On Modular Arithmetic and Polynomial Multiplication in Lattice-based Cryptography

-- Doctoral Defense for the PhD of HKBU

PhD Candidate: Junhao HUANG (黄军浩)
Supervisor: Dr. Donglong CHEN
03/06/2025

Outline

01 Introduction

1.1 Background

1.2 Lattice-based Cryptography

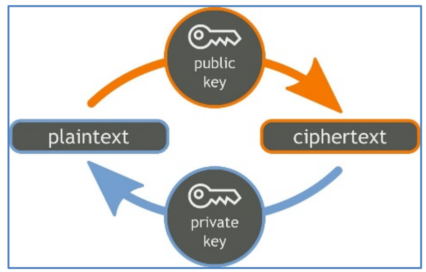
1.3 Contributions

1.1.1 Quantum Computers

Quantum computers are being developed rapidly. **Shor's algorithm** in quantum computers would break the existing **public-key cryptosystem (PKC)** in **polynomial time**.

2021: **66b** Zuchongzhi;
2021: IBM's **126b** Eagle;
2022: IBM's **433b** Osprey.
2024: Google's **Willow**
5min vs 10^{25} years

Shor



RSA

ElGamal

ECC

Threat

Protocols

DNS, TLS, SSH...

Browsers

Chrome, Safari, Edge…

Messaging

WhatsApp, Facebook, Wechat…

This prompted the cryptographic community to search for **suitable alternatives** to traditional PKC.

1.1.2 Post-quantum Cryptography

NIST initiated a standardization project in 2016 to solicit, evaluate, and standardize the **post-quantum cryptographic algorithms (PQC). Chinese ICCS** started to call for commercial PQC standardization in 2025 [1].

Table 1: Round 3 and Round 4 NIST PQC finalists

Round	Round 3		Round 4	
Types	KEM	DSA	KEM	DSA
Schemes	**Kyber**	**Dilithium**	**Kyber (ML-KEM)**	**Dilithium (ML-DSA)**
	Saber	**Falcon**	-	**Falcon (FN-DSA)**
	NTRU	Rainbow	-	Sphincs+ (SLH-DSA)
	Classic McEliece	-	-	-

Lattice-Based Cryptography (LBC) is the most promising alternative in terms of security and efficiency. Therefore, we **will focus on LBC.**

[1] https://www.niccs.org.cn/

1.1.3 Internet of Things

The **Internet of Things (IoT)** is pervasive in many aspects of modern life, such as smart healthcare, smart transportation, industrial IoT, smart tourism, and wearable technology.

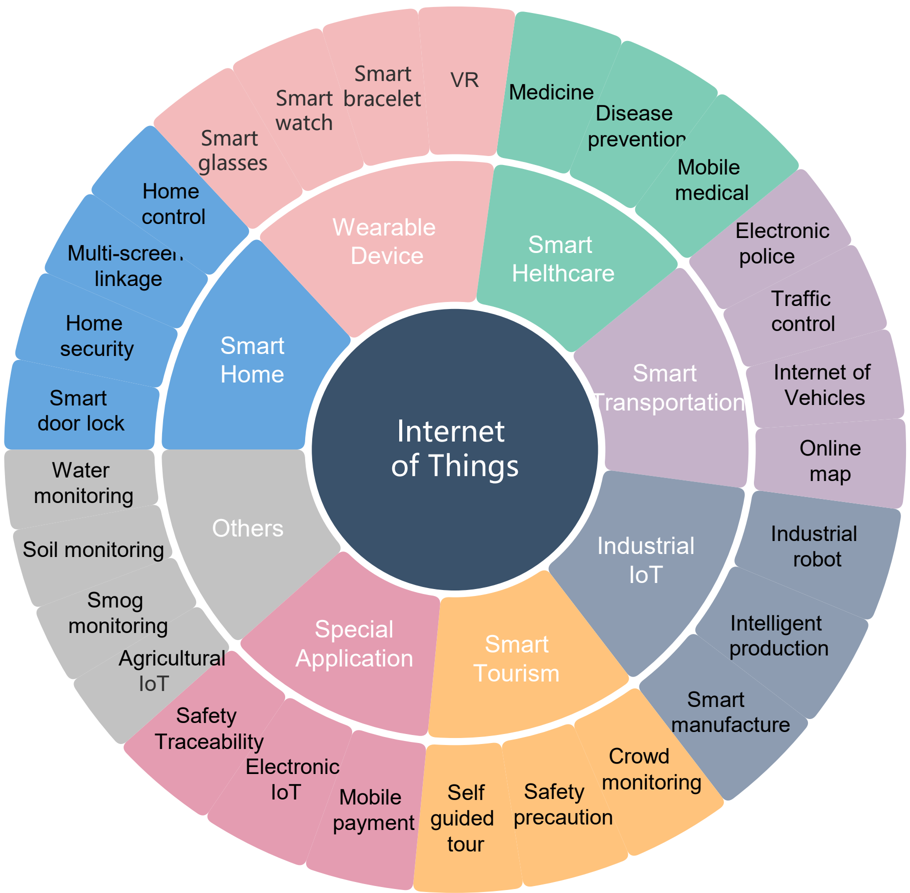
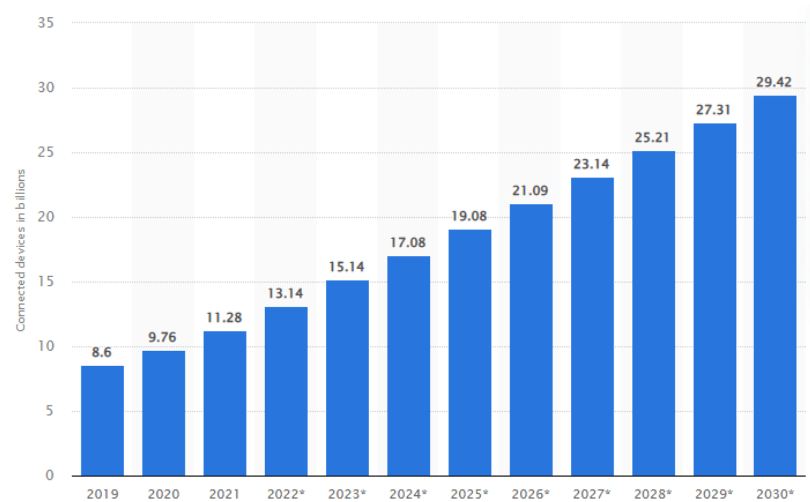


Table 2: Number of Internet of Things (IoT) connected devices worldwide (billion) from 2019 to 2021



It requires huge effort to protect billions of IoT devices from the threat of quantum computing.

1.1.4 PQC on the IoT: Challenges

The IoT devices are distinct from the traditional CPUs.

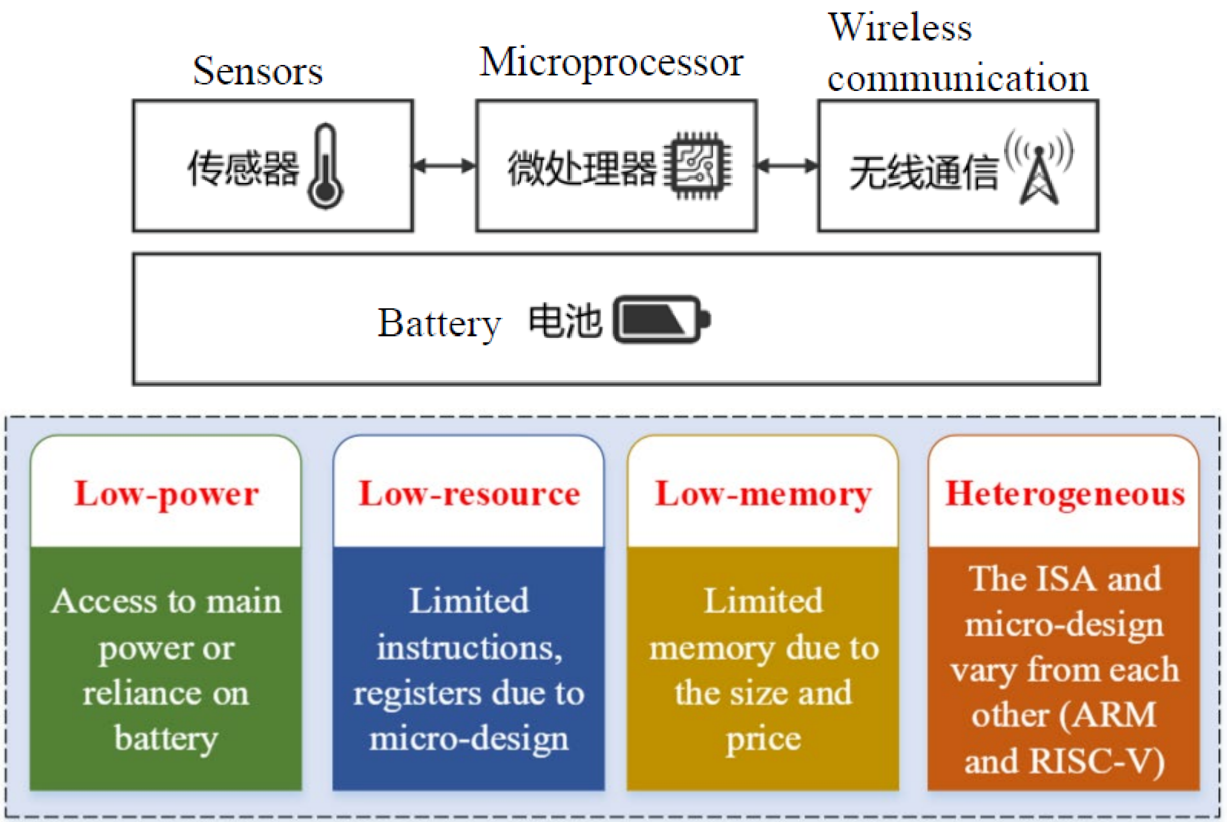


Figure 1.1: Four major limitations of IoT devices

Challenges: explore the **efficient, lightweight and secure LBC implementation** tailored for heterogeneous IoT devices.

