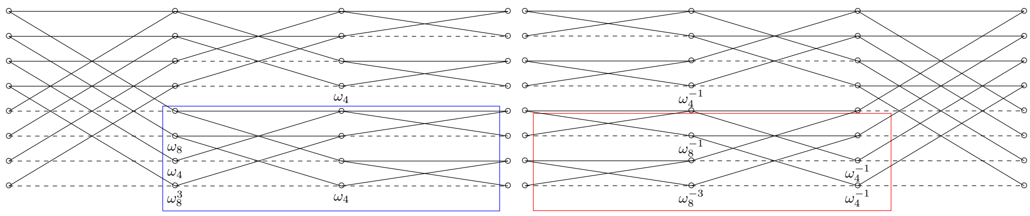
3.5.2 GS for NTT and CT for iNTT

Computing $\mathtt{NTT}_{2^k:\zeta:\omega}$ with GS butterflies is mapping $\mathbb{Z}_m[x]/\left\langle x^{2^i} - \zeta^{2^i} \right\rangle$ to $\mathbb{Z}_m[x]/\left\langle x^{2^i} - 1 \right\rangle$ whenever $i > 0$. After mapping $\mathbb{Z}_m[x]/\left\langle x^{2^k} - \zeta^{2^k} \right\rangle$ to $\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle$ and then to

$$\mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - 1 \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - \omega^{2^{k-1}} \right\rangle,$$



(a) CT for NTT over $x^8 - 1$ and over $x^4 + 1$. (b) CT for iNTT over $x^8 - 1$ and over $x^4 + 1$.



(c) GS for NTT over $x^8 - 1$ and over $x^4 + 1$. (d) GS for iNTT over $x^8 - 1$ and over $x^4 + 1$.

Figure 1: CT and GS butterflies over $x^8 - 1$ and $x^4 + 1$.

$\mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - \omega^{2^{k-1}} \right\rangle$ is mapped to $\mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - 1 \right\rangle$ immediately. It is clear to see that the result is also the bit-reversal of

$$\mathbb{Z}_m[x]/\langle x - \zeta \rangle \times \mathbb{Z}_m[x]/\langle x - \zeta\omega \rangle \times \cdots \times \mathbb{Z}_m[x]/\left\langle x - \zeta\omega^{2^k - 1} \right\rangle.$$

Now we can invert with CT butterflies to derive the CT algorithm for $\texttt{NTT}^{-1}_{n:\zeta:\omega}$. If $\zeta^{-1} \neq \pm 1$, then we can absorb $2^k - 1$ multiplications by 2^{-k}as shown in Figure 1. We implement the CT algorithm for $\texttt{NTT}^{-1}_{n:\zeta:\omega}$ on Cortex-M4.

3.6 NTT for NTT-unfriendly rings

For multiplying polynomials over finite integer rings not amiable for NTTs, since the coefficients of the result are bounded, we can choose an NTT-friendly modulus large enough to compute the result as in $\mathbb{Z}$, and then reduce to the target coefficient ring [FSS20, CHK+21].

For Saber, since we are multiplying a matrix by a vector with the polynomial modulus $x^{256} + 1$, the resulting (signed) coefficients are within $\pm\frac{\mu}{2} \cdot \frac{8192}{2} \cdot 256 \cdot l = \pm 12582912$. Therefore, if we choose a modulus $q' > 25165824 = 2 \cdot 12582912$ satisfying $2n|\mathbf{O}(q')$, we can compute the multiplication with length-n negacyclic NTTs in $\mathbb{Z}_{q'}$.

3.7 Incomplete NTT

Let $n = r_0 r_1$, $r_0|\mathbf{O}(m)$, and ω be a principal r_0-th root of unity. Incomplete NTT, written as $\texttt{NTT}_{r_0:1:\omega}$, refers to re-writing x^{r_1} as y followed by $\texttt{NTT}_{r_0:1:\omega}$ treating y as the indeterminate. Rewrite the degree-$(n-1)$ $\boldsymbol{a}(x)$ as degree-$(r_0 - 1)$ $\boldsymbol{a}'(y)$ where $a'_i = \sum_{j=0}^{r_1 - 1} a_{ir_1 + j} x^j$. Explicitly, $\texttt{NTT}_{r_0:1:\omega}$ maps $\boldsymbol{a}(x)$ to $(\boldsymbol{a}'(1), \boldsymbol{a}'(\omega), \ldots, \boldsymbol{a}'(\omega^{r_0 - 1}))$.

We can apply the incomplete NTT for multiplying polynomials. For $\boldsymbol{a}(x)\boldsymbol{b}(x)$ mod $(x^{r_0 r_1} - 1)$, we implement it as

$$\texttt{NTT}^{-1}_{r_0:1:\omega}\left(\texttt{base\_mul}_{r_0:r_1:\omega}\left(\texttt{NTT}_{r_0:1:\omega}(\boldsymbol{a}(x)), \texttt{NTT}_{r_0:1:\omega}(\boldsymbol{b}(x))\right)\right)$$

where $\texttt{base\_mul}_{r_0:r_1:\omega}$ means r_0 multiplications of degree-$(r_1 - 1)$ polynomials, each is over a suitable $x^{r_1} - \omega^i$.

Table 2: Summary of NTT approaches.

	M3		M4		
	Unmasked		Unmasked		Masked
	32-bit	16-bit	32-bit	16-bit	32-bit + 16-bit
Opt	speed	speed/stack	speed/stack	stack	speed/stack
Modulus	25171457	$3329, 7681$	3329×7681	$3329, 7681$	$44683393, 769$
`NTT`	8-layer-CT	6-layer-CT	6-layer-CT		
`base_mul`	1×1	4×4	4×4		
$\texttt{NTT}^{-1}$	8-layer-GS	6-layer-CT	6-layer-CT		

4 NTTs for `MatrixVectorMul`

In Table 2, we give a summary of the implemented NTTs. On Cortex-M4, we implement incomplete NTT/iNTT with 6 layers of CT butterflies for all implementations. On Cortex-M3, we implement both a 32-bit approach and a 16-bit approach to find the optimal one. For the 32-bit approach, we implement *complete* NTT with 8 layers of CT butterflies and *complete* iNTT with 8 layers of GS butterflies. For the 16-bit approach, we implement incomplete NTT/iNTT with 6 layers of CT butterflies.

This section is organized as follows: First, we analyze strategies for reducing stack usage of `MatrixVectorMul` in Section 4.1. Next, we go through our implementation on Cortex-M4 in Section 4.2: our stack-optimized implementation for unmasked Saber in Section 4.2.1, and speed-optimized and stack-optimized implementations for masked Saber in Section 4.2.2. Finally, we present our implementation on Cortex-M3 in Section 4.3, covering 32-bit NTT in Section 4.3.1, and 16-bit NTT in Section 4.3.2.

4.1 Reducing stack usage for `MatrixVectorMul`

The state-of-the-art Saber implementations [CHK+21] using NTTs have thus far not been thoroughly optimized for minimal stack consumption. The authors exclusively optimize for speed and do not report any stack usage. Later, Van Beirendonck and Hwang refactored the implementation to reduce stack usage without degrading speed.[3] In this section, we give a more thorough analysis of speed-memory trade-offs.

The most memory-consuming operation in Saber is the `MatrixVectorMul` $A^T s$ in key generation and As' in encryption. In all implementations, we employ on-the-fly generation of A, and consequently, only need one polynomial of A in memory. For computing $A^T s$ in key generation, we can compute the NTT for s on-the-fly but accumulate the entire result in the NTT domain with l accumulators. This is because the first component of the result only depends on the first column of A and the first component of s. For computing As' during encryption, we compute the entire NTT of s with l polynomial buffers but hold only one buffer for accumulation. This is because a component of the result is an inner product of a row of A and s', and is computed in order. In summary, for computing $A^T s$, the most memory-consuming part is the accumulation in the NTT domain. And for computing As', the most memory-consuming part is transforming s' into the NTT domain. In the most speed-optimized and the most stack-optimized implementations, there is no downside to this. But they result in different speed-memory trade-offs as shown below.

We now show that there are four ways for computing the product, which we will name strategies A, B, C, and D. They are distinguished by caching the NTTs of s or not and accumulating in the NTT domain or not.

A. We cache $\texttt{NTT}(s)$ and accumulate values in the NTT domain;
B. we cache $\texttt{NTT}(s)$ and accumulate values in normal domain;

[3] https://github.com/mupq/pqm4/pull/181

$$\begin{tikzcd}
\mathbb{Z}_{q'}[x]/\left\langle x^{256}+1\right\rangle \arrow[r, leftrightarrow, "\mathtt{NTT}_{64:\omega_{q':128}:\omega^2_{q':128}}"] \arrow[d, leftrightarrow, "\mathtt{CRT}"'] & \prod\limits_{i=0}^{63} \mathbb{Z}_{q'}[x]/\left\langle x^4-\omega^{2i+1}_{q':128}\right\rangle \arrow[d, leftrightarrow, "\mathtt{CRT}"'] \\
\prod\limits_{k=0,1}\left(\mathbb{Z}_{p_k}[x]/\left\langle x^{256}+1\right\rangle\right) \arrow[r, leftrightarrow, "\mathtt{NTT}_{64:\omega_{p_k:128}:\omega^2_{p_k:128}}"] & \prod\limits_{k=0,1}\left(\prod\limits_{i=0}^{63} \mathbb{Z}_{p_k}[x]/\left\langle x^4-\omega^{2i+1}_{p_k:128}\right\rangle\right)
\end{tikzcd}$$

Figure 2: Split of polynomial rings with `CRT` for incomplete NTT implementation for Saber. Blue arrows are isomorphisms via NTT. If $q' = p_0 p_1$, the red arrows are isomorphisms via `CRT` with $\omega_{q':128} = \mathtt{CRT}(\omega_{p_0:128}, \omega_{p_1:128})$.

C. we re-compute $\mathtt{NTT}(s)$ and accumulate values in the NTT domain;

D. we re-compute $\mathtt{NTT}(s)$ and accumulate values in normal domain.

All four strategies apply to $A^T s$ and As'. A is the fastest, and D consumes the least amount of memory. B and C run in comparable cycles but result in different degrees of trade-off for memory. For reducing the memory usage of $A^T s$, B is much better than C since B effectively reduces the size of accumulators. On the other hand, for reducing the memory usage of As', C is much better than B, since C avoids caching the entire $\mathtt{NTT}(s')$.

On Cortex-M4, A corresponds to the implementation in [CHK+21]; we additionally implement D for unmasked Saber, and A, C, and D for masked Saber. On Cortex-M3, we implement A for 32-bit NTT and strategies A, C, and D for 16-bit NTT.

4.2 Implementation on M4

For simplicity of discussion, throughout this section, we assume ω is a principal 128-th root of unity so $x^{256}+1 = x^{256} - \omega^{64}$. We illustrate our strategies only for `MatrixVectorMul` As' in encryption. However, the ideas apply analogously for $A^T s$ in key generation. For the concrete evaluation of the stack usage, we use l to refer to the matrix dimension ($l = 2$ for `LightSaber`, $l = 3$ for `Saber`, and $l = 4$ for `FireSaber`). For our masked implementation, we refer to `SABER_SHARES` as the number of shares. Our polynomial multiplication code works for any masking order. However, the other parts of masked Saber from [VBDK+20] only support first-order masking, and, hence, `SABER_SHARES` is always 2 in our experiments.

We exclusively use the Cooley–Tukey FFT algorithm to implement both the NTT and iNTT on the Cortex-M4. We recall the corresponding butterfly operations for 16-bit NTTs and 32-bit NTTs known from the literature [ABCG20, GKS21, ACC+21] in the following.

