IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–28. DOI:XXXXXXXX

Efficient and Compact High-order Masking Raccoon on Memory Constrained Devices

Junhao Huang[1], Yiteng Sun[2], Jipeng Zhang[3], Weijia Wang[2], Donglong Chen[4*], Haiyang Xue[1], Guoming Yang[1]

[1] Singapore Management University, Singapore
jhhuang_nuaa@126.com,haiyangxue@smu.edu.sg,gmyang@smu.edu.sg
[2] Shandong University, Qingdao, China sunyiteng@mail.sdu.edu.cn,wjwang@sdu.edu.cn
[3] National University of Singapore, Singapore jp-zhang@outlook.com
[4] Guangdong Provincial Key Laboratory IRADS, Beijing Normal-Hong Kong Baptist University, Zhuhai, China donglongchen@bnbu.edu.cn
* Corresponding Author.

Abstract. Raccoon, presented at CRYPTO 2024, is a provably secure lattice-based signature scheme featuring an $O(d \log d)$ masking overhead. Although it did not advance beyond the first-round of NIST's additional call for post-quantum digital signatures, Raccoon remains a promising candidate due to its low masking overhead and strong provable side-channel resistance.

This paper proposes the first efficient and compact implementation of high-order Raccoon on the ARM Cortex-M4, with a focus on optimizing its polynomial multiplication, sampling, and memory consumption. 1) For polynomial multiplication, we present a consecutive 32-bit NTT/INTT implementation for the two moduli in Raccoon, enabling optimal 3+3+3 layer merging to reduce memory-access overhead, together with extensive lazy reduction to minimize modular reductions. As a side contribution, we propose a novel negative double Montgomery reduction technique to accelerate double modular reductions in multi-moduli NTT. 2) For polynomial sampling, we identify and resolve a previously overlooked performance bottleneck caused by frequent incremental SHAKE256 invocations for small data chunks on ARM Cortex-M4. By transitioning to the one-shot approach, we achieve speedups of at least 1.80× across all sampling routines in Raccoon. 3) Finally, we explore memory-saving techniques to enable the practical deployment of high-order masked Raccoon (up to 32 orders) on resource-constrained IoT devices.

The proposed implementations outperform the Raccoon reference implementation by 1.86×–2.25×. Notably, high-order masked Raccoon-192 outperforms masked Dilithium3 by 3.11×, 8.07×, and 2.71× at masking orders $d = 2$, 4, and 8, respectively. Finally, the leakage evaluations confirm that the second-order Raccoon implementation provides strong resistance to differential power analysis (DPA) once micro-architectural transitional leakages are mitigated.

Keywords: Raccoon signature, side-channel attacks, modular arithmetic, lattice-based cryptography

1 Introduction

Raccoon [PEK+23, dPKPR24] is a lattice-based digital signature scheme designed with a focus on being masking-friendly. Similar to the NIST-standardized lattice-based signature scheme Dilithium [BDK+21], its security is based on the hardness of the Module-LWE and Module-SIS problems. Although Raccoon was submitted to the NIST call for Additional Digital Signature Schemes and did not advance to the second round, its low masking

overhead and strong provable side-channel resistance make it a promising candidate for specific applications, including smart cards, hardware security modules and IoT platform security. Furthermore, it is considered to be comparably threshold-friendly [DPKN+25].

In a side-channel attack (SCA), an adversary exploits information leaked by the physical execution of an algorithm, such as its timing behavior or power consumption. Due to the susceptibility of the lattice-based signature schemes, such as Dilithium [KAA21, FDK20, UMTS22] and Falcon [KA21, GMRR22, ZLYW23a], to SCA, masking techniques [ISW03] are commonly employed in their implementation to mitigate such attacks.

However, lattice-based signature schemes like Dilithium and Falcon include subroutines where masking is computationally expensive. For instance, in the masked implementation of Dilithium-Sign, the most efficient method to sample random errors involves first generating samples in a boolean masked form and then converting them into an arithmetic masked form. This process requires costly mask conversions [Gou01, CGV14, CGTV15]. Although mask conversions have improved over time, securely implementing these conversions still incurs a computational cost of at least $O(d^2)$, where d represents the number of shares into which sensitive data is divided. Moreover, rejection sampling, another critical component, also relies heavily on expensive mask conversions [BBE+18, BBE+19, MGTF19, GR19].