1.2.1 Lattice-based Cryptography

Lattice-based cryptography relies on the computational difficulty of lattice:

$$\mathcal{L}(b_1, \dots, b_m) = \{\sum_{i=1}^{m} x_i b_i, x_i \in \mathbb{Z}\}$$

, where $b_1, \dots, b_m$ are basis vectors. The lattice can be expressed as the sum of $x_i b_i$.

The hardness of two LBC finalists **Kyber and Dilithium** are based on the **MLWE** and **MSIS** problems:

- **Module Short Integer Solution (MSIS):** Given an $n \times m$ lattice $\boldsymbol{A} \in \mathbb{Z}_q^{n\times m}$, find a nonzero short integer vector $\boldsymbol{x} \in \mathbb{Z}^m$ satisfying $\boldsymbol{Ax} = \boldsymbol{0} \bmod q$.

- **Module Learning with Errors (MLWE):** Given an $n \times m$ lattice $\boldsymbol{A} \in \mathbb{Z}_q^{n\times m}$ and a randomly generated sample $\boldsymbol{e}$, recover $\boldsymbol{s} \in \mathbb{Z}_q^n$ from $(\boldsymbol{A}, \boldsymbol{A}^{\boldsymbol{T}}\boldsymbol{s} + \boldsymbol{e} \bmod q)$.

1.2.2 LBC Core Operations and Structure

❑ Time and memory consuming operations

- **Polynomial sampling:** SHA-3 (Keccak, 70% of running time)
- **Polynomial multiplication:** NTT/INTT ($O(nlogn)$ & modular arithmetic);
- **Matrix-vector product:** large memory consumption.

❑ LBC structure

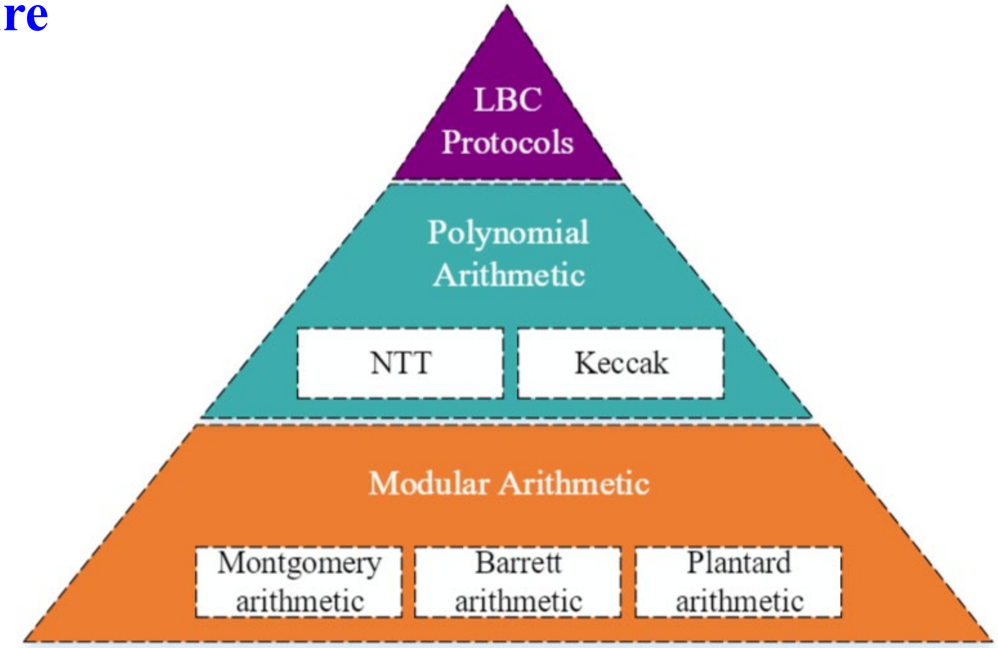
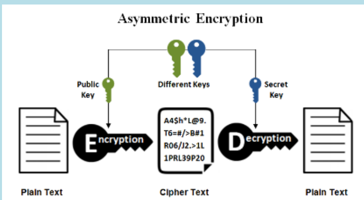


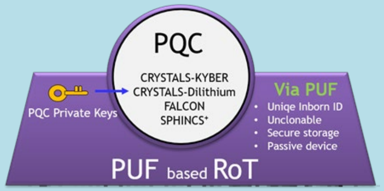
Figure 1.2: An overall structure of the LBC schemes

1.2.3 Cryptographic Engineering

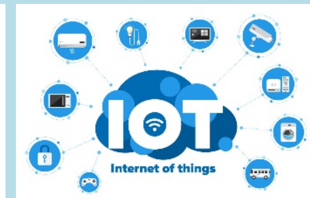
❑ Cryptography deployment in real-world devices

Cryptography: Theoretical Security



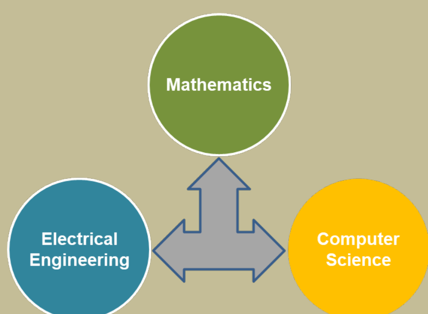


Cryptographic Engineering: Practical Security



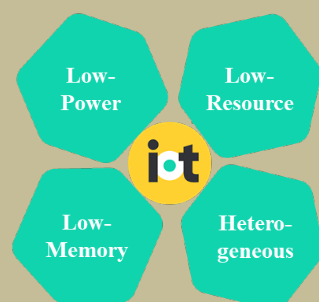
Motivations | Objectives

Implementation Efficiency



Low-lantency

Real-world Constraints



Time-memory trade-off

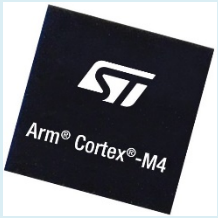
Implementation Security

Side-channel secure

1.3.1 Optimizations Overview

- **Memory optimizations & Side-channel protection**
  - **Secret-related operations**
    - Constant-time implementation
    - Masking technique
  - **Matrix-Vector Product**
    - Generation of the public matrix
  - **Protocols**
    - Other main steps in LBC protocols
- **Polynomial Multiplication**
  - **NTT**
  - **INTT**
  - **Multi-moduli NTT**
    - Butterfly units
    - Layer merging
    - Modular reduction of coefficients
    - Lazy reduction
- **Polynomial Sampling**
  - **Keccak Permutation**
    - Lazy rotation
    - Memory access
- **Modular Arithmetic**
  - **Modular multiplication by a constant**
    - Montgomery arithmetic
    - Barrett reduction
  - **Modular multiplication/reduction**
    - Plantard arithmetic
    - Specialized reduction

IoT Devices



1.3.2 Contributions

The contributions of this thesis are summarized as follows:

1. **Improved Plantard arithmetic tailored for LBC.**
   - Two improvements for Plantard arithmetic tailored for LBC;
   - Two variants of correctness proofs, demonstrating its robustness;
   - Excellent merits over the state-of-the-art modular arithmetic.
2. **Faster Plantard arithmetic, NTT, Keccak and LBC implementations.**
   - Faster Plantard arithmetic implementation on IoT platforms;
   - Optimized 16-bit NTT and multi-moduli NTT implementations with Plantard arithmetic;
   - Optimized Keccak permutation on the 32-bit ARMv7-M (over 20% speedups);
   - State-of-the-art Kyber, NTTRU, and Dilithium implementations on the target platforms.
3. **Efficient, lightweight and side-channel secure Raccoon implementations.**
   - Optimized the multi-moduli NTT of the 32-bit NTTs with Montgomery arithmetic;
   - Time complexity reduction of the masking gadgets;
   - Memory optimizations to enable high-order Raccoon on memory-constrained IoT devices.