32-bit CT butterflies. A straightforward implementation of 32-bit CT butterflies is using `smull` and `smlal` both giving 64-bit immediate results for $a \cdot (b\mathtt{R} \bmod {}^{\pm}\mathtt{Q})$ and multiplication by `Q` with accumulation. A 32-bit CT butterfly is to proceed with add-sub of (a_0, ba_1) [ACC+21, GKS21]as shown in Algorithm 6.

16-bit CT butterflies. We implement CT butterflies with `smul{b,t}{b,t}` and `smla{b,t}{b,t}` giving multiplications and multiplications with accumulations of specified halves. Furthermore, we can use `sadd16` and `ssub16` to do add-sub pairs in parallelas shown in Algorithm 7 [ABCG20].

4.2.1 New record on stack usage for unmasked Saber

For Saber, all polynomial multiplications are of the form of big $\times$ small, i.e., we compute $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ where $\boldsymbol{a}(x) \in \mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ and $\boldsymbol{b}(x) \in \mathbb{Z}_\mu[x]/\langle x^{256}+1\rangle$.

Previous work [CHK+21] has shown that this can be efficiently computed using NTTs by switching to an NTT-friendly prime $q' \geq 2 \cdot \frac{q}{2} \cdot \frac{\mu}{2} \cdot l$ which suffices for acquiring the result in $\mathbb{Z}$. The authors chose the prime 25165824. However, we instead use $q' = 7681 \cdot 3329 = 25570049 > 25165824$ which also allows the use of NTTs, but is advantageous in terms of stack usage.

NTT with composite modulus. Let p_0 and p_1 be distinct primes with 128 dividing both $\mathbf{0}(p_0)$ and $\mathbf{0}(p_1)$, $\omega_{p_0:128}$ and $\omega_{p_1:128}$ be principal 128-th roots of unity in $\mathbb{Z}_{p_0}$ and $\mathbb{Z}_{p_1}$, respectively. By CRT and incomplete NTTs, we have the following isomorphisms:

- $\mathbb{Z}_{p_0 p_1} \cong \mathbb{Z}_{p_0} \times \mathbb{Z}_{p_1}$
- $\mathtt{NTT}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}$ giving $\mathbb{Z}_{p_0}[x]/\langle x^{256}+1\rangle \cong \prod_{i=0}^{63} \mathbb{Z}_{p_0}[x]/\langle x^4 - \omega^{2i+1}_{p_0:128}\rangle$
- $\mathtt{NTT}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}$ giving $\mathbb{Z}_{p_1}[x]/\langle x^{256}+1\rangle \cong \prod_{i=0}^{63} \mathbb{Z}_{p_1}[x]/\langle x^4 - \omega^{2i+1}_{p_1:128}\rangle$

Together, we have $\mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}$ giving

$$\mathbb{Z}_{p_0p_1}[x]/\langle x^{256}+1\rangle \cong \prod_{i=0}^{63} \mathbb{Z}_{p_0p_1}[x]/\langle x^4 - \omega^{2i+1}_{p_0p_1:128}\rangle,$$

Figure 2 is an illustration of the isomorphisms.

Instead of implementing $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathbb{Z}_{p_0p_1}[x]/\langle x^{256}+1\rangle$ as applying $\mathtt{NTT}^{-1}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}$ on the $\mathtt{base\_mul}_{64:4:\omega_{p_0p_1:128}}$ of

$$\left(\mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x)), \mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}(\boldsymbol{b}(x))\right),$$

for saving memory, we apply $\mathtt{NTT}^{-1}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}$ on the CRT of

$$\begin{array}{l} \mathtt{base\_mul}_{64:4:\omega_{p_0:128}}\left(\mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x)) \bmod p_0, \mathtt{NTT}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}(\boldsymbol{b}(x))\right) \\ \mathtt{base\_mul}_{64:4:\omega_{p_1:128}}\left(\mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x)) \bmod p_1, \mathtt{NTT}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}(\boldsymbol{b}(x))\right) \end{array}.$$

The workflow goes as outlined in Algorithm ?? in Appendix C. We declare three 16-bit arrays in the order of buff1_16, buff2_16, buff3_16 and 32-bit pointers *buff1_32 = (uint32_t*)buff1_16, *buff2_32 = (uint32_t*)buff2_16 so we can access the memory as 32-bit arrays at some point. First, we compute $\mathtt{NTT}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}(\boldsymbol{a}(x))$ and store the result to the 32-bit array buff1_32. We then compute and put buff1_32 mod p_1 in the 16-bit array buff3_16. For computing buff1_32 mod p_0, we see that the result in buff1_32 won't be needed after reducing modp_0, so we compute and put buff1_32 mod p_0 in the 16-bit array buff1_16. This is doable if we compute modp_0 from the beginning. We proceed with computing $\mathtt{NTT}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}(\boldsymbol{b}(x))$ in the 16-bit array buff2_16 followed by $\mathtt{base\_mul}_{64:4:\omega_{p_1:128}}$ outputting to buff3_16, and computing $\mathtt{NTT}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}(\boldsymbol{b}(x))$ in the 16-bit array buff2_16 followed by $\mathtt{base\_mul}_{64:4:\omega_{p_0:128}}$ outputting to buff2_16. Next we compute the explicit CRT, *giving 32-bit coefficients as in the NTT domain with coefficient ring* $\mathbb{Z}_{p_0p_1}$, and put the result in the 32-bit array buff1_32. Finally, we compute $\mathtt{NTT}^{-1}_{64:\omega_{p_0p_1:128}:\omega^2_{p_0p_1:128}}$ and reduce the coefficient ring to $\mathbb{Z}_q$.

Memory layout. For implementing stack optimized MatrixVectorMul in encapsulation of unmasked Saber, we employ a variant of Strategy D: we declare arrays

```
uint16_t buff1_16[256], buff2_16[256], buff3_16[256], acc_16[256]
```

multiply an element of A by an element of s' with the above strategy, accumulate the result to acc_16, and finally derive an element of b'. In total, only 1536 bytes are needed if the accumulator is excluded.

Comparison with previous stack optimized implementation. We compare the memory usage of polynomial multiplication to the currently most stack optimized implementation – 4 levels of memory efficient Karatsuba [MKV20]. Ignoring the extra $O(\log n)$ memory overhead for Karatsuba, we focus on the polynomial buffers for the multiplicands and the result. For the Karatsuba approach, one needs 512 bytes for the accumulator, 512 bytes for holding a component of A, and 1022 bytes for the degree-510 result – almost the same as the NTT approach with composite modulus. Essentially, *any algorithm not exploiting the negacyclic property requires such amount of memory.* We only find the work by [PC20] giving a non-NTT-based approach exploiting the negacyclic property, but the authors reported that they were not able to achieve a smaller footprint than the Karatsuba by [MKV20].

4.2.2 `MatrixVectorMul` for masked Saber

A masked implementation of Saber decapsulation using Toom–Cook multiplication is given in [VBDK+20]. We improve this implementation by replacing `MatrixVectorMul` and `InnerProd` with NTT-based multiplications. As secret polynomials s and s' are masked arithmetically modulo q, the multiplications are no longer big $\times$ small, but rather big $\times$ big, i.e., all input polynomials are in $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$. Therefore, the coefficients of the product can be larger than 32-bit. This implies switching to an NTT-friendly 25-bit modulus and performing 32-bit NTTs no longer produces correct results.

Instead, we propose combining a 32-bit NTT with a 16-bit NTT to compute the 48-bit value and then reduce each coefficient to $\mathbb{Z}_q$. We compute 32-bit NTT and 16-bit NTT by choosing $p_0 = 44683393 = 349089 \cdot 128 + 1$ and $p_1 = 769 = 6 \cdot 128 + 1$ as moduli. Their product $q' = p_0 p_1 = 44683393 \cdot 769 = 34361529217 > 34359738368 = 2 \cdot \left(\frac{q}{2}\right)^2 \cdot 256 \cdot 4$ shows that after applying `CRT`, we derive the result as in $\mathbb{Z}$.

For computing $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$, we compute $\boldsymbol{a}(x)\boldsymbol{b}(x)$ in $\mathbb{Z}_{p_0}[x]/\langle x^{256}+1\rangle$ with 32-bit NTT and in $\mathbb{Z}_{p_1}[x]/\langle x^{256}+1\rangle$ with 16-bit NTT. Then, we apply `CRT` to obtain the result in $\mathbb{Z}_{q'}[x]/\langle x^{256}+1\rangle$ which coincides with the result in $\mathbb{Z}[x]/\langle x^{256}+1\rangle$. Finally, we reduce the coefficient ring to $\mathbb{Z}_q$.

First, we show how to multiply two 16-bit polynomials, $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ within 3072 bytes. The idea is simple: we compute the 32-bit $\texttt{NTT}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}$ of $\boldsymbol{a}(x)$, store the result in a 32-bit array, compute the 16-bit $\texttt{NTT}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}$ of $\boldsymbol{a}(x)$, and store the result in a 16-bit array. For $\boldsymbol{b}(x)$, we declare a 32-bit array and a 16-bit array, and compute 32-bit $\texttt{NTT}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}$ and 16-bit $\texttt{NTT}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}$ as for $\boldsymbol{a}(x)$. Then, we perform in-place 32-bit $\texttt{base\_mul}_{64:4:\omega_{p_0:128}}$ followed by in-place 32-bit $\texttt{NTT}^{-1}_{64:\omega_{p_0:128}:\omega^2_{p_0:128}}$, and in-place 16-bit $\texttt{base\_mul}_{64:4:\omega_{p_1:128}}$ followed by in-place 16-bit $\texttt{NTT}^{-1}_{64:\omega_{p_1:128}:\omega^2_{p_1:128}}$. Finally, we apply `CRT` followed by reduction to $\mathbb{Z}_q$.Algorithm 4 is an illustration of the idea.

Memory layout for speed-optimized implementations. For implementing speed-optimized `MatrixVectorMul` in encapsulation of masked Saber, we employ a shared variant of Strategy A, and declare arrays

$$\begin{cases} \texttt{uint32\_t s\_NTT\_32[SABER\_SHARES][}l\texttt{][256]} \\ \texttt{uint16\_t s\_NTT\_16[SABER\_SHARES][}l\texttt{][256]} \\ \texttt{uint32\_t buff\_32[256], acc\_32[SABER\_SHARES][256]} \\ \texttt{uint16\_t buff\_16[256], acc\_16[SABER\_SHARES][256]} \end{cases}.$$

For each share of s', we compute the 32-bit NTTs and 16-bit NTTs of it and store them in `s_NTT_{32, 16}`. For computing an element of shared b', we repeat the following l times: compute the 32-bit NTT and 16-bit NTT of an element of A; multiply them by the corresponding element of each share of s' using $\texttt{base\_mul}_{64:4:\omega_{p_0:128}}$ and $\texttt{base\_mul}_{64:4:\omega_{p_1:128}}$;

accumulate the results to accumulators `acc_{32, 16}`; compute the 32-bit iNTT and 16-bit iNTT for each share; and finally, solve `CRT` and reduce to $\mathbb{Z}_q$ for each share.