In contrast, the masking-friendly Raccoon avoids the need for mask conversions and achieves a lower masking computational overhead of $O(d \log d)$. Nevertheless, while the implementations and leakage assessment of Dilithium [MGTF19, CGTZ23, ABC+23, CGL+24] and Falcon [EFG+22, ZLYW23b, CC24, KA24, BPT+25, ZZ26] on resource-constrained devices have been extensively studied, the (high-order masked) implementation and side-channel evaluation of Raccoon remain largely unexplored.

1. Although the Raccoon team provides both portable reference and hardware implementations to demonstrate the performance of Raccoon, the performance of Raccoon on memory-constrained IoT devices, such as Cortex-M4, has not yet been explored.

2. High-order masked implementations of Raccoon require substantial memory, posing challenges for the deployment on memory-constrained IoT devices. This is because high-order masking requires splitting each polynomial into many shared polynomials, up to 32 shares, leading to significant memory consumption.

3. Furthermore, while leakage evaluation has been performed on FPGA implementations [dPPRS23], the deployment and side-channel assessment of Raccoon on real-world microarchitectures remain unexplored, as highlighted by [dPKPR24] as future work.

Motivated by this, this paper aims to fill the gap by presenting the first efficient and compact high-order masked Raccoon implementations for the ARM Cortex-M4, along with the leakage assessment of masked Raccoon to demonstrate its strong resistance to differential power analysis (DPA) attacks on real-world microarchitectures.

Contributions. Specifically, the contributions of this paper are fourfold.

- We present an efficient, consecutive 32-bit NTT/INTT implementation for the multi-moduli NTT in Raccoon. By implementing these transforms consecutively, we utilize an optimal 3+3+3 layer merging strategy to minimize memory access overhead. We further employ extensive lazy reduction strategies to significantly reduce the total number of modular reductions. Additionally, we observe that the state-of-the-art 32-bit Montgomery reduction on the Cortex-M4 is suboptimal for the double modular reductions required by Raccoon's multi-moduli NTT. To address this, we revisit Seiler's variant of Montgomery reduction and propose a novel, faster approach termed negative double Montgomery reduction. Although this method produces negative intermediate results, we demonstrate that it does not affect the subsequent

NTT/INTT computations. Moreover, we also present optimized 64-bit polynomial arithmetic tailored to Raccoon's parameters.

- We identify and resolve a previously overlooked performance bottleneck in incremental SHAKE variants on the ARM Cortex-M4. We observe that the Raccoon implementation frequently invokes incremental SHAKE256 absorb and squeeze routines for excessively small data chunks. While this granularity does not increase the core Keccak permutation count, our experiments demonstrate that it incurs substantial overhead on resource-constrained devices. To resolve this, we transition to the one-shot strategy to reduce the SHAKE256 invocations. Integrating this optimization across all sampling routines in Raccoon yields performance speedups ranging from 1.80× to 2.67×.

- To enable high-order Raccoon on memory-constrained IoT devices, we also explore several memory optimization techniques for the lightweight implementation of high-order Raccoon, addressing the challenges posed by memory-constrained IoT devices. These optimizations improve the feasibility and practicality of deploying the high-order Raccoon, ensuring strong side-channel resistance on memory-limited devices.

- Finally, we perform a leakage assessment of the masked Raccoon implementation with $d = 2$ on ARM Cortex-M4. We show that when specific countermeasures are taken to prevent the transitional leakage, Raccoon would demonstrate strong resistance to differential power analysis (DPA) attack.

Code: The implementations of this paper are open-sourced at https://github.com/JunhaoHuang/pqm4/tree/racc.

2 Preliminaries

We first briefly introduce the Raccoon scheme, then recall the time-consuming components of Raccoon, including Montgomery arithmetic and polynomial multiplication.