Objective Achieved

Mathematical Improvement & Efficiency

Implementation Efficiency & Security

Implementation Efficiency & Security & Lightweight

02 Improved Plantard Arithmetic

2.1.1 State-of-the-art Modular Arithmetic

❑ State-of-the-art modular arithmetic, i.e., $a \times b \ \% \ q$

Both Montgomery (1985) and Barrett multiplication (1986) for l-bit modulus ($l = 16$ or 32):

- need **3** multiplications;
- use the product $c = a \times b$ **twice**;
- support **signed inputs** in a large domain, which enable a lazy reduction strategy

Algorithm 1 Signed Montgomery multiplication

Input: Constant $\beta = 2^l$ where l is the machine word size, odd q such that $0 < q < \frac{\beta}{2}$, and operand a, b such that $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$

Output: $r \equiv ab\beta^{-1} \bmod q, r \in (-q, q)$

1: $c = c_1\beta + c_0 = a \cdot b$
2: $m = c_0 \cdot q^{-1} \bmod^{\pm} \beta$
3: $r = c_1 - \lfloor m \cdot q/\beta \rfloor$
4: return r

Algorithm 2 Barrett multiplication

Input: Operand a, b such that $0 \leq a \cdot b < 2^{2l'+\gamma}$, the modulus q satisfying $2^{l'-1} < q < 2^{l'}$, and the precomputed constant $\lambda = \lfloor 2^{2l'+\gamma}/q \rfloor$

Output: $r \equiv a \cdot b \bmod q, r \in [0, 3q]$

1: $c = a \cdot b$
2: $t = \lfloor (c \cdot \lambda)/2^{2l'+\gamma} \rfloor$
3: $r = c - t \cdot q$
4: return r

2.1.2 Original Plantard Arithmetic

❑ Plantard's seminal word-size modular arithmetic

Algorithm 15 Original Plantard multiplication [92]

Input: Unsigned integers $a, b \in [0, q]$, $q < \frac{2^l}{\phi}$, $\phi = \frac{1+\sqrt{5}}{2}$, $q' \equiv q^{-1} \bmod 2^{2l}$, where l is the machine word size

Output: $r \equiv ab(-2^{-2l}) \bmod q$ where $r \in [0, q]$

1: $r = \left[\left([[abq']_{2l}]^l + 1\right) q\right]^l$ ▷ bq' could be precomputed when b is constant

2: **return** r

// $[a]_l \leftarrow a \bmod 2^l, [a]^l \leftarrow a \gg l,$

Plantard multiplication:

Pros:

- When one of the **operands (*b*) is fixed,** it is **one multiplication fewer than Montgomery arithmetic**. **(Suitable for NTT computation!)**

Cons:

- Introduces an $l \times 2l$-bit multiplication bq'. **(Only suitable on specific platforms)**
- only supports unsigned integers in a **small domain** $[0, q]$. **(How to support signed integers in a larger input range?)**

2.2 Improved Plantard Arithmetic

❑ Improved Plantard arithmetic (TCHES2022)

- **Tailored for LBC word size moduli:** proposed a new modulus restriction $q < 2^{l-\alpha-1}$ by introducing a small integer $\alpha > 0$; provided **two versions of correctness proof.**
  - Following the proof of the original Plantard arithmetic paper.
  - The CRT interpretation from Prof. Guangwu Xu[2].
- **Larger input range:** from unsigned integers $[0, q]$ to **signed integers in $[-q2^{\alpha}, q2^{\alpha}]$;**
- **Smaller output range:** from $[\mathbf{0}, \boldsymbol{q}]$ signed integer in $[-\frac{q+1}{2}, \frac{q}{2})$;
- **Inherent advantage:** when b is a constant, it can save **one multiplication** by precomputing bq' mod 2^{2l}.

Algorithm 16 Improved Plantard multiplication

Input: Two signed integers $a, b \in [-q2^{\alpha}, q2^{\alpha}], q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $r = \left[\left(\left[[abq']_{2l}\right]^{l} + 2^{\alpha}\right) q\right]^{l}$

2: **return** r

[1] **Junhao Huang**, Jipeng Zhang, et al*. Improved Plantard Arithmetic for Lattice-based Cryptography[J]. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES),* 2022, 2022(4): 614-636.

[2] Yanze Yang, Yiran Jia, and Guangwu Xu. On modular algorithms and butterfly operations in number theoretic transform. Cryptology ePrint Archive,2024.

2.3 Further Improvement of Plantard Arithmetic

❑ Plantard arithmetic with larger input range (TIFS2024)

- The improved Plantard multiplication supports signed inputs in $[-q2^{\alpha}, q2^{\alpha}] \in (-2^{l-1}, 2^{l-1})$, i.e., the product of $ab \in (-2^{2l-2}, 2^{2l-2})$.
- Further extend the input range to $ab \in [q2^{l} - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$. (refer the **correctness proof** to the thesis)
- For Kyber, when b is a constant, the previous range of $a \in [-64q, 64q]$. After the improvement, the range of a is increased up to $a \in [-137q, 230q]$, **2.14 × larger.**

Algorithm 17 Plantard multiplication with enlarged input range

Input: Two signed integers a, b such that $ab \in [q2^{l} - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha}), q < 2^{l-\alpha-1}, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$ where $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $r = \left[\left([[abq']_{2l}]^{l} + 2^{\alpha}\right) q\right]^{l}$

2: **return** r

$$a_{max} < (2^{2l} - q2^{l+\alpha})/b_{max}$$

$$= (2^{32} - 3329 \times 2^{19})/3328 \approx 230.13q.$$

$$a_{min} > (q2^{l} - q2^{l+\alpha})/b_{max}$$

$$= (3329 \times 2^{16} - 3329 \times 2^{19})/3328 \approx -137.85q.$$

[1] **Junhao Huang**, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Cetin Kaya Koc, Donglong Chen*. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices[J]. *IEEE Transactions on Information Forensics & Security (TIFS),* 2024.

2.4 Another Variant of Plantard Arithmetic

❑ Another Variant of signed Plantard arithmetic

- Daichi et al.[2] concurrently proposed another variant of signed Plantard arithmetic in 2022.
- The **rounding-to-nearest operations** in their version are not architecture-friendly in most platforms.
- In one of the coauthored work[1], we manage to **replace one rounding-to-nearest with one flooring operation**, reducing one rounding-to-nearest operation.

Algorithm 18 Signed Plantard multiplication [15]

Input: Two signed integers a, b with $|a|, |b| \leq 2^{l-1}$, the odd modulus $q < 2^{l-1}$ and

$q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in [-\frac{q-1}{2}, \frac{q-1}{2}]$

1: $r = abq' \bmod^{\pm} 2^{2l}$

2: $r = \lfloor r/\beta \rceil$

3: $r = \lfloor rq/\beta \rceil$

4: **return** r

Source code 1
SIGNED PLANTARD MULTIPLICATION

```
int64_t signedPlantardMul(int64_t A, int64_t B) {
   return (((A*B + 0x80000000)>>32)*P + 0x80000000)>>32;
}
```

[1] Jipeng Zhang, Yuxing Yan, **Junhao Huang**, and Cetin Kaya Koc. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES),* 2025(1), 2025.

[2] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi.Efficient Word Size Modular Multiplication over Signed Integers. In 29th IEEESymposium on Computer Arithmetic, ARITH 2022, Lyon, France, September12-14, 2022, pages 94–101. IEEE, 2022.

2.5 Comparisons

❑ **Excellent merits over the state-of-the-art**

- **Efficiency:** Plantard multiplication is **one multiplication faster** than the state-of-the-art Montgomery and Barrett multiplication when ***b* is a constant**.
- **Input range**: $[q2^{l} - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$ vs $[-q2^{l-1}, q2^{l-1}]$ for $\alpha \geq 0$, at least $2^{\alpha+1}$ times bigger than Montgomery's;
- **Output range:** $[-\frac{q+1}{2}, \frac{q}{2})$ vs $(-q, q)$, only **half** of the Montgomery's

Algorithm 7 Signed Montgomery multiplication [Sei18]

Input: Operand a, b such that $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $\beta = 2^l$ with the machine word size l, the odd modulus $q \in (0, \frac{\beta}{2})$

Output: $r \equiv ab\beta^{-1} \bmod q, r \in (-q, q)$

1: $c = c_1\beta + c_0 = a \cdot b$
2: $m = c_0 \cdot q^{-1} \bmod^{\pm} \beta$ ▷ $\bmod^{\pm}$ obtains a signed product, q^{-1} is a precomputed constant
3: $t_1 = \lfloor m \cdot q/\beta \rfloor$ ▷ shift right operation
4: $r = c_1 - t_1$
5: **return** r

With all these merits, how to efficiently turn the theoretical improvement into actual improvements is the remaining question.

03 Efficient LBC on IoT Devices

3.1.1 Target Schemes

- **Kyber**
  - **The only KEM scheme to be standardized.**
  - Module-LWE problem ($\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{e}$).
  - Parameters: $n = 256, \boldsymbol{q = 3329 < 2^{12}}, k = 2,3,4, Z_{3329}[X]/(X^{256} + 1)$.
- **NTTRU**
  - An **NTT-friendly** variant of NTRU KEM scheme proposed in TCHES2019.
  - The KeyGen, Encaps and Decaps are $30 \times, 5 \times$, **and** $8 \times$ faster than the respective procedures in the NTRU schemes.
  - Parameters: $\boldsymbol{n = 768, q = 7681}, Z_{7681}[X] /(X^{768} - X^{384} + 1)$.
- **Dilithium**
  - **One out of three final DSA to be standardized.**
  - Module-LWE problem and Module-SIS problem.
  - Parameters: $\boldsymbol{n = 256, q = 8380417 < 2^{23}}, Z_{8380417}[X] /(X^{256} + 1)$.