Memory layout for stack-optimized implementations. For implementing stack optimized `MatrixVectorMul` in decapsulation of masked Saber, we employ a shared variant of Strategy D, and declare arrays

$$\begin{cases} \texttt{uint32\_t s\_NTT\_32[256], buff\_32[256]} \\ \texttt{uint16\_t s\_NTT\_16[256], buff\_16[256], acc\_16[SABER\_SHARES][256]} \end{cases}.$$

We repeat l times computing the shares of an element of b'. For computing the shares of a polynomial product, we repeat l times for the following. We first expand an element of A and store it in `buff_16`. Then we compute the 32-bit NTT and in-place 16-bit NTT for the element and the result is stored in `buff_{32, 16}`. Next, we repeat `SABER_SHARES` times clearing the arrays `s_NTT_{32, 16}`, computing 32-bit NTT and 16-bit NTT of a share of s' and storing them in `s_NTT_{32, 16}`, computing in-place $\texttt{base\_mul}_{64:4:\omega_{p_0}:128}$ and $\texttt{base\_mul}_{64:4:\omega_{p_1}:128}$, in-place 32-bit iNTT and 16-bit iNTT, solving with `CRT`, and finally, accumulating the result to the corresponding share of `acc_16`. In total, 3072 bytes are needed if accumulators are excluded.

Comparison with masked Saber with Toom–Cook. We first compare the stack usage. From [VBDK+20], the polynomial multiplication is implemented as a Toom-4 followed by 2 levels of Karatsuba. Therefore, the memory usage for entire evaluation of one polynomial is $2 \cdot 256 \cdot \frac{7}{4} \cdot \left(\frac{3}{2}\right)^2 = 1568$ bytes. With carefully optimized accumulation, 3076 bytes are used. In total, 3588 bytes are needed because of the additional buffer of an element of A. For our stack optimized implementation, we only need 3072 bytes. Next we compare the number of NTTs computed in the speed optimized implementation. We compute 9 32-bit NTTs and 9 16-bit NTTs for A, 6 32-bit NTTs and 6 16-bit NTTs for the shared secret, 6 32-bit iNTTs and 6 16-bit iNTTs for the shared results. In summary, we need 15 32-bit NTTs, 15 16-bit NTTs, 6 32-bit iNTTs, and 6 16-bit iNTTs. Given that one 16-bit NTT takes 0.79× of one 32-bit NTT and one 16-bit iNTT takes 0.82× of one 32-bit iNTT, then essentially we need the equivalent of 26.85 32-bit NTTs and 10.92 32-bit iNTTs. Compared to [CHK+21], we only need about 2.24× 32-bit NTTs and 3.64× 32-bit iNTTs, which is obviously faster than the shared variant of Toom–Cook.

4.3 Implementation on M3

Due to the more limited instruction set and the early terminating long multiplications on the Cortex-M3, the 32-bit butterflies from the previous section can only be used with some restrictions. In general, there are two approaches to still benefit from NTTs on the Cortex-M3: One can either implement 32-bit NTTs, but avoid the early terminating multiplication instructions for secret inputs, or one exclusively uses 16-bit NTTs and computes the `CRT` of the results. The former approach resembles the Cortex-M4 approach from [CHK+21] and the previous section, while the latter is similar to the AVX2 implementation from [CHK+21]. We implement both approaches and compare their performance.

We start by describing the butterfly implementations. For the 32-bit approach, we use CT for the NTT and GS for the iNTT, while for the 16-bit approach we use CT for both.

32-bit CT butterflies. The 32-bit CT butterflies with `smull` and `smlal` are functionally correct on Cortex-M3. However, these instructions are early-terminating and can only be used when computing on public data. We denote the 5-instruction 32-bit butterflies as `NTT_leak` on Cortex-M3. For computing the NTT of the secret values s and s' on Cortex-M3, we implement `smull_const` and `smlal_const` with radix-2^{16} schoolbook multiplication as suggested in [GKS21]. The CT butterfly using `smull_const` and `smlal_const` is illustrated in Algorithm 8. The result of the multiplication by `Qprime` needs to be split into upper and lower 16 bits before we can use them in the subsequent multiplication by `Q`.

32-bit GS butterflies. As implemented for CT butterflies, we also use `smull_const` and `smlal_const` for implementing 32-bit GS butterfliesas shown in Algorithm 12. Both coefficients are initially loaded as 32-bit values because the add-sub is computed before the multiplication. We then split the result of $a_0 - a_1$ into halves for Montgomery multiplication.

16-bit CT butterflies. A straightforward implementation of 16-bit CT butterflies is using `mul` and `mla` with `sxth` for extracting the lower 16 bits [GKS21]as shown in Algorithm 9.

4.3.1 32-bit NTT for `MatrixVectorMul`

We implement strategy A for `MatrixVectorMul` using 32-bit NTTs on Cortex-M3. An important observation is that A is public, so we can employ `NTT_leak` on A. This greatly improves the performance since among the $l^2 + 2l$ NTTs/iNTTs, l^2 of them are computation for A. On the other hand, the NTTs of secret and `base_mul` can only be computed with `smull_const` and `smlal_const`. We use the constant-time 32-bit CT and GS butterflies for the NTT and iNTT on secret data, respectively. Using `smull_const` and `smlal_const` leads to a much higher register pressure during the entire multiplication. Due to that, we do not benefit from using incomplete NTTs as the 2×2 base multiplication already exhausts the available registers. Therefore, we compute complete NTTs.

4.3.2 16-bit NTTs for `MatrixVectorMul`

We implement strategies A, C, and D with the 16-bit NTT approach for `MatrixVectorMul` on Cortex-M3. Our results show that the 16-bit approach is faster than the 32-bit approach. For strategy A, this corresponds to the AVX2 implementation from [CHK+21]. We also carry out the stack optimization on Cortex-M4 and implement strategies C and D.

4.3.3 A Note on combining 32-bit and 16-bit

There is an interesting observation when comparing the cycles of `MatrixVectorMul` implemented using 32-bit and 16-bit NTTs, which suggests that the 16-bit approach is better. However, one 8-layer `NTT_leak` is only about 1.15× of two 6-layer 16-bit NTTs, giving a hint that 6-layer `NTT_leak` might be a faster approach. We experimented with combining `NTT_leak` and constant-time 16-bit NTTs for strategy A. One may first process A with 6-layer `NTT_leak` followed by transforming the result into two 16-bit NTTs by the map $i \mapsto (i \bmod p_0, i \bmod p_1)$. However, our experiments show that the performance gain with `NTT_leak` for A is canceled out by computing the map $i \mapsto (i \bmod p_0, i \bmod p_1)$. Therefore, we did not use this trick in our implementation.

5 Result

This section presents our results on the Cortex-M3 and Cortex-M4. We first describe our target platforms and setup and then present the results in Section 5.1. Section 5.2 evaluates the side-channel resistance of our masked implementation.

Cortex-M4 setup. We target the `STM32F407-DISOVERY` board featuring a `STM32F407VG` Cortex-M4 microcontroller with 196 kB of SRAM and 1 MB of flash. Our benchmarking setup is based on pqm4 [KRSS]; we clock the core at 24 MHz with no flash wait states.

Table 3: Cycle counts for `NTT`, `base_mul`, $\mathtt{NTT}^{-1}$ on the Cortex-M3 and the Cortex-M4. For each of the first three columns, the cycles for a polynomial multiplication will be $2 \cdot \mathtt{NTT}(\text{or } \mathtt{NTT} + \mathtt{NTT\_leak}) + \mathtt{NTT}^{-1} + \mathtt{base\_mul} + \mathtt{CRT}(\text{if not } -)$. The `NTT` of the column 32-bit + 16-bit contains a layer of `sbfx` to reduce elements to $\mathbb{Z}_q$. For the last two columns, they together implement a polynomial multiplication, and the cycles is the sum of the two columns. One of the 16-bit `base_mul` is preceded with modular reduction to save load and store instructions. For the stack usage, the first three columns are for a polynomial multiplication. The stack usage of the last two columns are the bytes occupied by the functions. But the actual stack usage is 1 536 bytes, since the arrays are overlapped.

	M3		M4		
	2 × 16-bit	32-bit	32-bit + 16-bit	32-bit	16-bit + 16-bit
`NTT`	16 774	31 056	6 116 + 4 852	5 853	4 374+ 4 822
`NTT_leak`	–	19 363	–	–	–
$\mathtt{NTT}^{-1}$	19 079	37 394	5 872 + 4 817	7 137	–
`base_mul`	11 933	8 532	4 186 + 2 966	–	3731 + 2 965
modp_i	–	–	–	–	0 + 1 171
`CRT`	4 642	–	4 503	–	2 435
`poly_mul`	69 202	96 345	44 280	32 488	
Bytes(speed opt)	2 048	2 048	3 072	–	–
Bytes(stack opt)	1 536	–	2 048	1 536	1 024

Cortex-M3 setup. Our Cortex-M3 target platform is the `Nucleo-F207ZG` board containing a `STM32F207ZG` core with 128 kB of SRAM and 1 MB of flash. Our benchmarking setup is based on pqm3.[4] We clock the core at 30 MHz to avoid having flash wait states.

Keccak and Randomness. For both implementations, we use the `ARMv7-M` assembly implementation of Keccak from the XKCP[5] which is operational on the Cortex-M3 and the Cortex-M4. This implementation is also contained in both pqm3 and pqm4. For randomness required in key generation and encapsulation, we use the hardware RNG.

All code is compiled with `arm-none-eabi-gcc` Version 10.2.0 with `-O3`.

5.1 Performance

We report results for a single polynomial multiplication in Table 3. Each of the first three columns is realizing a polynomial multiplication as computing `NTT` on both polynomials, `base_mul` for the `NTT`s, and finally $\mathtt{NTT}^{-1}$(and followed by `CRT` if needed). For the last two columns, they together realize a polynomial multiplication as computing one 32-bit `NTT` and two 16-bit `NTT`s, two 16-bit `base_mul`s, `CRT` giving a *32-bit* polynomial, and finally a 32-bit $\mathtt{NTT}^{-1}$.

We report results of our implementations of unmasked Saber as shown in Table 5. Detailed numbers for MatrixVectorMul and InnerProd are given in Appendix A.

For the ARM Cortex-M3, our speed-optimized NTT implementation of (unmasked) Saber requires only 65.0%-70.7% of the time and 45.0%-51.2% of stack space compares to the Toom–Cook implementation available in pqm3. Our stack-optimized implementation is still 5.6%-13.0% faster while requiring 70.3%-79.9% less stack space.

For the Cortex-M4, we outperform the NTT implementation by Chung et al. [CHK+21] by 2.2%-6.9% while needing considerably less stack. Compared to previous speed-optimized Toom implementations [MKV20] we require significantly less stack space (65.8%-67.7% less)

[4] https://github.com/mupq/pqm3
[5] https://github.com/XKCP/XKCP

while only requiring 68.9%-75.5% of the time. For heavily stack-optimized implementations, we require about the same or slightly less amount of stack while achieving a vast speed-up.

Masked Saber results are shown in Table 6. Our speed-optimized approach is outperforming Toom–Cook by 15.4%. Our stack-optimized approach is using 72.3% of the stack of Toom–Cook, and is only a little slower than Toom–Cook. In trading speed for memory, we implement strategy C, outperforming Toom–Cook in both speed and memory.

Notes on joint implementation with Kyber NTT optimized with stack and program size. Due to the flexibility of choosing moduli, one can share the 16-bit NTT implementations between Kyber and Saber. But we do not recommend this. For joint implementation in software, neither Kyber nor Saber will be optimal for the following reasons: (1) The Kyber NTT is 7 layers, while the optimal NTT for Saber is 6 layers; (2) Saber requires two 16-bit primes where their product must be larger than 25165824. The smallest suitable prime are 3329 and 7681. The first reason implies `MatrixVectorMul` for Saber is suboptimal, and the second reason implies more reductions are required for NTT of Kyber since $7681 > 3329$.