2.1 Notations

Let f be a function. If f is deterministic, the output of $f(x)$ is assigned to y using $y := f(x)$. If f is randomized, the assignment is denoted as $y \leftarrow f(x)$. For a given probability distribution $\mathcal{D}$, $y \leftarrow \mathcal{D}$ indicates that y is sampled from $\mathcal{D}$. Let v be a positive integer. Any integer $x \in \mathbb{Z}$ can be expressed as $x = 2^v \cdot x_h + x_l$, where $x_h \in \mathbb{Z}$ and $x_l \in [-2^{v-1}, 2^{v-1} - 1]$. The function $\lfloor \cdot \rceil_v$ is defined as $\lfloor x \rceil_v = \lfloor x/2^v \rceil = x_h$. The function $\lfloor \cdot \rfloor$ denotes the floor operation.

Let q be an integer and n a power of two. The quotient ring of integers modulo q is denoted as $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. The ring $\mathcal{R}_q$ is defined as $\mathbb{Z}_q[x]/(x^n + 1)$, representing the quotient of the polynomial ring $\mathbb{Z}_q[x]$ by the ideal generated by $(x^n + 1)$. Italic lowercase letters represent scalars, which are elements of $\mathbb{Z}$, $\mathbb{Z}_q$, or $\mathcal{R}_q$. Bold lowercase letters denote vectors, while bold uppercase letters indicate matrices. Let $f \in \mathcal{R}_q$ be a polynomial. Its negation is defined as $-f = -f_0 + \cdots + (-f_{n-1})x^{n-1}$, where each coefficient of f is negated.

For $z \in \mathbb{Z}$, we define $z \bmod^{\pm} q \in \{-\lfloor \frac{q}{2} \rfloor, -\lfloor \frac{q}{2} \rfloor + 1, \cdots, \lfloor \frac{q-1}{2} \rfloor - 1, \lfloor \frac{q-1}{2} \rfloor\}$ and $z \bmod^{+} q \in \{0, 1, \cdots, q-1\}$ to represent the canonical signed representative and canonical unsigned representative, respectively. From an implementation perspective, consider a 32-bit signed or unsigned multiplication where the lower 32 bits of the result are obtained. The binary representations of $z \bmod^{\pm} 2^{32}$ and $z \bmod^{+} 2^{32}$ are identical.

In Raccoon, arithmetic masking is used. A sensitive variable $\mathbf{x}$ is split into d components as $(\mathbf{x}_i)_{i \in [d]}$, such that $\mathbf{x} = \sum_{i \in [d]} \mathbf{x}_i \bmod q$. A d-sharing variable $(\mathbf{x}_i)_{i \in [d]}$ is denoted as $[\![\mathbf{x}]\!]$.

Algorithm 1 Raccoon key generation (keygen) [PEK+23]

Input: $\emptyset$
Output: vk, sk
1: $\mathsf{seed} \leftarrow \{0,1\}^{\kappa}$
2: $\mathbf{A} := \mathtt{ExpandA}(\mathsf{seed})$
3: $[\![\mathbf{s}]\!] \leftarrow l \times \mathtt{ZeroEncoding}(d)$
4: $[\![\mathbf{s}]\!] \leftarrow \mathtt{AddRepNoise}([\![\mathbf{s}]\!], u_t, \mathtt{rep})$
5: $[\![\mathbf{t}]\!] := \mathbf{A} \cdot [\![\mathbf{s}]\!]$
6: $[\![\mathbf{t}]\!] \leftarrow \mathtt{AddRepNoise}([\![\mathbf{t}]\!], u_t, \mathtt{rep})$
7: $\mathbf{t} := \mathtt{Decode}([\![\mathbf{t}]\!])$
8: $\mathbf{t} := \lfloor \mathbf{t} \rceil_{v_t}$
9: **return** $(\mathsf{vk} = (\mathsf{seed}, \mathbf{t}), \mathsf{sk} = (\mathsf{vk}, [\![\mathbf{s}]\!]))$

2.2 Raccoon

Raccoon's core functions include key generation (Algorithm 1), signing (Algorithm 2), and verification (Algorithm 3). Both key generation and signing are implemented in masked form, with an unmasked version available when $d = 1$.