3.1.2 Target Platforms

❑ ARM Cortex-M4: Relative high power, resource and memory IoT platform

- NIST's reference 32-bit platform for evaluating PQC in IoT scenarios (a popular **pqm4** repository: https://github.com/mupq/pqm4);
- **1MB flash, 192KB RAM;**
- **14** 32-bit usable general-purpose registers, **32** 32-bit floating-point registers;
- SIMD (DSP) extensions: **uadd16, usub16** instructions perform addition and subtraction for two packed 16-bit vectors;
- **1-cycle** multiplication instructions: **smulw{b,t}, smul{b,t}{b,t};**
- Relative expensive **load/store** instructions: **ldr, ldrd, vldm.**
- To utilize the efficient SIMD instructions on Cortex-M4, the size of the coefficients is limited to **16-bit signed integer.**

3.1.2 Target Platforms

❑ **ARM Cortex-M3: Low resource IoT platform**

- **14** 32-bit usable general-purpose registers, **no** floating-point registers;
- **Non-constant time** full multiplication instructions: **umull, smull, umlal** and **small**; No **SIMD extensions** and limited multiplication instructions: **mul, mla (1, 2 cycles).**
- Inline barrel shifter operation, e.g., **add rd, rn, rm, asr #16,** which can **merge the addition and shifting operations in 1 instruction.**
- **512KB flash, 96KB RAM;**

❑ **SiFive Freedom RISC-V: Extremely low resource and memory IoT platform**

- **Open-source ISA;**
- **Only 16KB RAM;**
- **30** 32-bit usable general-purpose registers, **no** floating-point registers;
- No **SIMD extensions** and limited multiplication instructions: **mul, mulh (5-cycle)**;

3.1.3 Polynomial multiplications

❑ 16-bit NTT

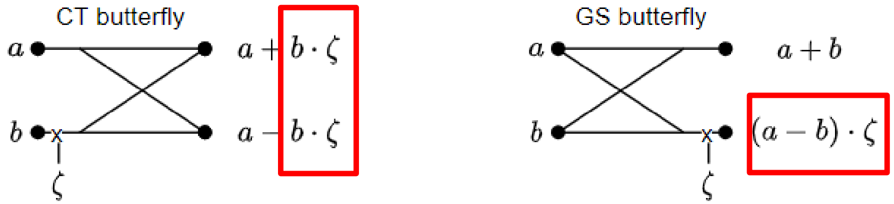
- Both Kyber and NTTRU use **16-bit NTT** for polynomial multiplication.
- The polynomial ring $Z_q[X]/f(X)$ implemented with NTT factors the large-degree polynomial f(X) as
  - $f(x) = \prod_{i=0}^{n'-1} f_i(x) \bmod q,$

where $f_i(X)$ are small degree polynomials like $(X^2 - r)$ and $(X^3 \pm r)$ for Kyber and NTTRU, respectively.

❑ 32-bit NTT

- Dilithium normally uses **32-bit NTT** for polynomial multiplication.
- The polynomial ring $Z_q[X]/f(X)$ of Dilithium implemented with 32-bit NTT factors the large-degree polynomial f(X) as
  - $f(x) = \prod_{i=0}^{n'-1} f_i(x) \bmod q,$

where $f_i(X)$ are small degree polynomials like $(X - r)$ for Dilithium.



Modular multiplication with the twiddle factors can be speeded up with Plantard arithmetic.

3.2.1 Faster 16-bit Plantard Arithmetic on Cortex-M4

❑ Faster Plantard multiplication by a constant on Cortex-M4

- The Plantard multiplication by a constant **(b is a constant) saves one multiplication bq' by precomputing $bq' \bmod^{\pm} 2^{2l}$.**
- The 16×32-bit multiplication abq' is then implemented with **smulwb** instruction. The rest of the computations can be simply implemented with **one smlabb instruction.**
- The Plantard multiplication by a constant on Cortex-M4 is **1-instruction faster** than the state-of-the-art Montgomery's.

Algorithm 19 The 2-cycle improved Plantard multiplication by a constant on Cortex-M4

Input: An l-bit signed integer $a \in [-2^{l-1}, 2^{l-1})$, a precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r_{top} = ab(-2^{-2l}) \bmod^{\pm} q, r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed

2: `smulwb` r, bq', a ▷ $r \leftarrow [[abq']_{2l}]^l$

3: `smlabb` $r, r, q, q2^{\alpha}$ ▷ $r_{top} \leftarrow [q[r]_l + q2^{\alpha}]^l$

4: **return** r_{top}

3.2.2 Faster 16-bit Plantard Arithmetic on Cortex-M3 and RISC-V

❑ Faster Plantard multiplication by a constant on Cortex-M3 and RISC-V

- **Cortex-M3:** We can merge the **addition and shift operation** using the barrel shifter operation as in Step 3 of Algorithm 4.
- **RISC-V:** We can use **muh** with $q2^l$ to merge the **mul and asr** operation in the final two steps of Algorithm 4.
- Both implementations are **1-multiplication faster** than the Montgomery's.

Algorithm 4 Efficient Plantard multiplication by a constant for Kyber on Cortex-M3

Input: An 32-bit signed integer $a \in [-157q, 230q]$, a pre-computed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod^{\pm} 2^{32}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod 2^{2l}$ ▷ precomputed
2: **mul** r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
3: **add** $r, 2^{\alpha}, r$, asr#16 ▷ $r \leftarrow ([r]^l + 2^{\alpha})$
4: **mul** r, r, q
5: **asr** $r, r, \#16$ ▷ $r \leftarrow [rq]^l$
6: **return** r

Algorithm 5 Efficient Plantard multiplication by a constant for Kyber on RISC-V

Input: An 32-bit signed integer $a \in [-157q, 230q]$, a pre-computed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{32}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **mul** r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
3: **srai** $r, r, \#16$
4: **addi** $r, r, 2^{\alpha}$ ▷ $r \leftarrow ([r]^l + 2^{\alpha})$
5: **mulh** $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$
6: **return** r

3.2.3 Faster 16-bit/32-bit Plantard Arithmetic on Other Platforms

❑ Faster 32-bit Plantard multiplication by a constant on 64-bit RISC-V

➢ The 32-bit Plantard arithmetic can be extended to 64-bit RISC-V. The instruction sequences are the same as the 16-bit Plantard arithmetic on 32-bit RISC-V [1,2].

❑ Faster 16-bit Plantard multiplication by a constant on customized RISC-V

➢ Customized SIMD instruction (**asravi**) for Plantard arithmetic. **Two instructions** faster than the Montgomery arithmetic on the same platform [3].

Algorithm 24 Efficient Plantard multiplication by a constant for Dilithium on RV64IM

Input: An 64-bit signed integer $a \in [-130686q, 131457q]$, a precomputed 64-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{64}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

$bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
`mul` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
`srai` $r, r, \#32$
`addi` $r, r, \#256$ ▷ $r \leftarrow ([r]^l + 2^\alpha)$
`mulh` $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$
return r

Algorithm 25 Efficient Plantard multiplication by a constant for NTRU and Hawk on customized RISC-V SIMD ISA

Input: An 32-bit signed integer a, a precomputed 32-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{32}, q2^l = q \times 2^l$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q$

$bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
`mulv` r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
`asravi` $r, r, 2^\alpha, l$ ▷ $r \leftarrow ([r]^l + 2^\alpha)$
`mulvh` $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$
return r

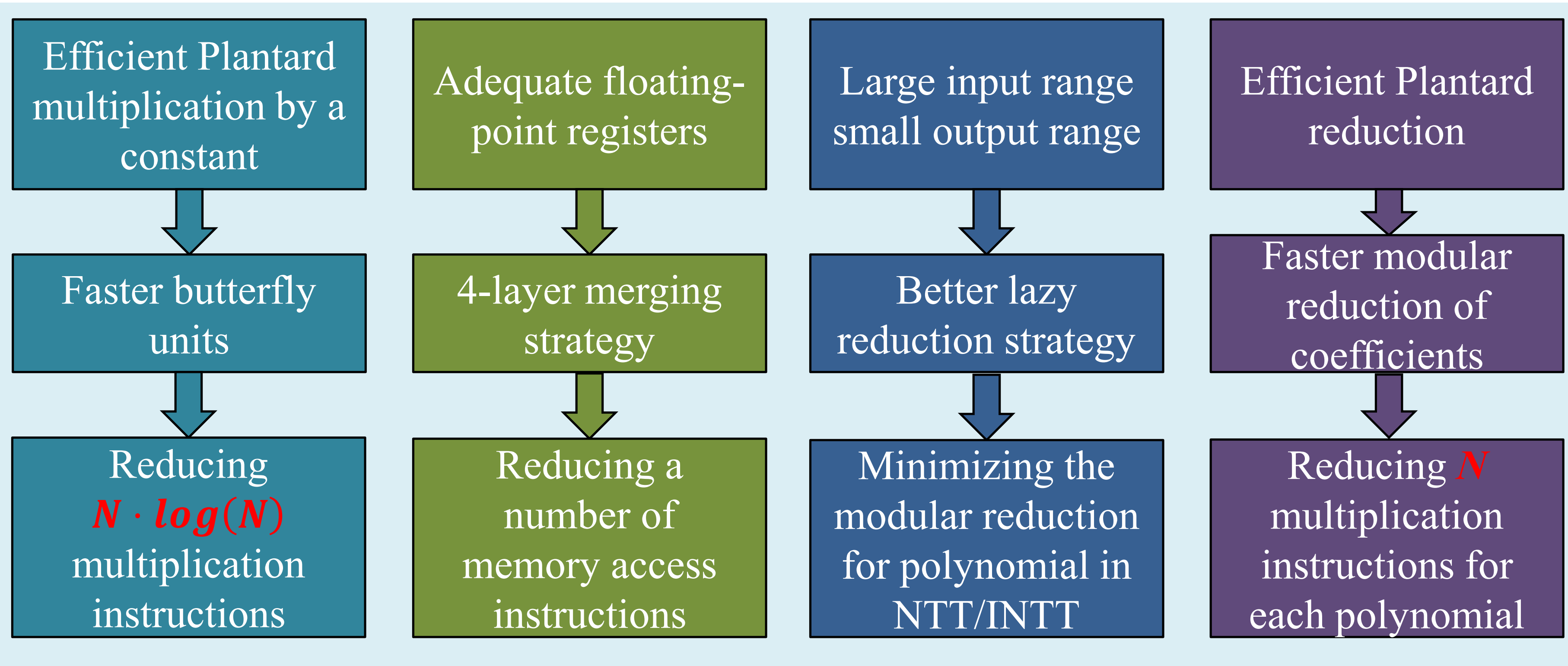
[1] Jipeng Zhang, Yuxing Yan, **Junhao Huang**, and Cetin Kaya Koc. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES),* 2025(1), 2025.