5.2 Leakage Evaluation of Masked Saber

We adopt the test vector leakage assessment (TVLA) methodology to perform leakage detection. We made use of CW1173 ChipWhisperer-Lite [Newb] to collect the power consumption traces at a sampling rate of 59.04 MS/s. The target board is CW308 UFO [Newc] with ChipWhisperer platform - CW308_STM32F4 (ST Micro STM32F405) [Newa] on which we run our implementations at the frequency of 7.38 MHz. We focus on the key decapsulation and capture three sets of power traces corresponding to the test vectors in Table 4 [ISO16]. Then, compute Welch's t-test to identify the differentiating features between Set 1 and Set 2, and between Set 1 and Set 3.

Table 4: Test Vectors of `Saber` for captured power traces

Set Number	Test vector properties
Set 1	Fixed secret key, Fixed ciphertext
Set 2	Fixed secret key, Randomly-chosen ciphertexts
Set 3	Randomly-chosen secret keys, Fixed ciphertext

The maximum number of samples on the CW1173 ChipWhisperer-Lite is 24573 [Newb]. Thus, we cannot capture the whole power trace of a full `Saber` decapsulation. In our experiment we only capture traces of the power consumption toward the beginning of the key decapsulation, which is an inner product of polynomial multiplications between ciphertext and the secret key, which is implemented using the NTT. There are four steps: NTT of the ciphertext, NTT of the secret key, base multiplication, and the iNTT.

In the first experiment, we do the TVLA on the power traces of Set 1 and Set 2, which is corresponding to the randomly-chosen ciphertexts and fixed-chosen ciphertexts with a fixed secret key. In the second step, doing the NTT of the secret key, there is no leakage, which is expected since the secret key is fixed in our first experiment. The first and the third steps, doing the NTT of ciphertext and base multiplications between the NTT results of ciphertext and the secret key, show leakage, which is expected since the ciphertext is public information. After the base multiplication, finally, the inverse NTT shows no leakage in the protected version. By contrast, there is leakage in the unprotected version. Figure 3a and Figure 3 show the t-tests of unprotected `Saber` and masked `Saber` on power traces of Set 1 and Set 2. Each figure can be separated into two parts by the black lines: 1. doing base multiplication between the NTT of ciphertext and the NTT of the secret key; 2. doing the inverse NTT. We can see that the t-statistic value of the masked `Saber` is inside the ± 4.5 [WO19] interval (red line) for all the points in time during the NTT^{-1}, which implies that the protected implementation is secure against first-order attacks.

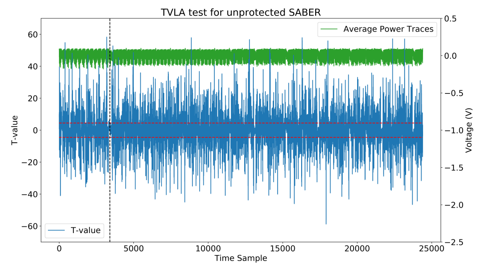
In the second experiment, we do the TVLA on the power traces of Set 1 and Set 3, which is corresponding to the randomly-chosen secret keys and fixed-chosen secret keys with a fixed ciphertext. In the second step, doing the NTT of the secret key shows no leakage in the protected version. By contrast, there is leakage in the unprotected version. Figure 4a and Figure 4b show the t-tests of unprotected `Saber` and masked `Saber` on power traces of Set 1 and Set 3. Each figure can be separated into two parts by the black lines: 1. doing the NTT of ciphertext; 2. doing the NTT of the secret key. We can see that the t-statistic value of the masked `Saber` is inside the ± 4.5 [WO19] interval, the red lines in the figures, for all the points in time during the NTT, which implies that the protected implementation is secure against first-order attacks.

Our masked `Saber` implementation as described in Section 4.2.2 only differs from [VBDK+20] in `MatrixVectorMul` and `InnerProd`. Hence, the masked Keccak implementation remains unchanged. To verify that this implementation is indeed secure, we perform another set of exeperiments targeting the beginning of the SHA3-512 function, which is the absorb step in the Keccak sponge construction. Then, we do the TVLA on the power traces of Set 1 and Set 2, which is corresponding to the randomly-chosen ciphertexts and fixed-chosen ciphertexts with a fixed secret key. In masked `Saber`, turning the masks on or off can activate or deactivate the countermeasure. Figure 5a and Figure 5b show the t-tests of Keccak implementation in masked `Saber` on power traces of Set 1 and Set 2 with masks off and with masks on, respectively. We can see that the t-statistic value of the masked `Saber` with masks on is inside the ± 4.5 [WO19] interval (the red lines in the figures) for all the points. It means that the masked `Saber` implementation is secure against first-order attacks when the masks are on.

Acknowledgements

The authors thank Michiel Van Beirendonck and Jan-Pieter D'Anvers for sharing their source code of masked Saber.

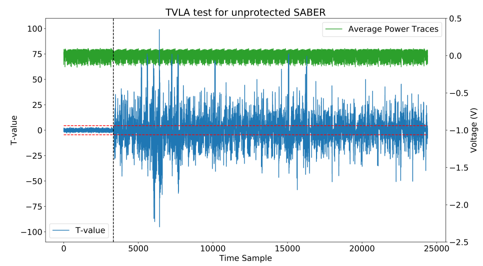
We also thank Ministry of Science and Technology for grants 109-2221-E-001-009-MY3 and 109-2923-E-001-001-MY3, the Sinica Investigator Award AS-IA-109-M01, Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).



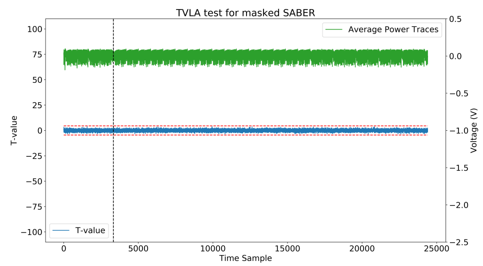
(a) T-test of unprotected `Saber` on power traces of Set 1 and Set 2

(b) T-test of masked `Saber` on power traces of Set 1 and Set 2

Figure 3: T-test results on traces of Set 1 and 2

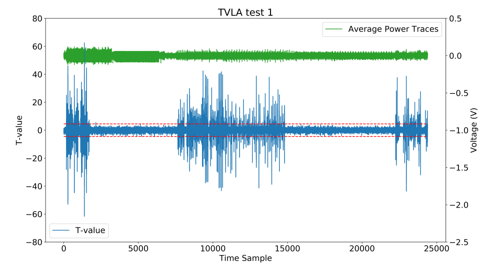


(a) T-test of unprotected `Saber` on power traces of Set 1 and Set 3

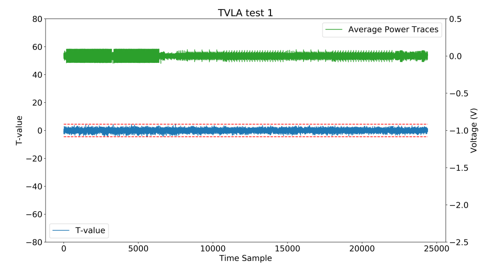


(b) T-test of masked `Saber` on power traces of Set 1 and Set 3

Figure 4: T-test of Keccak implementation in masked `Saber` with masks off



(a) T-test of Keccak implementation in masked `Saber` with masks off



(b) T-test of Keccak implementation in masked `Saber` with masks on

Figure 5: T-test of Keccak implementation in masked `Saber` on traces Set 1 and 2

Table 5: Speed and stack results for unprotected Saber on Cortex-M3 and Cortex-M4. Key generation, encapsulation, and decapsulation are denoted as **K**, **E**, and **D**, respectively.

			LightSaber		Saber		FireSaber	
			cc	stack	cc	stack	cc	stack
M3	pqm3 Toom (speed)	**K**	710k	9 652	1 328k	13 252	2 171k	20 116
		E	967k	11 372	1 738k	15 516	2 688k	22 964
		D	1 081k	12 116	1 902k	16 612	2 933k	24 444
	This work 16-bit (speed, A)	**K**	**540k**	5 756	**939k**	6 788	**1 439k**	7 812
		E	**715k**	6 436	**1 194k**	7 468	**1 751k**	8 492
		D	**749k**	6 436	**1 237k**	7 468	**1 811k**	8 492
	This work 16-bit (stack, D)	**K**	632k	**3 420**	1 253k	**3 932**	1 955k	**4 444**
		E	887k	**3 204**	1 614k	**3 332**	2 427k	**3 460**
		D	923k	**3 204**	1 657k	**3 332**	2 487k	**3 460**
	This work 32-bit (speed, A)	**K**	594k	5 732	1 057k	6 756	1 553k	7 788
		E	800k	6 412	1 330k	7 444	1 883k	8 468
		D	877k	6 420	1 429k	7 452	2 016k	8 476
M4	[MKV20] (stack)	**K**	612k	3 564	1 230k	4 348	2 046k	5 116
		E	880k	**3 148**	1 616k	3 412	2 538k	3 668
		D	976k	**3 164**	1 759k	3 420	2 740k	3 684
	[CHK+21] (speed)	**K**	360k	14 604	658k	23 284	1 008k	37 116
		E	513k	16 252	864k	32 620	1 255k	40 484
		D	498k	16 996	835k	33 824	1 227k	41 964
	This work 32-bit (speed, A)	**K**	**353k**	5 764	**644k**	6 788	**990k**	7 812
		E	**487k**	6 444	**826k**	7 468	**1 208k**	8 484
		D	**456k**	6 440	**777k**	7 484	**1 152k**	8 500
	This work hybrid (stack, D)	**K**	423k	**3 428**	819k	**3 940**	1 315k	**4 452**
		E	597k	3 204	1 063k	**3 332**	1 617k	**3 468**
		D	583k	3 220	1 039k	**3 348**	1 594k	**3 484**

Table 6: Results for Masked Saber on the Cortex-M4.

	Saber Decapsulation	
	cc	stack
[VBDK+20]	2 833k	11 656
32-bit + 16-bit (speed, A)	**2 385k**	16 140
32-bit + 16-bit (C)	2 615k	10 476
32-bit + 16-bit (stack, D)	2 846k	**8 432**

Table 7: Masking overhead of cycle counts and stack usage.

	unmasked A		unmasked D	
	cc	stack	cc	stack
masked A	3.07	2.16	2.30	4.82
masked C	3.37	1.40	2.52	3.13
masked D	3.66	1.13	2.74	2.52

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.

[AB74] RC Agarwal and C Burrus. Fast convolution using fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

[ABC+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://classic.mceliece.org/.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020. https://doi.org/10.13154/tches.v2020.i3.336-357.

[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.

[ARM10] ARM Limited. *Cortex-M3 Devices Generic User Guide*, December 2010. https://github.com/ARM-software/CMSIS/blob/master/CMSIS/Pack/Example/Documents/dui0552a_cortex_m3_dgug.pdf.

[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/.

[Ber] Daniel J. Bernstein. Multidigit multiplication for mathematicians. http://cr.yp.to/papers.html#m3.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2019*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019. https://doi.org/10.1007/978-3-030-23696-0_11.

[Bou89] Nicolas Bourbaki. *Algebra I.* Springer, 1989.

[Car14] Robert Daniel Carmichael. *The theory of numbers.* Number 13. J. Wiley & Sons, Incorporated, 1914.

[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/.

[CF94] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of computation*, 62(205):305–324, 1994.

[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/.

[DV78] Eric Dubois and Anastasios N. Venetsanopoulos. The discrete fourier transform over finite rings with application to fast convolution. *IEEE Computer Architecture Letters*, 27(07):586–593, 1978.