In the `keygen` process, d-sharings of the small errors $[\![s]\!]$ and $[\![e]\!]$ are generated in lines 4 and 6, respectively. The verification key is computed as an LWE sample $(\mathbf{A}, \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$.

Raccoon follows the Fiat-Shamir paradigm, similar to Dilithium. However, unlike Dilithium, the `sign` process of Raccoon does not require rejection sampling, which makes it more masking-efficient. The main steps of `sign` are as follows: (1) Commit (lines 4 to 9): Random noise is generated in masked form, and the LWE commitment $\mathbf{w}$ is computed in masked form. Afterward, $\mathbf{w}$ is unmasked. (2) Challenge (lines 10 and 11): A challenge is derived using the message msg, the verification key vk, and the unmasked LWE commitment $\mathbf{w}$. (3) Response (lines 14 to 18): A response $(\mathbf{h}, \mathbf{z})$ is computed based on the random noise, the private key, and the challenge. The second part of the response, $\mathbf{z}$, is unmasked since it only involves public data. (4) `CheckBounds`: This procedure is used solely to verify the correctness of the signature and does not leak any information about the secret key. Therefore, no masking is required to protect this step.

The `verify` process aligns closely with other lattice-based Fiat-Shamir signatures, following standard procedures.

The $[\![z]\!] \leftarrow \mathtt{ZeroEncoding}(d)$ subroutine takes as input a power-of-two integer d and outputs a uniform d-sharing $[\![z]\!] \in \mathcal{R}_q^d$ of $0 \in \mathcal{R}_q$, denoted as $[\![0]\!]$. The $[\![x]\!]' \leftarrow \mathtt{Refresh}([\![x]\!])$ subroutine takes as input a d-sharing $[\![x]\!]$ of $x \in \mathcal{R}_q$, refreshes the input d-sharing, and outputs a fresh d-sharing satisfying $[\![x]\!]' = [\![x]\!] + [\![0]\!]$. $\mathtt{SU}(u, T) = [T] \cdot \mathcal{U}(-2^{u-1}, \ldots, 2^{u-1} - 1)$ represents the sum of T independent uniform random variables, each sampled from the uniform distribution $\mathcal{U}(-2^{u-1}, \ldots, 2^{u-1} - 1)$. The $[\![\mathbf{v}]\!] \leftarrow \mathtt{AddRepNoise}([\![\mathbf{v}]\!], u, \mathrm{rep})$ subroutine implements $\mathtt{SU}(u, d \cdot \mathrm{rep})$. For each masked polynomial $[\![v_{i,j}]\!]$ of the masked vector $[\![\mathbf{v}]\!]$, every share of $[\![v_{i,j}]\!]$ is updated by adding small uniform noise, then $[\![v_{i,j}]\!]$ is refreshed. This process is repeated rep times. For details on detailed parameters and other auxiliary functions used in Raccoon, we refer to [PEK+23].

2.3 Modular arithmetic

The choice of modular arithmetic depends on the modulus size and the characteristics of the target platform. Since this work primarily concerns 32-bit modular arithmetic on 32-bit ARM Cortex-M4, Montgomery [Mon85] and Barrett [Bar86] multiplications are two natural candidates for NTT and INTT implementations. Among them, Montgomery multiplication is the state-of-the-art choice on Cortex-M4 platforms [GKS20, ACC+22],

Algorithm 2 Raccoon signature generation (`sign`) [PEK+23]

Input: $\mathsf{sk} = (\mathsf{vk}, [\![\mathbf{s}]\!])$ and $\mathsf{msg} \in \{0,1\}^*$.
Output: Signature $\mathsf{sig} = (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$