[2] Xinyi Ji, Jiankuo Dong, **Junhao Huang**, Zhijian Yuan, Wangchen Dai, Fu Xiao,and Jingqiang Lin. Eco-crystals: Efficient cryptography crystals on standard risc-v isa. IEEE Transactions on Computers, pages 1–13, 2024.

[3] Zewen Ye, **Junhao Huang**, Tianshun Huang, Yudan Bai, Jinze Li, Hao Zhang,Guangyan Li, Donglong Chen, Ray C. C. Cheung, and Kejie Huang. PQN-TRU: acceleration of ntru-based schemes via customized post-quantum processor. IEEE Transactions on Computers, 2025.

3.3.1 Optimized 16-bit NTT Implementation

❑ Optimizations summary



The **proposed improved Plantard arithmetic** make it possible to replace previous state-of-the-art Montgomery arithmetic in the NTT implementation on **Cortex-M4, Cortex-M3, RISC-V and etc**, further improving the performance of LBC.

3.3.2 The 16-bit NTT Results

❑ The 16-bit NTT results on Cortex-M4

Table 4.2: Cycle counts for the core polynomial arithmetic in Kyber and NTTRU on Cortex-M4, i.e., NTT, INTT, base multiplication, and base inversion.

Scheme	Implementation	NTT	INTT	Base Mult	Base Inv
Kyber	[12]	6 822	6 951	2 291	-
	This work[a]	5 441	5 775	2 421	-
	Speed-up	20.24%	16.92%	-5.67%	-
	Stack[3]	5 967	5 917	2 293	-
	Speed[3]	5 967	5 471	1 202	-
	This work[b]	4 474	4 684/4 819/4 854	2 422	-
	Speed-up (stack)	25.02%	20.84%/18.56%/17.97%	-5.58%	-
	Speed-up (speed)	25.02%	14.38%/11.92%/11.28%	-101.41%	-
NTTRU	[79]	102 881	97 986	44 703	100 249
	This work	17 274	20 931	10 550	40 763
	Speed-up	83.21%	78.64%	76.40%	59.34%

[a] Implementation based on [12], [b] Implementation based on the stack-friendly code of [3].

3.3.2 The 16-bit NTT Results

❑ The 16-bit NTT results on Cortex-M3 and RISC-V

Table 4.3: Cycle counts for the core polynomial arithmetic in Kyber, namely NTT, INTT and base multiplication, in Kyber on Cortex-M3, SiFive Freedom E310, and PQRISCV.

Platform	Implementation	NTT	INTT	Base Multiplication
Cortex-M3	Denisa et al. [55]	10 874	13 049	4 821
	This work (stack)	8 026	8 594/8 799	4 311
	This work (speed)	8 026	8 594	3 028/3 922/5 851
	Speedup (stack)	26.19%	34.14%/32.57%	1.06%
	Speedup (speed)	26.19%	34.14%	37.19%/18.65%/-21.37%
SiFive Freedom E310	Denisa et al. [54]	24 353	36 513	-[a]
	This work (stack)	15 888	15 719/16 227	10 020
	This work (speed)	15 888	15 719	4 893/5 662/9 313
	Speedup (stack)	34.76%	56.95%/55.53%	-
	Speedup (speed)	34.76%	56.95%	-
PQRISCV	Denisa et al. [54]	28 417	42 636	-[a]
	This work (stack)	21 975	23 666/24 146	12 236
	This work (speed)	21 975	23 666	7 747/9 795/13 068
	Speedup (stack)	22.67%	44.49%/43.37%	-
	Speedup (speed)	22.67%	44.49%	-

a. [54] did not provide results for base multiplication.

3.4.1 16-bit NTT vs 32-bit NTT

❑ 16-bit NTT vs 32-bit NTT on Cortex-M3

- **Cortex-M3** does not have **constant-time full multiplication**, which may lead to insecure 32-bit modular multiplication implementation (side-channel attack).
- The constant-time 32-bit modular multiplication takes **6-8 instructions**.
- The constant-time 32-bit CT butterfly takes **19 instructions, compared to 5 instructions for 16-bit CT butterfly;**
- **The 16-bit NTT is at least 2~3 × faster than 32-bit NTT on Cortex-M3 [1].**

				NTT	NTT^{-1}	$\circ$
Dilithium[a]	[GKOS18]	constant-time	M4	10 701	11 662	–
	This work	constant-time	M4	8 540	8 923	1 955
	This work	variable-time	M3	19 347	21 006	4 899
	This work	constant-time	M3	33 025	36 609	8 479
Kyber[b]	[ABCG20]	constant-time	M4	6 855	6 983	2 325
	This work	constant-time	M3	10 819	12 994	4 773
NewHope1024[c]	[ABCG20]	constant-time	M4	68 131	51 231	6 229
	This work	constant-time	M3	77 001	93 128	18 722

[a] $n = 256, q = 8380417$ (23 bits), 8 layer $\text{NTT}/\text{NTT}^{-1}$
[b] $n = 256, q = 3329$ (12 bits), 7 layer $\text{NTT}/\text{NTT}^{-1}$
[c] $n = 1024, q = 12289$ (14 bits), 10 layer $\text{NTT}/\text{NTT}^{-1}$

[1] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Com-pact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans.Cryptogr. Hardw. Embed. Syst., 2021*(1):1–24, 2021.

3.4.2 Polynomial multiplication of Dilithium

❑ Small polynomial multiplications: cs_i, ct_i

- In Dilithium signature generation and verification, there exists a **small polynomial c** with at most **τ nonzero coefficients (± 1)** and **the rest of coefficients are 0**.
- The coefficient range of s_i is $[-\eta, \eta]$, then the coefficients of the product cs_i are smaller than **$\beta = \tau \cdot \eta$ (smaller than 16-bit)**.
- The coefficient range of t_i is smaller than 2^{12} or 2^{10}, then the coefficients of the product ct_i are smaller than **$\beta' = \tau \cdot 2^{12}$ or $\beta' = \tau \cdot 2^{10}$ (bigger than 16-bit).**
- According to [CHK+21, Section 2.4.6], these kinds of polynomial multiplications can be treated as multiplications over $Z_{q'}[X]/(X^n + 1)$ with a **well-selected modulus** $q' > 2\beta$ or $q' > 2\beta'$. In sum, we can use **16-bit NTT for cs_i and 32-bit NTT for ct_i.**

Table 1: Dilithium parameters [DKL+18]

NIST security level	2	3	5
q [modulus]	8380417	8380417	8380417
n [the order of polynomial]	256	256	256
d [drop bits from $\mathbf{t}$]	13	13	13
τ [# of ± 1's in c]	39	49	60
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k,l) [dimensions of $\mathbf{A}$]	(4,4)	(6,5)	(8,7)
η [secret key range]	2	4	2
$\beta = \tau \cdot \eta$ [$c\mathbf{s}_i$ coefficient range]	78	196	120
$\mathbf{t}_0$ coefficient range	2^{12}	2^{12}	2^{12}
$\mathbf{t}_1$ coefficient range	2^{10}	2^{10}	2^{10}

3.4.3 Proposed cs_i, ct_i Implementations on Cortex-M3

❑ 16-bit NTT over 769 for cs_i

- The coefficient range of s_i is $[-\eta, \eta]$, then the coefficients of the product cs_i are smaller than **$\beta = \tau \cdot \eta$ =78, 196 and 120 for three security levels.** [AHKS22] used **FNT over 257** for Dilithium2 and Dilithium5, and used **NTT over 769** for Dilithium3.
- **On Cortex-M3:** We optimize the **16-bit NTT over 769 with Plantard arithmetic** for all Dilithium variants, because we can then combine it with **multi-moduli NTT.**

❑ Multi-moduli NTT with two 16-bit NTTs for ct_i

- The coefficient range of t_i is 2^{12} or 2^{10}, then the coefficients of the product ct_i are smaller than **$\beta' = \tau \cdot 2^{12} = 245760, q' > 2\beta' = 491520$**. We choose a composite modulus $q' = 769 \times 3329 = 2560001$ and perform NTT computations over $Z_{q'}[X]/(X^n + 1)$.
- **On Cortex-M3:** We optimize ct_i with the **multi-moduli NTT over the $q' = 769 \times 3329$** for all three Dilithium variants and **separately optimize the 16-bit NTT over 769 and 3329 with Plantard arithmetic.**

$$\mathbb{Z}_{q_0 q_1} \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1};$$
$$\mathbb{Z}_{q_0}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_0}[X]/(X^2 - \zeta_0^j), j = 1, 3, 5, \ldots, 255;$$
$$\mathbb{Z}_{q_1}[X]/(X^{256}+1) \cong \mathbb{Z}_{q_1}[X]/(X^2 - \zeta_1^j), j = 1, 3, 5, \ldots, 255;$$