[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/.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):239–280, 2020. https://doi.org/10.13154/tches.v2020.i4.239-280.

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

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, 2021. https://doi.org/10.46586/tches.v2021.i1.1-24.

[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.

[HP21] Daniel Heinz and Thomas Pöppelmann. Combined fault and dpa protection for lattice-based cryptography. 2021. https://eprint.iacr.org/2021/101.

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

[ISO16] ISO. ISO/IEC 17825: Information technology – Security techniques – Security requirements for cryptographic modules, 2016.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbf{Z}_{2^m}[x]$ on cortex-m4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security*, pages 281–301, 2019. https://eprint.iacr.org/2018/1018.

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

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom–Cook multiplication: an application to module-lattice based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):222–244, 2020. https://doi.org/10.13154/tches.v2020.i2.222-244.

[Mon85] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X.

[MS90] Peter Montgomery and Robert D. Silverman. An FFT extension to the $p-1$ factoring algorithm. *Mathematics of Computation*, 54(190):839–854, April 1990.

[Newa] Technology Inc NewAE. Chipwhisperer platforms. https://rtfm.newae.com/Targets/Target%20Defines/.

[Newb] Technology Inc NewAE. Cw1173 chipwhisperer-lite. https://rtfm.newae.com/Capture/ChipWhisperer-Lite/.

[Newc] Technology Inc NewAE. Cw308 ufo. https://rtfm.newae.com/Targets/CW308%20UFO/.

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

[PC20] Irem Keskinkurt Paksoy and Murat Cenk. Tmvp-based multiplication for polynomial quotient rings and application to saber on arm cortex-m4. *IACR Cryptol. ePrint Arch.*, 2020:1302, 2020.

[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.

[VBDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. *IACR Cryptol. ePrint Arch.*, 2020:733, 2020. https://eprint.iacr.org/2020/733.

[WO19] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). volume 11923 of *LNCS*, pages 256–284. Springer, 2019.

A Cycle count of `MatrixVectorMul` and `InnerProd`

We give numbers for `MatrixVectorMul` as shown in Table 8. `MatrixVectorMul` is transforming all the components into NTT domain, computing `base_mul`, and then applying $\mathtt{NTT}^{-1}$(following a `CRT` or followed by a `CRT` if needed).

We give numbers for `InnerProd` as shown in Table 9. The `InnerProd` in decryption is transforming all the components into NTT domain, computing `base_mul`, and then $\mathtt{NTT}^{-1}$(following a `CRT` or followed by a `CRT` if needed). For the `InnerProd` in encryption, we can re-use $\mathtt{NTT}(s)$ from the `MatrixVectorMul` and can, consequently, save some operations in case there is sufficient stack space available for caching.

B Isomorphism

We show here how `CRT` preserves the split of NTTs. Let q_0 and q_1 be coprime integers with $n|\mathbf{0}(q_0 q_1)$, $\omega_{q_0:n}$ and $\omega_{q_1:n}$ be principal n-th roots of unity in $\mathbb{Z}_{q_0}$ and $\mathbb{Z}_{q_1}$, and $\omega_{q_0 q_1:n} = \mathtt{CRT}(\omega_{q_0:n}, \omega_{q_1:n})$. Then we have the following:

$$
\begin{aligned}
& \mathbb{Z}_{q_0 q_1}[x]/\langle x^n - 1\rangle \\
\cong\ & (\mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1})[x]/\langle x^n - 1\rangle \\
\cong\ & \mathbb{Z}_{q_0}[x]/\langle x^n - 1\rangle \times \mathbb{Z}_{q_1}[x]/\langle x^n - 1\rangle \\
\cong\ & \left(\prod_{i=0}^{n-1} \left(\mathbb{Z}_{q_0}[x]/\left\langle x - \omega_{q_0:n}^i\right\rangle\right)\right) \times \left(\prod_{i=0}^{n-1} \left(\mathbb{Z}_{q_1}[x]/\left\langle x - \omega_{q_1:n}^i\right\rangle\right)\right) \\
\cong\ & \prod_{i=0}^{n-1} \left(\mathbb{Z}_{q_0}[x]/\left\langle x - \omega_{q_0:n}^i\right\rangle \times \mathbb{Z}_{q_1}[x]/\left\langle x - \omega_{q_1:n}^i\right\rangle\right) \\
\cong\ & \prod_{i=0}^{n-1} \left((\mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1})[x]/\left\langle x - (\omega_{q_0:n}, \omega_{q_1:n})^i\right\rangle\right) \\
\cong\ & \prod_{i=0}^{n-1} \left(\mathbb{Z}_{q_0 q_1}[x]/\left\langle x - \omega_{q_0 q_1:n}^i\right\rangle\right)
\end{aligned}
$$

Table 8: Cycle counts for speed- and stack-optimized implementations of `MatrixVectorMul` on the Cortex-M3 and Cortex-M4. (A) and (D) denote the strategies used. The memory to hold the secret is counted in this table.

	Level	Implementation	Cycles	Bytes
M3	`LightSaber`	16-bit speed(A)	**199k**	4 096
		16-bit stack(D)	279k	**2 688(K) or 2 304(E)**
		32-bit speed(A)	249k	4 096
	`Saber`	16-bit speed(A)	**391k**	5 120
		16-bit stack(D)	631k	**3 200(K) or 2 432(E)**
		32-bit speed(A)	458k	5 120
	`FireSaber`	16-bit speed(A)	**644k**	6 144
		16-bit stack(D)	1 123k	**3 712(K) or 2 560(E)**
		32-bit speed(A)	724k	6 144
M4	`LightSaber`	32-bit speed(A)	**68k**	4 096
		hybrid stack(D)	130k	**2 688(K) or 2 304(E)**
	`Saber`	32-bit speed(A)	**136k**	5 120
		hybrid stack(D)	293k	**3 200(K) or 2 432(E)**
	`FireSaber`	32-bit speed(A)	**225k**	6 144
		hybrid stack(D)	522k	**3 712(K) or 2 560(E)**

Table 9: Cycle counts for speed- and stack-optimized implementations of `InnerProd` on the Cortex-M3 and Cortex-M4.

	Level	Implementation	Cycles		Bytes
			Dec	Enc	
M3	`LightSaber`	16-bit speed(A)	**116k**	**83k**	3 072
		16-bit stack(D)	140k	–	**2 048**
		32-bit speed(A)	155k	93k	3 072
	`Saber`	16-bit speed(A)	**164k**	**114k**	3 072
		16-bit stack(D)	210k	–	**2 048**
		32-bit speed(A)	215k	122k	3 072
	`FireSaber`	16-bit speed(A)	**211k**	**144k**	3 072
		16-bit stack(D)	281k	–	**2 048**
		32-bit speed(A)	274k	150k	3 072
M4	`LightSaber`	32-bit speed(A)	**40k**	**28k**	3 072
		hybrid stack(D)	65k	–	**2 048**
	`Saber`	32-bit speed(A)	**57k**	**39k**	3 072
		hybrid stack(D)	98k	–	**2 048**
	`FireSaber`	32-bit speed(A)	**74k**	**51k**	3 072
		hybrid stack(D)	130k	–	**2 048**

C Memory layout for Multi-moduli NTTs

Algorithm 4 16-bit (big, big) polynomial multiplication(s) requiring 3 074 bytes of memory.

Declare arrays $\begin{cases} \texttt{uint32\_t buff1\_32[256], buff2\_32[256]} \\ \texttt{uint16\_t buff1\_16[256], buff2\_16[256]} \end{cases}$

1: $\begin{cases} \texttt{buff1\_32[0-255]} = \texttt{NTT}_{64:\omega_{p_0}:128:\omega^2_{p_0}:128}(\texttt{src1[0-255]}) \\ \texttt{buff1\_16[0-255]} = \texttt{NTT}_{64:\omega_{p_1}:128:\omega^2_{p_1}:128}(\texttt{src1[0-255]}) \end{cases}$

2: $\begin{cases} \texttt{buff2\_32[0-255]} = \texttt{NTT}_{64:\omega_{p_0}:128:\omega^2_{p_0}:128}(\texttt{src2[0-255]}) \\ \texttt{buff2\_16[0-255]} = \texttt{NTT}_{64:\omega_{p_1}:128:\omega^2_{p_1}:128}(\texttt{src2[0-255]}) \end{cases}$

3: $\begin{cases} \texttt{buff1\_32[0-255]} = \texttt{base\_mul}_{64:4:\omega_{p_0}:128}(\texttt{buff1\_32[0-255]}, \texttt{buff2\_32[0-255]}) \\ \texttt{buff1\_16[0-255]} = \texttt{base\_mul}_{64:4:\omega_{p_1}:128}(\texttt{buff1\_16[0-255]}, \texttt{buff2\_16[0-255]}) \end{cases}$

4: $\begin{cases} \texttt{buff1\_32[0-255]} = \texttt{NTT}^{-1}_{64:\omega_{p_0}:128:\omega^2_{p_0}:128}(\texttt{buff1\_32[0-255]}) \\ \texttt{buff1\_16[0-255]} = \texttt{NTT}^{-1}_{64:\omega_{p_1}:128:\omega^2_{p_1}:128}(\texttt{buff1\_16[0-255]}) \end{cases}$

5: $\texttt{des[0-255]} = \texttt{CRT}(\texttt{buff1\_32[0-255]}, \texttt{buff1\_16[0-255]}) \bmod q$

Algorithm 5 16-bit (big, small) polynomial multiplication(s) requiring 1 536 bytes of memory.

Declare arrays `uint16_t buff1_16[256], buff2_16[256], buff3_16[256]`

Declare pointers $\begin{cases} \texttt{uint32\_t *buff1\_32 = (uint32\_t*)buff1\_16} \\ \texttt{uint32\_t *buff2\_32 = (uint32\_t*)buff1\_16} \end{cases}$

1: $\texttt{buff1\_32[0-255]} = \texttt{NTT}_{64:\omega_{p_0p_1}:128:\omega^2_{p_0p_1}:128}(\texttt{src1[0-255]})$

2: $\texttt{buff3\_16[0-255]} = \texttt{buff1\_32[0-255]} \bmod p_1$

3: $\texttt{buff1\_16[0-255]} = \texttt{buff1\_32[0-255]} \bmod p_0$

4: $\texttt{buff2\_16[0-255]} = \texttt{NTT}_{64:\omega_{p_1}:128:\omega^2_{p_1}:128}(\texttt{src2[0-255]})$

5: $\texttt{buff3\_16[0-255]} = \texttt{base\_mul}_{64:4:\omega_{p_1}:128}(\texttt{buff3\_16[0-255]}, \texttt{buff2\_16[0-255]})$

6: $\texttt{buff2\_16[0-255]} = \texttt{NTT}_{64:\omega_{p_0}:128:\omega^2_{p_0}:128}(\texttt{src2[0-255]})$

7: $\texttt{buff2\_16[0-255]} = \texttt{base\_mul}_{64:4:\omega_{p_0}:128}(\texttt{buff1\_16[0-255]}, \texttt{buff2\_16[0-255]})$

8: $\texttt{buff1\_32[0-255]} = \texttt{CRT}(\texttt{buff2\_16[0-255]}, \texttt{buff3\_16[0-255]})$

9: $\texttt{des[0-255]} = \texttt{NTT}^{-1}_{64:\omega_{p_0p_1}:128:\omega^2_{p_0p_1}:128}(\texttt{buff1\_32[0-255]}) \bmod q$

D Implementation of butterflies

Algorithm 6 CT_32 [ACC+21, GKS21].