1: $(\mathsf{vk}, [\![\mathsf{s}]\!]) := \mathsf{sk}, (\mathsf{seed}, \mathsf{t}) := \mathsf{vk}$
2: $\mu := \mathrm{H}(\mathrm{H}(\mathsf{vk}) \| \mathsf{msg})$
3: $\mathbf{A} := \texttt{ExpandA}(\mathsf{seed})$
4: $[\![\mathbf{r}]\!] \leftarrow l \times \texttt{ZeroEncoding}(d)$
5: $[\![\mathbf{r}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{r}]\!], u_w, \texttt{rep})$
6: $[\![\mathbf{w}]\!] := \mathbf{A} \cdot [\![\mathbf{r}]\!]$
7: $[\![\mathbf{w}]\!] \leftarrow \texttt{AddRepNoise}([\![\mathbf{w}]\!], u_w, \texttt{rep})$
8: $\mathbf{w} := \texttt{Decode}([\![\mathbf{w}]\!])$
9: $\mathbf{w} := \lfloor \mathbf{w} \rceil_{v_w}$
10: $c_{\mathsf{hash}} := \texttt{ChalHash}(\mathbf{w}, \mu)$
11: $c_{\mathsf{poly}} := \texttt{ChalPoly}(c_{\mathsf{hash}})$
12: $[\![\mathbf{s}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{s}]\!])$
13: $[\![\mathbf{r}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{r}]\!])$
14: $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$
15: $[\![\mathbf{z}]\!] \leftarrow \texttt{Refresh}([\![\mathbf{z}]\!])$
16: $\mathbf{z} := \texttt{Decode}([\![\mathbf{z}]\!])$
17: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - 2^{v_t} \cdot c_{\mathsf{poly}} \cdot \mathbf{t}$
18: $\mathbf{h} := \mathbf{w} - \lfloor \mathbf{y} \rceil_{v_w}$
19: $\mathsf{sig} := (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$
20: **if** $\{\texttt{CheckBounds}(\mathsf{sig}) = \mathsf{FAIL}\}$ **goto** Line 4
21: **return** sig

whereas Barrett multiplication achieves better performance on SIMD architectures such as NEON and AVX2 [BHK+22].

Compared with Montgomery multiplication, the Barrett multiplication in [BHK+22, Algorithm 9] requires the same number of multiplication operations, but returns an output whose absolute value is bounded by $\frac{3q}{2}$, whereas Montgomery multiplication in [GKS20, ACC+22] returns an output bounded by q. Consequently, on Cortex-M4, Barrett multiplication provides no instruction-count advantage and yields a larger output range by $\frac{q}{2}$ compared to Montgomery multiplication. This larger range may require extra modular reductions in NTT/INTT implementations. Moreover, the Barrett multiplication in [BHK+22, Algorithm 9] requires two twiddle-related constants, namely ζ and $\lfloor \frac{\zeta R}{q} \rceil$, which further increases the memory-access cost of twiddle factors in NTT/INTT. Therefore, we focus on state-of-the-art Montgomery arithmetic for 32-bit modular arithmetic in this paper. For further details on Barrett multiplication, we refer the reader to [BHK+22].

To clarify the discussion, we introduce two terms: original Montgomery reduction [Mon85] and Seiler's Montgomery reduction variant [Sei18]. Let β denote the word size required such that q fits within one word. For example, $\beta = 2^{32}$ for $2^{16} < q < 2^{32}$. Let a represent the double-width value to be reduced and $a' \equiv a\beta^{-1} \mod q$.

The original Montgomery reduction, also referred to as $\mathbf{mont}_q^-(a)$ in [BHK+22, Alg 2], is computed as:

$$a' \leftarrow \frac{a + (a(-q^{-1} \bmod \beta) \bmod^{\pm} \beta)q}{\beta}.$$

Seiler's variant of Montgomery reduction, often called signed Montgomery reduction [Sei18, Alg 3] and denoted as $\mathbf{mont}_q^+(a)$ in [BHK+22, Alg 2], is calculated as:

$$a' \leftarrow \frac{a - (a(q^{-1} \bmod \beta) \bmod^{\pm} \beta)q}{\beta}.$$

Algorithm 3 Raccoon signature verification (`verify`) [PEK+23]