3.4.4 Multi-moduli NTTs for ct_i

❑ Multi-moduli NTTs for ct_i on Cortex-M3

Algorithm 4 Multi-moduli NTT for computing 32-bit NTT on Cortex-M3

Input: Declare arrays: `int32_t c_32[256],t_32[256],tmp_32[256],res_32[256]`

Input: Declare pointers:

```
int16_t *cl_16=(int16_t*)c_32;
int16_t *ch_16=(int16_t*)&c_32[128];
int16_t *tl_16=(int16_t*)t_32;
int16_t *th_16=(int16_t*)&t_32[128];
int16_t *tmpl_16=(int16_t*)tmp_32;
int16_t *tmph_16=(int16_t*)&tmp_32[128];
```

1: `cl_16[256]` $\leftarrow c$, `ch_16[256]` $\leftarrow c$ ▷ Pre-store c in the bottom and top halves of `c_32` as 16-bit arrays
2: `tl_16[256]` $\leftarrow t$, `th_16[256]` $\leftarrow t$ ▷ Pre-store t in the bottom and top halves of `t_32` as 16-bit arrays
3: `cl_16[256]` $= \mathrm{NTT}_{q_0}($`cl_16`$)$ ▷ $\hat{c}_0 = \mathrm{NTT}_{q_0}(c)$
4: `ch_16[256]` $= \mathrm{NTT}_{q_1}($`ch_16`$)$ ▷ $\hat{c}_1 = \mathrm{NTT}_{q_1}(c)$
5: `tl_16[256]` $= \mathrm{NTT}_{q_0}($`tl_16`$)$ ▷ $\hat{t}_0 = \mathrm{NTT}_{q_0}(t)$
6: `th_16[256]` $= \mathrm{NTT}_{q_1}($`th_16`$)$ ▷ $\hat{t}_1 = \mathrm{NTT}_{q_1}(t)$
7: `tmpl_16[256]` $= \mathrm{basemul}_{q_0}($`cl_16, tl_16`$)$ ▷ $\hat{c}_0 \cdot \hat{t}_0 = \mathrm{basemul}_{q_0}(\hat{c}_0, \hat{t}_0)$
8: `tmph_16[256]` $= \mathrm{basemul}_{q_1}($`ch_16, th_16`$)$ ▷ $\hat{c}_1 \cdot \hat{t}_1 = \mathrm{basemul}_{q_1}(\hat{c}_1, \hat{t}_1)$
9: `tmpl_16[256]` $= \mathrm{INTT}_{q_0}($`tmpl_16`$)$ ▷ $\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0)$
10: `tmph_16[256]` $= \mathrm{INTT}_{q_1}($`tmph_16`$)$ ▷ $\mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1)$
11: `res_32[256]` $= \mathrm{CRT}($`tmpl_16, tmph_16`$)$ ▷ $\mathrm{CRT}(\mathrm{INTT}_{q_0}(\hat{c}_0 \cdot \hat{t}_0), \mathrm{INTT}_{q_1}(\hat{c}_1 \cdot \hat{t}_1))$
12: **return** `res_32`

3.4.5 Dilithium's NTT Results

❑ The 16-bit NTT and multi-moduli NTT results on Cortex-M3

- Using the Plantard arithmetic, the **16-bit NTT, INTT, and pointwise multiplication** on Cortex-M3 are **4.22×, 4.29×, and 2.14× faster** than the constant-time 32-bit NTT, INTT, and pointwise multiplication, respectively. Compared to the 32-bit variable-time NTT, INTT, and pointwise multiplication, the speed ups are **2.48×, 2.46×, and 1.24×,** respectively.
- The **proposed multi-moduli NTT, INTT and pointwise multiplication** implementations yield **52.76% ~ 54.76%** performance improvements compared to the constant-time 32-bit NTT. And over **19.47% and 19.07% speed-ups** compared with the variable-time 32-bit NTT and INTT.

Platform	Prime	Ref.	NTT	INTT	Pointwise	CRT
M3	8380417	[GKS20] constant-time	33 077	36 661	8 528	✗
	8380417	[GKS20] variable-time	19 405	21 051	4 944	✗
	3329×7681	[ACC+22]	16 770	19 056	11 927	4 637
	769	This work	7 830	8 543	3 989	✗
	769×3329	This work	15 626	17 037	8 061	3 735

3.5.1 Efficient Polynomial Sampling: Keccak

❑ Pipelining memory access

```
.macro xor5    result,b,g,k,m,s
    ldr        \result, [r0, #\b]
    ldr        r1, [r0, #\g]
    eors       \result, \result, r1
    ldr        r1, [r0, #\k]
    eors       \result, \result, r1
    ldr        r1, [r0, #\m]
    eors       \result, \result, r1
    ldr        r1, [r0, #\s]
    eors       \result, \result, r1
.endm
```

```
.macro xor5    result,b,g,k,m,s
    ldr        \result, [r0, #\b]
    ldr        r1,  [r0, #\g]
    ldr        r5,  [r0, #\k]
    ldr        r11, [r0, #\m]
    ldr        r12, [r0, #\s]
    eors       \result, \result, r1
    eors       \result, \result, r5
    eors       \result, \result, r11
    eors       \result, \result, r12
.endm
```

❑ Lazy rotations

- Utilize the inline barrel shifter instruction on ARMv7-M to **merge the xor and ror instructions,** which could help to reduce some cycles.
- We proposed **two variants of Keccak implementation** considering the code size effect. One has better performance but requiring larger code size. And one has smaller code size and an acceptable performance.

3.5.2 Keccak Results

❑ Keccak results on Cortex-M3 and M4

➢ Combining the **pipelining memory access** and **lazy rotations** techniques, we achieve up to **24.78% and 21.4%** performance boosts on Cortex-M3 and M4, respectively

Table 4.1: Keccak-p[1600, 24] benchmark on Cortex-M3 and M4.

Ref.	Implementation characteristics*		Speed (clock cycles)		Code size	RAM
	ldr/str	lazy ror	M3	M4	(bytes)	(bytes)
XKCP	mostly grouped	✗	13 015	11 725	5 576	**264**
This work	grouped	✗	10 785	10 219	5 772	**264**
	grouped	✓ (3/4)	9 981	9 415	6 556	**264**
	grouped	✓ (4/4)	**9 789**	**9 218**	9 536	**264**

*All listed implementations take advantage of the in-place processing and bit-interleaving techniques.

3.6 LBC Results: Kyber and NTTRU

❑ Kyber and NTTRU results on Cortex-M4 without Keccak optimization

3%

55%

Scheme	Implementation	KeyGen			Encaps			Decaps		
		$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$	$k=2$	$k=3$	$k=4$
Kyber	[12]	454k 2 464	741k 2 696	1 177k 3 584	548k 2 168	893k 2 640	1 367k 3 208	506k 2 184	832k 2 656	1 287k 3 224
	This work[a]	446k 2 464	729k 2 696	1 162k 3 584	542k 2 168	885k 2 640	1 357k 3 208	497k 2 184	818k 2 656	1 270k 3 224
	Stack[3]	439k 2 608	717k 3 056	1 139k 3 576	534k 2 160	871k 2 660	1 329k 3 236	484k 2 176	797k 2 676	1 233k 3 252
	Speed[3]	438k 4 268	711k 6 732	1 129k 7 748	531k 5 252	864k 6 284	1 316k 7 292	479k 5 260	787k 6 308	1 217k 7 300
	This work[b]	430k 2 608	702k 3 056	1 119k 3 576	526k 2 160	861k 2 660	1 314k 3 236	472k 2 176	780k 2 676	1 211k 3 252
NTTRU	[79]	526k 9 384			431k 8 748			559k 10 324		
	This work	267k 9 372			237k 7 452			254k 8 816		

[a] Implementation based on [12], [b] Implementation based on the stack-friendly code of [3].

3.6 LBC Results: Kyber

❑ **Kyber results on Cortex-M3 and RISC-V without Keccak optimization**

	Platform	Implementation	Kyber512			Kyber768			Kyber1024		
			KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
		Denisa et al.[55]	541k	650k	622k	878k	1 054k	1 010k	1 388k	1 602k	1 543k
			2 212	2 300	2 308	3 084	2 772	2 788	3 596	3 284	3 300
5%	Cortex-M3	This work (stack)	519k	628k	590k	844k	1 025k	967k	1 342k	1 563k	1 486k
			2 212	2 300	2 308	3 084	2 772	2 788	3 596	3 284	3 300
		This work (speed)	518k	626k	587k	842k	1 017k	958k	1 333k	1 548k	1 471k
			3 268	3 860	3 860	4 044	4 636	4 636	4 812	5 404	5 404
		Denisa et al.[54]	2 229k	2 927k	2 856k	4 166k	5 071k	4 957k	6 696k	7 809k	7 662k
			6 544	9 200	9 984	10 640	13 808	14 944	15 760	19 440	21 056
30%	PQRISCV	This work (stack)	1 937k	2 355k	2 100k	3 147k	3 822k	3 467k	4 964k	5 794k	5 344k
			2 408	2 488	2 520	2 952	3 016	3 032	3 464	3 528	3 544
		This work (speed)	1 926k	2 339k	2 084k	3 104k	3 768k	3 413k	4 890k	5 704k	5 254k
			3 432	4 024	4 040	4 216	4 808	4 840	5 032	5 608	5 656
31%	SiFive Freedom E310	This work (stack)	1 497k	1 812k	1 601k	2 413k	2 929k	2 635k	3 794k	4 435k	4 045k
			2 580	2 660	2 708	3 060	3 124	3 156	3 572	3 636	3 668
		This work (speed)	1 597k	1 903k	1 674k	2 731k	3 203k	2 919k	-	-	-
			3 620	4 212	4 244	4 340	4 932	4 964	-	-	-

3.6 LBC Results: Dilithium

❑ Kyber and Dilithium results on Cortex-M3/4 with Keccak optimization

Table 6: PQC benchmark on the Cortex-M4 using the `pqm4` framework. Averaged over 1000 executions.