Symbol: $\mathtt{R} = 2^{32}$
Constants: $\mathtt{Q}$, $\omega' = \omega\mathtt{R} \bmod {}^{\pm}\mathtt{Q}$, $\mathtt{Qprime} = -\mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$
Input: $\mathtt{c0} = a_0$, $\mathtt{c1} = a_1$
Output: $\mathtt{c0} = a_0 + \omega a_1$, $\mathtt{c1} = a_0 - \omega a_1$

```
smull tmp, c1, c1, ω′
mul tmp2, tmp, Qprime
smlal tmp, c1, tmp2, Q
                                        ▷ c1 = mMul(a₁, ω′)
add c0, c0, c1
sub c1, c0, c1, lsl #1
```

Algorithm 7 CT_2x16_SIMD [ABCG20].

Symbol: $\mathtt{R} = 2^{16}$
Constants: $\mathtt{QQprime} = \mathtt{Q}|| - \mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$, $\omega' = \omega\mathtt{R} \bmod {}^{\pm}\mathtt{Q}$
Input: $\mathtt{c0} = a_1||a_0$, $\mathtt{c1} = a_3||a_2$
Output: $\mathtt{c0} = (a_1 + \omega a_3)||(a_0 + \omega a_2)$, $\mathtt{c1} = (a_1 - \omega a_3)||(a_0 - \omega a_2)$

```
smulbb tmp, c1, ω′
smultb c1, c1, ω′
smulbb tmp2, tmp, QQprime
smlabt tmp, tmp2, QQprime, tmp
smulbb tmp2, c1, QQprime
smlabt c1, tmp2, QQprime, c1
pkhtb tmp, c1, tmp, asr #16
                                        ▷ tmp = mMul(a₃, ω′) || mMul(a₂, ω′)
ssub16 c1, c0, tmp
sadd16 c0, c0, tmp
```

Algorithm 8 CT_32_schoolbook [GKS21]

Symbol: $\mathtt{R} = 2^{32}$
Constants: $\begin{cases} 2^{16}\mathtt{Q\_h} + \mathtt{Q\_l} = \mathtt{Q} \\ 2^{16}w'_h + w'_l = \omega' = \omega\mathtt{R} \bmod {}^{\pm}\mathtt{Q} \end{cases}$, $\mathtt{Qprime} = -\mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$
Input: $\mathtt{c0} = a_0$, $2^{16}\mathtt{c1\_h} + \mathtt{c1\_l} = a_1$
Output: $\mathtt{c0} = a_0 + \omega a_1$, $\mathtt{c1\_l} = a_0 - \omega a_1$

```
smull_const tmp, tmp2, c1_l, c1_h, ω′_l, ω′_h
mul c1_h, tmp, Qprime
ubfx c1_l, c1_h, #0, #16
sbfx c1_h, c1_h, #16, #16
smlal_const tmp, tmp2, c1_l, c1_h, Q_l, Q_h, tmp3
                                        ▷ tmp2 = mMul(a₁, ω′)
sub c1_l, c0, tmp2
add c0, c0, tmp2
```

Algorithm 9 `CT_16` [GKS21].

Symbol: $\mathtt{R} = 2^{16}$
Constants: $\mathtt{Q}$, $\omega' = \omega\mathtt{R} \bmod {}^{\pm}\mathtt{Q}$, $\mathtt{Qprime} = -\mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$
Input: $\mathtt{c0} = a_0$, $\mathtt{c1} = a_1$
Output: $\mathtt{c0} = a_0 + \omega a_1$, $\mathtt{c1} = a_0 - \omega a_1$

```
mul c1, c1, ω'
mul tmp, c1, Qprime
sxth tmp, tmp
mla c1, tmp, Q, c1
                                  ▷ c1 = mMul(a_1, ω')
add c0, c0, c1, asr #16
sub c1, c0, c1, asr #15
```

Algorithm 10 Schoolbook long multiplication `smull_const` [GKS21]

Input: $\begin{cases} 2^{16}\mathtt{a\_h} + \mathtt{a\_l} = a \\ 2^{16}\mathtt{b\_h} + \mathtt{b\_l} = b \end{cases}$
Output: $2^{32}\mathtt{c\_h} + \mathtt{c\_l} = a \cdot b$

```
mul c_l, a_l, b_l
mul c_h, a_h, b_h
mul a_h, a_h, b_l
mla a_h, a_l, b_h, a_h
adds c_l, c_l, a_h, lsl #16
adc c_h, c_h, a_h, asr #16
```

Algorithm 11 Schoolbook long multiplication `smlal_const` [GKS21]

Input: $\begin{cases} 2^{16}\mathtt{a\_h} + \mathtt{a\_l} = a \\ 2^{16}\mathtt{b\_h} + \mathtt{b\_l} = b \\ 2^{32}\mathtt{c\_h} + \mathtt{c\_l} = c \end{cases}$
Output: $2^{32}\mathtt{c\_h} + \mathtt{c\_l} = c + a \cdot b$

```
mul tmp, a_l, b_l
adds c_l, c_l, tmp
mul tmp, a_h, b_h
adc c_h, c_h, tmp
mul tmp, a_h, b_l
mla tmp, a_l, b_h, tmp
adds c_l, c_l, tmp, lsl #16
adc c_h, c_h, tmp, asr #16
```

Algorithm 12 `GS_32_schoolbook` [GKS21]

Symbol: $\mathtt{R} = 2^{32}$
Constants: $\begin{cases} 2^{16}\mathtt{Q\_h} + \mathtt{Q\_l} = \mathtt{Q} \\ 2^{16}w'_h + w'_l = \omega' = \omega\mathtt{R} \bmod {}^{\pm}\mathtt{Q} \end{cases}$, $\mathtt{Qprime} = -\mathtt{Q}^{-1} \bmod {}^{\pm}\mathtt{R}$
Input: $\mathtt{c0} = a_0$, $\mathtt{c1} = a_1$
Output: $\mathtt{c0} = a_0 + a_1$, $\mathtt{c1} = \omega(a_0 - a_1)$

```
sub tmp, c0, c1
add c0, c0, c1
ubfx tmp2, tmp, #0, #16
asr tmp, tmp, #16
smull_const tmp1, c1, tmp2, tmp, ω'_l, ω'_h
mul tmp, tmp1, Qprime
ubfx tmp2, tmp, #0, #16
sbfx tmp, tmp, #16, #16
smlal_const tmp1, c1, tmp2, tmp, Q_l, Q_h, tmp3
                                  ▷ c1 = mMul((a_0 - a_1), ω')
```

E CT butterflies for NTT and iNTT

NTT with CT butterflies for radix-2 cyclic convolution. Let $2^k | \mathbf{0}(m)$, and ω be a principal 2^k-th root of unity.

The FFT trick with CT butterflies for $\mathtt{NTT}_{2^k:1:\omega}$ over $x^{2^k} - 1$ is applying the isomorphism

$$\begin{aligned}\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle &\cong \mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - 1 \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - \omega^{2^{k-1}} \right\rangle \\ &\cong \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - 1 \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \omega^{2^{k-1}} \right\rangle \\ &\quad \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \omega^{2^{k-2}} \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \omega^{2^{k-1}+2^{k-2}} \right\rangle\end{aligned}$$

to the polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{2^k-1} a_i x^i$ recursively.

Explicitly, $\boldsymbol{a}(x)$ is mapped to $\begin{cases} \sum_{i=0}^{2^{k-1}} (a_i + a_{i+2^{k-1}})\, x^i = \boldsymbol{a}(x) \bmod (x^{2^{k-1}} - 1) \\ \sum_{i=0}^{2^{k-1}} (a_i - a_{i+2^{k-1}})\, x^i = \boldsymbol{a}(x) \bmod (x^{2^{k-1}} - \omega^{2^{k-1}}) \end{cases}$

and then to $\begin{cases} \boldsymbol{a}(x) \bmod (x^{2^{k-2}} - 1) \\ \boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \omega^{2^{k-1}}) \\ \boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \omega^{2^{k-2}}) \\ \boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \omega^{2^{k-1}+2^{k-2}}) \end{cases}$. If we apply the isomorphism all the way down to linear polynomials, we see that the result is the bit-reversal of $\boldsymbol{a}(1), \boldsymbol{a}(\omega), \ldots, \boldsymbol{a}(\omega^{2^k-1})$.

iNTT with CT butterflies for radix-2 cyclic convolution. To implement $\mathtt{NTT}^{-1}_{2^k:1:\omega}$ with CT butterflies, we only need to operate on the bit-reversed polynomial with inverted ω. This approach is called 'decimation in time' in the literature. Essentially, the isomorphism is the same as $\mathtt{NTT}_{2^k:1:\omega^{-1}}$. We suggest interested readers to refer to [Ber] for writing down the isomorphism. In this section, we only write down the indices explicitly as follows.

Consider re-writing $\left(b^{(0)}_{\mathtt{bitrev}_k(i)}\right)_{0\le i<2^k:(1)} = \left(b_{\mathtt{bitrev}_k(i)}\right)_{0\le i<2^k:(1)}$ as

$$\left(b^{(0)}_{\mathtt{bitrev}_{k-1}(i)}, b^{(0)}_{\mathtt{bitrev}_{k-1}(i)+1}\right)_{0\le i<2^{k-1}:(1)},$$

then $\mathtt{NTT}^{-1}_{2^k:1:\omega}$ with CT butterflies is applying the computation

$$\begin{aligned} & \left(b^{(0)}_{\mathtt{bitrev}_{k-1}(i)}, b^{(0)}_{\mathtt{bitrev}_{k-1}(i)+1}\right)_{0\le i<2^{k-1}:(1)} \\ \mapsto\quad & \left(\left(b^{(0)}_{\mathtt{bitrev}_{k-1}(i)} + b^{(0)}_{\mathtt{bitrev}_{k-1}(i)+1}\right) + \left(b^{(0)}_{\mathtt{bitrev}_{k-1}(i)} - b^{(0)}_{\mathtt{bitrev}_{k-1}(i)+1}\right) x\right)_{0\le i<2^{k-1}:\left(1,\omega^{-2^{k-1}}\right)} \\ =:\quad & \left(b^{(1)}_{\mathtt{bitrev}_{k-1}(i)}\right)_{0\le i<2^{k-1}:\left(1,\omega^{-2^{k-1}}\right)} \end{aligned}$$

recursively.

In general, at layer $k-l$, the map below is computed,

$$\begin{aligned} & \left(b^{(k-l)}_{\mathtt{bitrev}_l(i)}\right)_{0\le i<2^l:\left(1,\omega^{\mathtt{bitrev}_{k-l}(1)},\ldots,\omega^{\mathtt{bitrev}_{k-l}(2^{k-l}-1)}\right)} \\ \mapsto\quad & \left(b^{(k-l+1)}_{\mathtt{bitrev}_{l-1}(i)}\right)_{0\le i<2^{l-1}:\left(1,\omega^{\mathtt{bitrev}_{k-l+1}(1)},\ldots,\omega^{\mathtt{bitrev}_{k-l+1}(2^{k-l+1}-1)}\right)} \end{aligned}.$$

It is easily seen that if the input is the bit-reversal of $\boldsymbol{a}(1)$, $\boldsymbol{a}(\omega)$, $\ldots$, $\boldsymbol{a}(\omega^{n-1})$, the result is $2^k\boldsymbol{a}(x)$. Finally, we multiply each coefficient of the result by $\frac{1}{2^k}$ to derive $\boldsymbol{a}(x)$.

As a side note, we also give an improved implementation for radix-2 cyclic NTT in Appendix F. This is only a slight improvement, but it shows that even for the most commonly known NTT, there are still optimizations left.