Input: $\mathsf{vk} = (\mathsf{seed}, \mathbf{t})$, msg, and $\mathsf{sig} = (c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z})$
Output: OK (accept) or FAIL (reject).
1: $(c_{\mathsf{hash}}, \mathbf{h}, \mathbf{z}) := \mathsf{sig}, (\mathsf{seed}, \mathsf{t}) := \mathsf{vk}$
2: **if** $\{\texttt{CheckBounds}(\mathsf{sig}) = \mathsf{FAIL}\}$ **then** FAIL
3: $\mu := \mathrm{H}(\mathrm{H}(\mathsf{vk})||\mathsf{msg})$
4: $\mathbf{A} := \texttt{ExpandA}(\mathsf{seed})$
5: $c_{\mathsf{poly}} := \texttt{ChalPoly}(c_{\mathsf{hash}})$
6: $\mathbf{y} := \mathbf{A} \cdot \mathbf{z} - 2^{v_t} \cdot c_{\mathsf{poly}} \cdot \mathbf{t}$
7: $\mathbf{w}' := \lfloor \mathbf{y} \rceil_{v_w} + \mathbf{h}$
8: $c'_{\mathsf{hash}} := \texttt{ChalHash}(\mathbf{w}', \mu)$
9: **if** $c_{\mathsf{hash}} \neq c'_{\mathsf{hash}}$ **return** FAIL
10: **return** OK

The advantage lies in its ability to use single-width operations for implementation. Since $a - (a(q^{-1} \bmod \beta) \bmod^{\pm} \beta)q$ is divisible by β, the lower half of the subtraction can be safely ignored. In contrast, the original version cannot achieve this simplification, as the addition in its computation may result in a carry from the lower to the upper half.

Regarding the output range, the bound is given by:

$$|a'| \leq \frac{|a| + |(\_ \bmod^{\pm} \beta)q|}{\beta} \leq \frac{|a|}{\beta} + \frac{q}{2}.$$

This inequality holds for both the original and Seiler's versions.

2.4 Polynomial multiplication in Raccoon

The polynomial ring used in Raccoon is $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, where $q = q_1 \times q_2$, $q_1 = 2^{24} - 2^{18} + 1$, $q_2 = 2^{25} - 2^{18} + 1$, and $n = 512$. To enable efficient polynomial multiplication on 32-bit platforms such as the Cortex-M4, the Chinese Remainder Theorem (CRT) is employed, similar to previous works [ACC+22, HP23, HAZ+24]. The CRT divides the 49-bit NTT into two smaller NTTs possessing 24-bit and 25-bit moduli respectively. This approach is referred to as multi-moduli NTT in [ACC+22] and can be expressed as an isomorphism $\mathbb{Z}_q \cong \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$. This decomposition enables efficient implementation of the 24- and 25-bit NTTs using 32-bit multipliers on 32-bit platforms. The polynomial rings for the 24- and 25-bit NTTs in Raccoon are abbreviated as $\mathcal{R}_{q_i}$ for $i = 1, 2$.

Since $q_i - 1$ is divisible by $2n$, there exists a primitive $2n$-th root in $\mathbb{Z}_{q_i}$, which we denote as ζ_{q_i}, where $i \in \{1, 2\}$. The forward NTT involves the following mapping: $\mathbb{Z}_{q_i}[X]/(X^{512} + 1) \cong \prod_{j=0}^{511} \mathbb{Z}_{q_i}[X]/(X - \zeta_{q_i}^{2j+1})$. The inverse NTT (INTT) is the inverse of the above mapping. The NTT-based polynomial multiplication can be expressed as $f \cdot g = \mathrm{INTT}(\mathrm{NTT}(f) \odot \mathrm{NTT}(g))$, where $\odot$ represents degree-0 pointwise multiplication. The Cooley-Tukey (CT) butterfly algorithm [CT65] is often used to implement the forward NTT, while the Gentleman-Sande (GS) butterfly algorithm [GS66] is frequently employed to implement the INTT. It is pointed out in [ACC+22] that using the CT algorithm to implement the INTT also has advantages in certain situations.

The overall process of utilizing the multi-moduli NTT in Raccoon consists of three main steps:

1. Polynomial splitting: Two consecutive modular reductions are performed over q_1 and q_2. For example, for a polynomial coefficient a stored in a 64-bit data type, after two modular reductions, $a \bmod q_1$ and $a \bmod q_2$ directly overwrite the storage space of a. This in-place approach is memory-friendly and preferable to embedded devices.