	Scheme	Keccak Impl.	keygen		sign/encaps		verify/decaps	
			speed	hashing	speed	hashing	speed	hashing
15%	Dilithium2	XKCP	1 595k	83.47%	4 052k	64.53%	1 576k	80.47%
		This work	1 357k	80.57%	3 487k	60.02%	1 350k	77.2%
	Dilithium3	XKCP	2 828k	85.54%	6 523k	62.95%	2 702k	82.62%
		This work	2 394k	82.92%	5 574k	58.97%	2 302k	79.61%
	Dilithium5	XKCP	4 817k	86.6%	8 534k	68.08%	4 714k	84.69%
		This work	4 069k	84.14%	7 730k	63.05%	3 998k	81.95%
15%	Kyber512	XKCP	432k	80.12%	527k	82.86%	472k	73.76%
		This work	369k	76.75%	448k	79.85%	409k	69.74%
	Kyber768	XKCP	704k	79.04%	860k	82.38%	778k	74.75%
		This work	604k	75.59%	732k	79.32%	674k	70.84%
	Kyber1024	XKCP	1 122k	79.58%	1 314k	82.46%	1 208k	76.07%
		This work	962k	76.18%	1 119k	79.41%	1 043k	72.29%

04

Efficient Side-Channel Secure LBC on IoT Devices

4.1.1 Target Scheme: Raccoon

❑ Raccoon – Side-channel secure LBC scheme

- Raccoon: low-complexity masking-friendly **O(d·log d),** side-channel secure LBC scheme.

- Masking gadgets: Complex masking gadgets to secure against side-channel attacks. **(Efficient masking gadgets)**

- Hardness: **Module-LWE and Module-SIS**, similarly to the NIST standard Dilithium.

- Polynomial multiplication: $n = 512, \mathrm{q} = q_1 \cdot q_2 < 2^{49}, q_1 = 2^{24} - 2^{18} + 1, q_2 = 2^{25} - 2^{18} + 1, Z_q[X]/(X^{512} + 1)$. **(Efficient 49-bit NTT implementation)**

- Memory consumption: At high masking orders, **memory consumption** becomes the the major bottleneck for its deployment on IoT devices. **(Lightweight implementation of high-order Raccoon)**

4.1.2 Target Platforms

❑ ARM Cortex-M4: Relative high power, resource and memory IoT platform

- NIST's reference 32-bit platform for evaluating PQC in IoT scenarios (a popular **pqm4** repository: https://github.com/mupq/pqm4);
- **1MB flash, 192KB RAM;**
- **14** 32-bit usable general-purpose registers, **32** 32-bit floating-point registers;
- SIMD (DSP) extensions: **uadd16, usub16** instructions perform addition and subtraction for two packed 16-bit vectors;
- **1-cycle** multiplication instructions: **smulw{b,t}, smul{b,t}{b,t};**
- Relative expensive **load/store** instructions: **ldr, ldrd, vldm.**
- **New instructions involved: smmla, smmls, smlal.**

4.1.3 Polynomial multiplication

❑ 64-bit NTT

- Raccoon use a **64-bit NTT over a composite modulus q** for polynomial multiplication.
- The polynomial ring $Z_q[X]/f(X)$ implemented with NTT factors the large-degree polynomial f(X) as
  - $f(x) = \prod_{i=0}^{n'-1} f_i(x) \bmod q,$

where $f_i(X)$ are small degree polynomials like $(X - r)$.

❑ Multi-moduli NTT of 32-bit NTTs (more friendly on 32-bit IoT platforms)

- Using the CRT theorem, the **64-bit NTT can be split into two 32bit NTT over two 32-bit moduli q_1 and q_2**, which is more friendly on 32-bit platforms. The overall process is as follows:
  - **Polynomial splitting:** Two consecutive modular reductions are required to reduce the 64-bit polynomial coefficients modulo 32-bit q_1 and q_2.
  - **NTT operations:** Two 32-bit NTTs, pointwise multiplications and INTTs over q_1 and q_2.
  - **Reconstruction using CRT:** Combine the 32-bit results modulo q_1 and q_2 into 64-bit results using the CRT theorem.

4.2.1 Optimized Polynomial Multiplication

❑ Polynomial splitting

- Two variants of Montgomery arithmetic: depending on whether **$-q'$ or q'** is used;
- State-of-the-art Montgomery arithmetic (**2-cycle**) on Cortex-M4 use **$-q'$**; Not appropriate for in-place two consecutive modular reductions (Need at least **7 cycles**).
- We used **q'** instead and proposed a **2-instruction** faster negative double Montgomery reductions using the **smmla** instructions. **(Produce the negative of the correct results)**

Algorithm 33 Double modular reductions with original Montgomery reduction

Input: $a = a_0 + a_1 \cdot 2^{32}$

Output: $a \cdot 2^{-32} \bmod^{\pm} q_1, a \cdot 2^{-32} \bmod^{\pm} q_2$

1: `mul` $t_3, a_0, -q'_2$
2: `mov` t_1, a_0
3: `mov` t_2, a_1
4: `smlal` a_0, a_1, t_3, q_2
5: `mul` $t_3, t_1, -q'_1$
6: `smlal` t_1, t_2, t_3, q_1
7: `mov` a_0, t_2
8: **return** a_0, a_1

Algorithm 35 The proposed negative double Montgomery reductions

Input: The 64-bit coefficient $a = a_0 + a_1 \cdot 2^{32}$, moduli $q_1, q_2, q'_1 = q_1^{-1} \bmod 2^{32}, q'_2 = q_2^{-1} \bmod 2^{32}$

Output: $-a \cdot 2^{-32} \bmod^{\pm} q_1, -a \cdot 2^{-32} \bmod^{\pm} q_2$

1: `mul` t_1, a_0, q'_1
2: `mul` t_2, a_0, q'_2
3: `neg` a_1, a_1
4: `smmla` a_0, t_1, q_1, a_1
5: `smmla` a_1, t_2, q_2, a_1
6: **return** a_0, a_1

4.2.1 Optimized Polynomial Multiplication

❑ NTT for negative polynomials

- The proposed negative double Montgomery reductions produce **negative of the correct results.**
- The linearity of NTT computations ensures that **NTT(-x)=-NTT(x).** Therefore, it will not affect the correctness of the NTT computations.

Property 2 (Linearity of NTT [5]). *Let $a, b \in \mathbb{Z}_q$, and let x and y be polynomials in the polynomial ring R_q such that* $\mathrm{NTT}(x) = \hat{x}$ *and* $\mathrm{NTT}(y) = \hat{y}$. *Then, the NTT satisfies:* $\mathrm{NTT}(ax + by) = a\hat{x} + b\hat{y}$.

❑ The optimized 32-bit NTT/INTT implementations

- The **3+3+3 layer merging strategy** is used for the 9-layer NTT in Raccoon.
- **Lazy reduction** is comprehensively used to reduce unnecessary modular reductions. Only INTT with CT butterfly needs **modular reductions for 64 coefficients modulo q_1, q_2.**

4.2.2 Optimized Raccoon Masking Gadgets

❑ Lazy reduction for Raccoon's masking gadgets

- We thoroughly reduce the **conditional additions/subtractions** in Raccoon masking gadgets: **ZeroEncoding, Refresh, AddRepNoise, and Decode.**
- We carefully analyze the **output range of these gadgets** and ensure a correct Raccoon implementation.

Table 5.2: Complexity reduction and output range using the lazy reduction

	# of conditional operation	Output range (absolute value)
ZeroEncoding	$2nd \cdot \log(d)$	$q \cdot \log(d)$
Refresh	$nd + 2nd \cdot \log(d)$	$\|x\| + q \cdot \log(d)$
	$2nd + 4nd \cdot \log(d)$	$\|x_i\| + q_i$
AddRepNoise	$nd \cdot \mathrm{rep}$	$\|x\| + \mathrm{rep} \cdot (2^{u_w} + q \cdot \log(d))$
Decode	$n \cdot (d-1)$	$d \cdot \|x\|$
	$2n \cdot (d-1)$	$d \cdot \|x_i\|$

4.3 Lightweight High-order Raccoon

❑ Streaming the matrix-vector multiplication

- Streaming the matrix A: save **80 KiB, 140 KiB, and 252 KiB** of memory for Raccoon-128, Raccoon-192, and Raccoon-256.
- Streaming the masked vector $[\![r]\!]$: reduce $\mathbf{4(\boldsymbol{l}-1)\boldsymbol{d}}$ **KiB** of memory.
- Other memory reuses: reduce $\mathbf{8\boldsymbol{k} + 4\boldsymbol{l}}$ **KiB** of memory.