NTT and iNTT with CT butterflies for convolution in general. Suppose ζ is invertible, and define $\texttt{NTT}_{2^k:\zeta:\omega}$ as in Equation 5, with the inverse $\texttt{NTT}^{-1}_{2^k:\zeta:\omega}$.

We show here how to implement the map with CT butterflies.

The easiest way to implement $\texttt{NTT}_{2^k:\zeta:\omega}$ with CT butterflies is to twist $\mathbb{Z}_m[x]/\left\langle x^{2^k} - \zeta^{2^k} \right\rangle$ to $\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle$ and proceed with CT butterflies for $\texttt{NTT}_{2^k:1:\omega}$ over $x^{2^k} - 1$. Readers can verify that the output is the bit-reversal of $\boldsymbol{a}(\zeta)$, $\boldsymbol{a}(\zeta\omega)$, $\ldots$, $\boldsymbol{a}(\zeta\omega^{2^k-1})$. Immediately, we also see that $\texttt{NTT}^{-1}_{2^k:\zeta:\omega}$ can be implemented by first computing $\texttt{NTT}^{-1}_{2^k:1:\omega}$ as in $\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle$ and then twisting $\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle$ to $\mathbb{Z}_m[x]/\left\langle x^{2^k} - \zeta^{2^k} \right\rangle$. If $\zeta \neq 1$, at the end of $\texttt{NTT}^{-1}_{2^k:1:\omega}$, we can merge multiplication by $\frac{1}{2^k}$ with twisting $\mathbb{Z}_m[x]/\left\langle x^{2^k} - 1 \right\rangle$ to $\mathbb{Z}_m[x]/\left\langle x^{2^k} - \zeta^{2^k} \right\rangle$.

For implementing $\texttt{NTT}_{2^k:\zeta:\omega}$, we can merge the twist with CT butterflies. Applying the isomorphisms

$$
\begin{aligned}
\mathbb{Z}_m[x]/\left\langle x^{2^k} - \zeta^{2^k} \right\rangle \cong\ & \mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - \zeta^{2^{k-1}} \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-1}} - \zeta^{2^{k-1}}\omega^{2^{k-1}} \right\rangle \\
\cong\ & \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \zeta^{2^{k-2}} \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-1}} \right\rangle \\
& \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-2}} \right\rangle \times \mathbb{Z}_m[x]/\left\langle x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-1}+2^{k-2}} \right\rangle
\end{aligned}
$$

$$
\begin{cases}
\boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \zeta^{2^{k-2}}) \\
\boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-1}}) \\
\boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-2}}) \\
\boldsymbol{a}(x) \bmod (x^{2^{k-2}} - \zeta^{2^{k-2}}\omega^{2^{k-1}+2^{k-2}})
\end{cases}
$$

. It is now clear that recursively applying the isomorphisms outputs the bit-reversal of $\boldsymbol{a}(\zeta)$, $\boldsymbol{a}(\zeta\omega)$, $\ldots$, $\boldsymbol{a}(\zeta\omega^{2^k-1})$.

F Faster CT-GS butterflies for cyclic NTT/iNTT

In this section we show a faster butterfly implementation for NTT/iNTT over $x^8 - 1$. We coin the term CT-GS butterfly for the implementation since it can be derived from either CT or GS butterfly. We first illustrate the idea with CT butterflies. Let's say implementing NTT of $(a_0, \ldots, a_7)$ over $x^8 - 1$ with CT butterflies is to derive $(a_0''', \ldots, a_7''')$ as follows:

1. $(a_0, \ldots, a_7) \mapsto (a_0', \ldots, a_7')$ where

$$\begin{aligned}(a_0', \ldots, a_3') &= (a_0, \ldots, a_3) + (a_4, \ldots, a_7) \\ (a_4', \ldots, a_7') &= (a_0, \ldots, a_3) - (a_4, \ldots, a_7)\end{aligned}$$

2. $(a_0', \ldots, a_7') \mapsto (a_0'', \ldots, a_7'')$ where

$$\begin{aligned}(a_0'', a_1'') &= (a_0', a_1') + (a_2', a_3') \\ (a_2'', a_3'') &= (a_0', a_1') - (a_2', a_3') \\ (a_4'', a_5'') &= (a_4', a_5') + \omega_4(a_6', a_7') \\ (a_6'', a_7'') &= (a_4', a_5') - \omega_4(a_6', a_7')\end{aligned}$$

3. $(a_0'', \ldots, a_7'') \mapsto (a_0''', \ldots, a_7''')$ where

$$\begin{aligned}a_0''' &= a_0'' + a_1'' \\ a_1''' &= a_0'' - a_1'' \\ a_2''' &= a_2'' + \omega_4 a_3'' \\ a_3''' &= a_2'' - \omega_4 a_3'' \\ a_4''' &= a_4'' + \omega_8 a_5'' \\ a_5''' &= a_4'' - \omega_8 a_5'' \\ a_6''' &= a_6'' + \omega_8^3 a_7'' \\ a_7''' &= a_6'' - \omega_8^3 a_7''\end{aligned}$$

The computation can be re-written as $(a_0, a_2, a_4, a_6) \mapsto (a_0'', \omega_4 a_2'', \omega_8 a_4'', \omega_8^3 a_6'')$ and $(a_1, a_3, a_5, a_7) \mapsto (a_1'', \omega_4 a_3'', \omega_8 a_5'', \omega_8^3 a_7'')$, followed by

$$\texttt{addSub4}\left((a_0'', \omega_4 a_2'', \omega_8 a_4'', \omega_8^3 a_6''), (a_1'', \omega_4 a_3'', \omega_8 a_5'', \omega_8^3 a_7'')\right)$$

where `addSub4` is component-wise add-sub giving a pair as result.

We present here a faster computation for $(a_1, a_3, a_5, a_7) \mapsto (a_1'', \omega_4 a_3'', \omega_8 a_5'', \omega_8^3 a_7'')$ as follows:

1. $(a_1, a_3, a_5, a_7) \mapsto (a_1', a_3', a_5', a_7')$
2. $(a_1', a_3') \mapsto (a_1'', a_3'')$
3. $a_3'' \mapsto \omega_4 a_3''$
4. $(a_5', a_7') \mapsto (\omega_8 a_5' + \omega_8^3 a_7', \omega_8^3 a_5' + \omega_8 a_7')$ where

$$\begin{aligned}&(\omega_8 a_5' + \omega_8^3 a_7', \omega_8^3 a_5' + \omega_8 a_7') \\ =& \left(\omega_8(a_5' + \omega_8^2 a_7'), \omega_8^3(a_5' + \omega_8^6 a_7')\right) \\ =& \left(\omega_8(a_5' + \omega_4 a_7'), \omega_8^3(a_5' + \omega_4^3 a_7')\right) \\ =& \left(\omega_8(a_5' + \omega_4 a_7'), \omega_8^3(a_5' - \omega_4 a_7')\right) \\ =& \left(\omega_8 a_5'', \omega_8^3 a_7''\right)\end{aligned}$$

Therefore, we can compute with 2 `smull`s and 2 `smlal`s for the 64-bit value of $(\omega_8 a_5' + \omega_8^3 a_7', \omega_8^3 a_5' + \omega_8 a_7')$ and then reduce them to 32-bit. To sum up, we are replacing 3 `smull`s + 3 Montgomery reductions + 1 add-sub with 2 `smull`s + 2 `smlal`s + 2 Montgomery reductions and save 3 cycles.

To see how to derive the same shape of computation from GS butterflies, let's say implementing NTT of $(a_0, \ldots, a_7)$ over $x^8 - 1$ with GS butterflies is to derive $(a_0''', \ldots, a_7''')$ as follows:

1. $(a_0, \ldots, a_7) \mapsto (a'_0, \ldots, a'_7)$ where

$$\begin{array}{rcr} a'_0 & = & a_0 + a_4 \\ a'_1 & = & a_1 + a_5 \\ a'_2 & = & a_2 + a_6 \\ a'_3 & = & a_3 + a_7 \\ a'_4 & = & a_0 - a_4 \\ a'_5 & = & (a_1 - a_5)\omega_8 \\ a'_6 & = & (a_2 - a_6)\omega_4 \\ a'_7 & = & (a_3 - a_7)\omega_8^3 \end{array}$$

2. $(a'_0, \ldots, a'_7) \mapsto (a''_0, \ldots, a''_7)$ where

$$\begin{array}{rcr} (a''_0, a''_4) & = & (a'_0, a'_4) + (a'_2, a'_6) \\ (a''_1, a''_5) & = & ((a'_1, a'_5) + (a'_3, a'_7))\,\omega_4 \\ (a''_2, a''_6) & = & (a'_0, a'_4) - (a'_2, a'_6) \\ (a''_3, a''_7) & = & ((a'_1, a'_5) - (a'_3, a'_7))\,\omega_4 \end{array}$$

3. $(a''_0, \ldots, a''_7) \mapsto (a'''_0, \ldots, a'''_7)$ where

$$\begin{array}{rcl} (a'''_0, a'''_2, a'''_4, a'''_6) & = & (a''_0, a''_2, a''_4, a''_6) + (a''_1, a''_3, a''_5, a''_7) \\ (a'''_1, a'''_3, a'''_5, a'''_7) & = & (a''_0, a''_2, a''_4, a''_6) - (a''_1, a''_3, a''_5, a''_7) \end{array}$$

We can re-write the computation $(a_0, \ldots, a_7) \mapsto (a''_0, \ldots, a''_7)$ as

$$\begin{cases} (a_0, a_2, a_4, a_6) \mapsto (a'_0, a'_2, a'_4, a'_6) \\ (a_1, a_3, a_5, a_7) \mapsto (a'_1, a'_3, a_1 - a_5, a_3 - a_7) \end{cases}$$

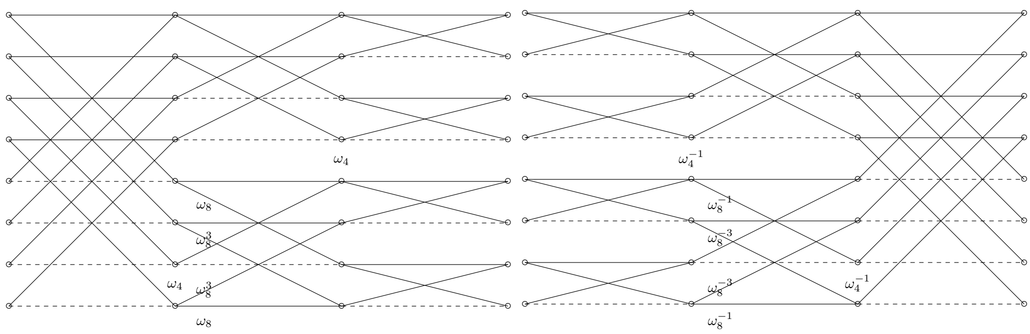
followed by

$$\begin{cases} (a'_0, a'_2, a'_4, a'_6) \mapsto (a''_0, a''_2, a''_4, a''_6) \\ (a'_1, a'_3) \mapsto (a''_1, a''_3) \\ (a_1 - a_5, a_3 - a_7) \mapsto (a''_5, a''_7) \end{cases}$$

Now if we compute $(a_1 - a_5, a_3 - a_7) \mapsto (a''_5, a''_7)$ as

$$\begin{array}{rcl} (a_1 - a_5, a_3 - a_7) & \mapsto & \big((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, (a_1 - a_5)\omega_8^3 + (a_3 - a_7)\omega_8\big) \\ & = & \big((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, (a_1 - a_5)\omega_8^3 - (a_3 - a_7)\omega_8^5\big) \\ & = & \big((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, \big((a_1 - a_5)\omega_8 - (a_3 - a_7)\omega_8^3\big)\,\omega_8^2\big) \\ & = & \big((a_1 - a_5)\omega_8 + (a_3 - a_7)\omega_8^3, \big((a_1 - a_5)\omega_8 - (a_3 - a_7)\omega_8^3\big)\,\omega_4\big) \\ & = & (a''_5, a''_7) \end{array}$$

then we derive the same shape of computation.