2. NTT operations: Perform NTTs over q_1 and q_2, followed by pointwise multiplications and INTTs modulo q_1 and q_2.

3. Reconstruction using CRT: Reconstruct the 32-bit results (e.g., $c_1 = c \bmod q_1, c_2 = c \bmod q_2$) modular q using the CRT theorem: $c = c_1 q_2 (q_2^{-1} \bmod q_1) + c_2 q_1 (q_1^{-1} \bmod q_2) \bmod q$.

2.5 Target platform: ARM Cortex-M4 processor

The ARM Cortex-M4 processor is a part of the ARMv7-M microcontroller profile, featuring sixteen 32-bit registers (`r0-r15`). Among these, `r13` is reserved as the stack pointer, `r14` as the link register, and `r15` as the program counter.

The Cortex-M4 employs a three-stage pipeline, where most instructions execute in a single cycle. Exceptions include branches, divisions, and memory operations, which may require additional cycles. For instance, `ldr` and `str` instructions typically take 2 and 1 cycles, respectively, though unaligned data can increase `ldr` latency to up to 3 cycles. When no dependencies exist, multiple `ldr` instructions can be pipelined, executing in $s+1$ cycles for s operations, while a `str` following an `ldr` incurs no additional delay. Equipped with a barrel shifter, the Cortex-M4 allows operand shifts or rotations within instructions without performance penalties. All multiplication instructions execute in a single cycle on the Cortex-M4.

We represent the product of two 32-bit integers as $a \cdot b$, the lower 32 bits of this product as $\text{low}_{32}(a \cdot b)$, and the higher 32 bits as $\text{high}_{32}(a \cdot b)$. A 64-bit integer $c_l + c_h \cdot 2^{32}$ is expressed as $\{c_l, c_h\}$. The following are several commonly used instructions, with those beginning with "s" indicating signed integer operations.

The instruction `mul` c, a, b computes $c \leftarrow \text{low}_{32}(a \cdot b)$. The instruction `smlal` c_l, c_h, a, b calculates $\{c_l, c_h\} \leftarrow \{c_l, c_h\} + a \cdot b$. The `smmul` c, a, b instruction computes $c \leftarrow \text{high}_{32}(a \cdot b)$. The `smmla` c, a, b, d instruction computes $c \leftarrow \text{high}_{32}(a \cdot b) + d$. The `smmls` c, a, b, d instruction computes $c \leftarrow \left\lfloor \frac{d \cdot 2^{32} - a \cdot b}{2^{32}} \right\rfloor$ and we will discuss it again in Section 3.

3 Optimized Raccoon implementation

This section presents the optimized Raccoon implementation on Cortex-M4.

3.1 Optimized polynomial multiplication

The polynomial multiplication in Raccoon operates in the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ with a 49-bit modulus q. To enable efficient polynomial multiplication on the 32-bit Cortex-M4 platform, the 49-bit NTT computations are divided into two 32-bit NTTs using the CRT theorem. This approach ensures compatibility with the platform's constraints while maintaining computational efficiency. This section focuses on optimizing the 32-bit NTT/INTT implementations and the polynomial splitting step, which involves double modular reductions.

During the forward NTT implementation, we separate the polynomial splitting and the two NTTs over q_i into two distinct assembly subroutines, since the double modular reduction subroutine can be reused by other operations in Raccoon. In contrast, we merge the INTTs over q_i with the CRT computation at the end of the inverse NTT, as the CRT can be efficiently integrated into the final INTT layer without incurring additional memory accesses.

3.1.1 The 32-bit NTT/INTT implementations

We first present our optimized 32-bit NTT/INTT implementations for q_1 and q_2. The NTT/INTT computations over q_1 and q_2 are followed by the in-place double modular reductions described in Subsubsection 3.1.2.

Due to the register constraints of the Cortex-M4, which has only 14 general-purpose registers, performing the NTT/INTT computations for both q_1 and q_2 simultaneously is challenging, which would prevent us from implementing a 3-layer merging strategy, restricting us to a 2-layer merging strategy instead, thus increasing the number of memory accesses. As a result, we perform the two consecutive NTT/INTT computations sequentially, ensuring that all available registers are fully utilized to accelerate each NTT/INTT computation over q_i.