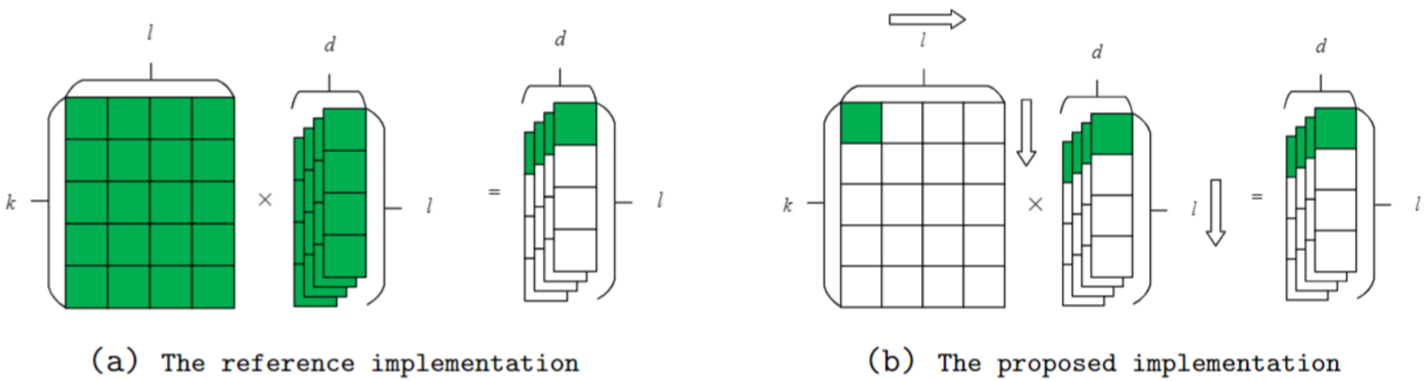


Figure 5.1: The matrix-vector multiplication implementations of $\mathbf{A}\times[\![\mathbf{r}]\!]=[\![\mathbf{w}]\!]$ in the sign of Raccoon

4.3.1 Results and Comparisons

❑ Polynomial arithmetic results on Cortex-M4

- The NTT and INTT are **2.54 × and 3.98 ×** faster than the reference implementation.
- The polynomial left- and right-shift are **3.25 × and 2.93× faster.**

Table 5.3: Cycle counts (cc) of the polynomial arithmetic of Raccoon-128 on Cortex-M4.

	Split	Join	NTT	INTT	Left-shift	Right-shift	Add	Addq[a]
Ref. [94]	9795	22613	118455	171670	12371	14420	7236	10835
This work	6231 (5718)	13908	46677	43026	3801	4942	5970	9047
Ref/This work	1.57 × (1.71×)	1.63×	2.54×	3.98×	3.25×	2.93×	1.21 × (1.81×)	1.20×

[a] Addq denotes the polynomial addition with conditional subtraction of q.

4.3.1 Results and Comparisons

❑ Masking gadgets results on Cortex-M4

- The **lazy reduction strategy** in the masking gadgets results in **1.38 × to 2.61×** speedups, which further improve the performance of Raccoon.

Table 5.4: Cycle counts (cc) of the masking gadgets of Raccoon-128 on Cortex-M4.

		ZeroEncoding	AddRepNoise	Refresh	NTT Refresh	Decode	NTT Decode
$d = 1$	Ref. [94]	3643	4621694	55	53	7228	7228
	This work	3643	2838159	55	53	7227	7228
$d = 2$	Ref. [94]	19377	4785073	40878	68634	10836	14937
	This work	14005	2941116	25776	36666	5972	5714
$d = 4$	Ref. [94]	100749	4909375	144062	199455	32423	44725
	This work	71581	3028599	95438	117095	17827	17059
$d = 8$	Ref. [94]	326040	17286545	4621694	523182	75590	104296
	This work	230879	11084177	278300	321628	41534	39743
$d = 16$	Ref. [94]	900440	17783937	1073626	1294684	161901	223416
	This work	636435	11434030	731780	826092	88926	85086
$d = 32$	Ref. [94]	2297856	73123134	2644266	3086391	334528	461657
	This work	1622473	47134002	1813245	2001820	183711	178759

4.3.1 Results and Comparisons

❑ Raccoon results on Cortex-M4

- ➢ The proposed implementations reduce **32.46%~40.01%** of the clock cycles compared to Raccoon's reference implementation.
- ➢ The proposed memory optimizations **enables the practical use of high-order Raccoon,** namely Raccoon-128 with $d = 16$, Raccoon-192 with $d = 8$, and Raccoon-256 with $d = 4,8$ on the selected platform.

Table 5.5: Cycle counts (cc) and stack usage (Bytes) of keygen, sign, and verify of Raccoon on Cortex-M4. Averaged over 1000 iterations.

	Implementation	Raccoon-128 [a]			Raccoon-192			Raccoon-256		
		keygen	sign	verify	keygen	sign	verify	keygen	sign	verify
$d = 1$	Ref.[94]	29073k	65719k	21851k	45518k	94450k	35862k	73878k	124020k	60837k
		83232	230752	111960	107864	290815	144800	140704	505320	185832
	This work	19637k	39628k	13226k	30044k	56658k	21460k	47631k	79214k	36098k
		82584	230104	111248	107232	332568	144152	140040	504664	185184
$d = 2$	Ref.[94]	35245k	72595k	21851k	53705k	103777k	35858k	85407k	136329k	60839k
		112008	284064	111960	140744	394720	144800	181660	583208	185832
	This work	22977k	43196k	13226k	34448k	61533k	21458k	53854k	85741k	36097k
		111360	283424	111312	140096	394080	144112	181120	574328	185184
$d = 4$	Ref.[94]	46043k	85151k	21849k	68019k	108992k	35859k	-	-	-
		164328	377292	111944	201180	504332	144800	-	-	-
	This work	28808k	50052k	13226k	42113k	67363k	21460k	64766k	216313k	36194k
		164352	262143	111312	201172	504324	144152	192956	381444	185184
$d = 8$	Ref.[94]	111445k	199892k	21852k	-	-	-	-	-	-
		262636	299007	111960	-	-	-	-	-	-
	This work	76364k	129326k	13226k	105337k	295197k	21447k	150611k	969455k	36193k
		262636	557604	111312	266848	488072	144152	344696	504648	185192
$d = 16$	Ref.[94]	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-
	This work	100786k	436475k	13284k						
		426492	611648	111320						

[a] The first row of each entry indicates the cycle count and the second row refers to stack usage.

05 Conclusions and Publications

- 5.1 Conclusions
- 5.2 Publications

5.1 Conclusions

❑ Theoretical improvements: Improved Plantard Arithmetic

- We proposed an **improved Plantard arithmetic** tailored for LBC.
- It has excellent merits over the original Plantard, Montgomery, and Barrett arithmetic.

❑ Implementation improvements: Efficient, lightweight and secure LBC

- We explored various optimizations for the **improved Plantard arithmetic, NTT, Keccak, Kyber, NTTRU, Dilithium and side-channel secure masking-friendly Raccoon** implementation on three IoT devices.
- All implementations are **open-source** and some of them have been merged into the NIST's official repository **pqm4.**
  - https://github.com/UIC-ESLAS/ImprovedPlantardArithmetic
  - https://github.com/UIC-ESLAS/Kyber_RV_M3
  - https://github.com/UIC-ESLAS/Dilithium-Multi-Moduli
  - https://github.com/JunhaoHuang/pqm4

5.2 Publications

[1] **Junhao Huang**, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, Donglong Chen*. Improved Plantard Arithmetic for Lattice-based Cryptography[J]. *IACR Transactions on Cryptographic Hardware and Embedded Systems* ***(TCHES),*** 2022, 2022(4).
(CCF-B & Top-tier Conference in Cryptographic Engineering)

[2] **Junhao Huang**, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Cetin Kaya Koc, Donglong Chen*. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices[J]. *IEEE Transactions on Information Forensics & Security (**TIFS**),* 2024. **(CCF-A & Top-tier Journal in Security)**

[3] **Junhao Huang**, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray CC Cheung, Cetin Kaya Koc, Donglong Chen*. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems (**TCHES**),* 2024, 2024(2).
(CCF-B & Top-tier Conference in Cryptographic Engineering)

[4] **Junhao Huang**, Jipeng Zhang, Weijia Wang, Xuan Yu, Donglong Chen, Efficient High-order Masking Raccoon on Memory Constrained Devices[J]. **(In Submission)**

5.2 Publications

[5] Jipeng Zhang, **Junhao Huang**, Lirui Zhao, Donglong Chen, Cetin Kaya Koc, ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519[C], ***Usenix Security***, 2024.
(CCF-A & Top-tier Conference in Security)
[6] Haosong Zhao, **Junhao Huang**, Zihang Chen, Kunxiong Zhu, Donglong Chen, Zhuoran Ji, Hongyuan Liu, VESTA: A Secure and Efficient FHE-based Three-Party Vectorized Evaluation System for Tree Aggregation Models[C], ***ACM SIGMETRICS***, 2025.
(CCF-B Flagship Conference in SIGMETRICS Community)
[7] Zewen Ye, **Junhao Huang**, Tianshun Huang, Yudan Bai, Jinze Li,Hao Zhang, Guangyan Li, Donglong Chen, Ray CC Cheung, Kejie Huang, PQNTRU: Acceleration of NTRU-based Schemes via Customized Post-Quantum Processor[J], *IEEE Transactions on Computers (**TC**)*, 2025.
(CCF-A Flagship Journal)
[8] Jipeng Zhang, Yuxing Yan, **Junhao Huang**, Cetin Kaya Koc*. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}-IM{B}{V}[J]. *IACR Transactions on Cryptographic Hardware and Embedded Systems (**TCHES**)*, 2025, 2025(1).
(CCF-B & Top-tier Conference in Cryptographic Engineering)

Thanks for listening!

Look forward to interesting questions and discussions!