(a) Modified butterflies for NTT over $x^8 - 1$. (b) Modified butterflies for iNTT over $x^8 - 1$.

Figure 6: Modified butterflies.

G C code for development

We write some C code to facilitate the development of assembly code. The C code is *not* for speed and security. Neither of the requirements are matched. The C code is to relieve programmers from starting from scratch with C when moving to a different architecture.

The most important feature supported by the C code is:

customizable merging of NTTs supporting (incomplete)radix-2 splits.

The C code consists of two parts:

1. Customizable strategy for generating twiddle factors.
2. Customizable and separable NTT call(s) in C to access memory exactly as what is planned to be later implemented in assembly.

Some differences from assembly implementation are that:

- The C code is using % for modular reduction, so there is no scaling as there are for Montgomery multiplication.
- The range of each computation in C is completely in $[-\frac{q}{2}, \frac{q}{2})$. So don't use the code for range analysis.

This section is organized as follows: Section G.1 provides a snapshot of the C code with the C structure `struct compress_profile`. Section G.2 introduces how twiddle factors are generated. Section G.3 illustrates the NTT in C accessing memory exactly as what is planned to be later implemented in assembly.

G.1 `struct compress_profile` and `NTT_params.h`

The central idea of the C code relies on the structure providing a snapshot on how layers are merged, in particular, how many layers are merged at a certain point.

Consider the declaration of the C structure:

```
struct compress_profile{
    int compressed_layers;
    int merged_layers[16];
};
```

`compressed_layers` tells us how many layers there are after the merge. `merged_layers` tells us at each merged layers `merged_layers[0-(compressed_layers - 1)]`, how many layers there are originally. To provide a more concrete pattern of NTT/iNTT, we also reserve the following symbols:

- `ARRAY_N`
- `NTT_N` dividing `ARRAY_N`
- `LOGNTT_N`

Once a `struct compress_profile` is declare the reserved symbols are defined, we now have the blueprint of our NTT implementation: We compute length-`NTT_N` on a length-`ARRAY_N` array. The `LOGNTT_N` layers of splitting is compressed into `compressed_layers` layers. After the compression, for each $i = 0, 1, \ldots,$ `compressed_layers` $- 1$, the i-th layer consists of `merged_layers[`i`]` layers of the splits.

To define the actual arithmetic for NTTs, one need to provide a header file `NTT_params.h`. The required fields for the header file vary from targeted implementations. In the most simplest form, one can produce NTT implementation in C by providing:

- Modulus `Q` coprime to `NTT_N`
- `invNQ` = `NTT_N`$^{-1}$ mod $^{\pm}$`Q`
- For cyclic NTT:
  - Principal `NTT_N`-th root of unity `omegaQ`
  - `invomegaQ` = `omegaQ`$^{-1}$ mod $^{\pm}$`Q`
- For negacyclic NTT:
  - Principal 2`NTT_N`-th root of unity `omegaQ`
  - `invomegaQ` = `omegaQ`$^{-1}$ mod $^{\pm}$`Q`

The produced C code for NTT can serve as the foundation for further assembly optimizations where individual merged layers can be replaced one at a time in any order.

To provide a full support of NTTs in assembly, one need to specify the size `R` for Montgomery reduction. The minimal requirement for `NTT_params.h` is therefore as follows:

- Modulus `Q` coprime to `NTT_N` and coprime to `R`
- `invNQ` = `NTT_N`$^{-1}$ mod $^{\pm}$`Q`
- Auxiliary factor `RmodQ` = `R` mod $^{\pm}$`Q` for multiplication by 1 with Montgomery multiplication
- Montgomery factor −`Q`$^{-1}$ mod $^{\pm}$`R`(or `Q`$^{-1}$ mod $^{\pm}$`R` if subtraction is used in Montgomery multiplication)
- For cyclic NTT:
  - Principal `NTT_N`-th root of unity `omegaQ`
  - `invomegaQ` = `omegaQ`$^{-1}$ mod $^{\pm}$`Q`
- For negacyclic NTT:
  - Principal 2`NTT_N`-th root of unity `omegaQ`
  - `invomegaQ` = `omegaQ`$^{-1}$ mod $^{\pm}$`Q`

G.2 gen_table.h

Now we go into the details on generating tables of twiddle factors. If one's purpose is to implement NTT in assembly with a single shot, then this section suffices. However, we strongly suggest readers to at least perform some calls of NTTs in C to see if the tables are generated as desired. We will elaborate the C implementation of NTT in the next section.

First, we generate twiddle factors for cyclic NTTs.

`gen_CT_table` generates a table of twiddle factors scaled by `scale` without compression and is an auxiliary function for `gen_streamlined_CT_table`.

`gen_streamlined_CT_table` generates a table of twiddle factors scaled by `scale` with compression implied by `_profile` where `pad` should always be 0 for C implementation and is occasionally 1 for assembly implementation with SIMD instructions.

```
void gen_CT_table \
    (int *des, int scale, int _omega, int _Q);
void gen_streamlined_CT_table \
    (int *des, int scale, int _omega, int _Q, \
    struct compress_profile *_profile, \
    int pad);
```

Now we generate twiddle factors for cyclic iNTTs.

`gen_inv_CT_table` generates a table of twiddle factors scaled by `scale` without compression and is an auxiliary function for `gen_streamlined_inv_CT_table`.

`gen_streamlined_inv_CT_table` generates a table of twiddle factors scaled by `scale` with compression implied by `_profile` where `pad` should always be 0 for C implementation and is occasionally 1 for assembly implementation with SIMD instructions.

```
void gen_inv_CT_table \
    (int *des, int scale, int _omega, int _Q);
void gen_streamlined_inv_CT_table \
    (int *des, int scale, int _omega, int _Q, \
    struct compress_profile *_profile, \
    int pad);
```

For polynomial multiplication in Saber, we need negacyclic NTT/iNTT. For negacyclic NTT and iNTT each of them can be implemented in at least two ways: NTT can be implemented directly or as a cyclic NTT following a twist, and iNTT can be implemented directly or as a cyclic iNTT preceding a twist. We first describe how to generate twiddle factors for twisting.

`gen_twist_table` generates a table of twiddle factors for twisting $x^{\texttt{NTT\_N}} - \texttt{\_omega}^{\texttt{NTT\_N}}$ to $y^{\texttt{NTT\_N}} - 1$ and by replacing `_omega` with $\texttt{\_omega}^{-1}$ it generates a table of twiddle factors for twisting $y^{\texttt{NTT\_N}} - 1$ to $x^{\texttt{NTT\_N}} - \texttt{\_omega}^{\texttt{NTT\_N}}$. In general, `gen_twist_table` generates a table of twiddle factors for twisting $x^{\texttt{NTT\_N}} - \zeta^{\texttt{NTT\_N}}$ to $y^{\texttt{NTT\_N}} - \xi^{\texttt{NTT\_N}}$ by setting $\texttt{\_omega} = \zeta\xi^{-1}$.

Now we describe how to generate tables of twiddle factors for negacyclic NTTs and iNTTs.

`gen_CT_negacyclic_table` generates a table of twiddle factors scaled by `scale` without compression and is an auxiliary function for `gen_streamlined_CT_negacyclic_table`.

`gen_streamlined_CT_negacyclic_table` generates a table of twiddle factors scaled by `scale` with compression implied by `_profile` for negacyclic NTT.

`gen_streamlined_inv_CT_negacyclic_table` generates a table of twiddle factors with compression implied by `_profile` for negacyclic iNTT where `pad` is occasionally 1 for assembly implementation with SIMD instructions; the twiddle factors for butterflies are scaled by `scale1` and the ones for twisting are scaled by `scale2`.

```
void gen_twist_table \
    (int *des, int scale, int _omega, int _Q);
void gen_CT_negacyclic_table \
    (int *des, int scale, int _omega, int _Q);
void gen_streamlined_CT_negacyclic_table \
    (int *des, int scale, int _omega, int _Q, \
    struct compress_profile *_profile, \
    int pad);
void gen_streamlined_inv_CT_negacyclic_table \
    (int *des, \
    int scale1, int _omega, \
    int scale2, int twist_omega, int _Q, \
    struct compress_profile *_profile, \
    int pad);
```

The last thing to deal with is the generation of twiddle factors for `base_mul`.

`gen_mul_table` generates the bit-reversal of $\texttt{\_omega}^0, \texttt{\_omega}^1, \ldots, \texttt{\_omega}^{\frac{\texttt{NTT\_N}}{2}-1}$ and scale them by `scale`.

`gen_all_mul_table` generates the bit-reversal of $\texttt{\_omega}^0, \texttt{\_omega}^1, \ldots, \texttt{\_omega}^{\texttt{NTT\_N}-1}$ and scale them by `scale`.

```
void gen_mul_table \
    (int *des, int scale, int _omega, int _Q);
void gen_all_mul_table \
    (int *des, int scale, int _omega, int _Q);
```

G.3 ntt_c.h

We provide C implementation for NTTs that are planned to be later implemented in assembly.

`CT_butterfly` computes the CT butterfly of the pair (`src[indx_a]`, `src[indx_b]`) and the constant `twiddle`. It is also a building block for `_m_layer_CT_butterfly` and `_m_layer_inv_CT_butterfly`.

```
void CT_butterfly \
    (int *src, \
    int indx_a, int indx_b, \
    int twiddle, int _Q);
```

`_m_layer_CT_butterfly` computes an `layers`-layer-CT-butterfly defined on the $2^{\texttt{layers}}$ entries `src[0 * step]`, `src[1 * step]`, $\dots$, `src[(1 « layers) * step]` and it is used in `compressed_CT_NTT` for computing merged CT butterflies.

`_m_layer_inv_CT_butterfly` computes an inverse of `layers`-layer-CT-butterfly defined on the $2^{\texttt{layers}}$ entries `src[0 * step]`, `src[1 * step]`, $\dots$, `src[(1 « layers) * step]` and it is used in `compressed_CT_inv_NTT` for computing merged CT butterflies.

```
void _m_layer_CT_butterfly \
    (int *src, \
    int layers, int step, \
    int *_root_table, int _Q);
void _m_layer_inv_CT_butterfly \
    (int *src, \
    int layers, int step, \
    int *_root_table, int _Q);
```

`compressed_CT_NTT` computes the customized NTT implied by `_profile`. The function will compute CT butterflies with `_m_layer_CT_butterfly` from layer `start_level` to layer `end_level` where $0 \leq$ `start_level` $\leq$ `end_level` $<$ `_profile->compressed_layers`.

`compressed_inv_CT_NTT` computes the customized NTT implied by `_profile`. The function will compute CT butterflies with `_m_layer_inv_CT_butterfly` from layer `start_level` to layer `end_level` where $0 \leq$ `start_level` $\leq$ `end_level` $<$ `_profile->compressed_layers`.

```
void compressed_CT_NTT \
    (int *src, \
    int start_level, int end_level, \
    int *_root_table, int _Q, \
    struct compress_profile *_profile);
void compressed_inv_CT_NTT \
    (int *src, \
    int start_level, int end_level, \
    int *_root_table, int _Q, \
    struct compress_profile *_profile);
```