We adopt the 3-layer merging approach, similar to that in Dilithium [AHKS22]. However, the 9-layer NTT computation in Raccoon is particularly well-suited for this strategy, as a 3+3+3 layer merging can be employed without requiring additional layers of memory accesses compared to the 8-layer NTT in Dilithium. The 32-bit modular multiplication by the twiddle factor in the butterfly units is implemented using Algorithm 4, as presented in [GKS20]. We also utilize the CT butterfly for both NTT and INTT implementations in Raccoon, as recommended by [ACC+22, AHKS22]. Unlike the GS butterfly, the CT butterfly helps limit coefficient growth and reduces the number of modular reductions required in INTT, as detailed below.

Lazy reduction in NTT/INTT. For Raccoon's $q = q_1 q_2 = 16515073 \times 33292289 = 549824583172097$, a 64-bit signed integer can accommodate absolute values up to $16775q$ while a 32-bit signed integer can accommodate values up to $130q_1$ and $64q_2$ for q_1 and q_2, respectively. To leverage the lazy reduction technique in NTT/INTT, we need to ensure that all of the polynomial coefficients will not exceed these bounds.

As detailed in Subsubsection 3.1.2, the double modular reductions split the 64-bit polynomial over $\mathbb{Z}_q$ into two 32-bit polynomials over $\mathbb{Z}_{q_i}$. This step ensures that all coefficients of the input polynomial are reduced to the range $(-q_i, q_i)$. Subsequently, the 9-layer 32-bit NTT, taking $(-q_i, q_i)$ as input, is executed using the Cooley-Tukey (CT) butterfly structure. After the 9-layer NTT, the coefficients increase by at most $9q_i$. These coefficients can then be directly used in the subsequent matrix-vector multiplication without requiring additional modular reductions during the NTT computation.

The CT butterfly algorithm is also employed in the INTT, as in [ACC+22]. The use of the CT algorithm in the INTT omits certain modular reductions, referred to as "light butterflies" [ACC+22]. The first three layers of INTT with light butterflies would expand $n/8$ of the coefficients by a factor of 8 and $n/8$ of the coefficients by a factor of 4. The subsequent three layers of light butterflies are applied to only 64 coefficients. Among these, 8 coefficients are expanded by a factor of 8, while another 8 coefficients are expanded by a factor of 4.

The input polynomial for the INTT typically originates from the matrix-vector multiplication, which produces polynomials with maximum coefficients in absolute value of up to $l \cdot q_i$, specifically $4q_i$, $5q_i$, and $7q_i$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. After the first three layers of the INTT, the coefficients may grow to $32q_i$, $40q_i$, and $56q_i$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. These values remain below the maximum bounds ($130q_1$ and $64q_2$). To apply the second three layers of light butterflies for the 64 coefficients, these coefficients must first be reduced using Barrett reduction. The remaining INTT computations, utilizing the standard CT butterfly algorithm, will increase the remaining 448 coefficients by $6q_i$, resulting in coefficients up to $38q_i$, $46q_i$, and $62q_i$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. These bounds remain within the maximum limits, ensuring that the CT butterflies can be executed safely and efficiently without requiring additional modular reductions. All coefficients will be

Algorithm 4 The state-of-the-art 32-bit Montgomery reduction [GKS20]

Input: The 64-bit coefficient $a = a_l + a_h \cdot 2^{32}$
Output: $a \cdot 2^{-32} \bmod q_i$
1: `mul` $t, a_l, -q_i'$
2: `smlal` a_l, a_h, t, q_i
3: **return** a_h

reduced at the final layer of the INTT together with the CRT computation, as discussed below. In summary, the INTT computation with CT butterflies only requires modular reductions for 64 coefficients for each q_i.

After INTT, the CRT computation is required to combine two 32-bit polynomials over $\mathbb{Z}_{q_i}$ to the 64-bit polynomial over $\mathbb{Z}_q$. Let c denote the 64-bit coefficient in the 64-bit polynomial over $\mathbb{Z}_q$, and $c_1 = c \bmod q_1$ and $c_2 = c \bmod q_2$ be two coefficients that need to be combined in the CRT computation. The CRT computation proceeds as follows: $c = c_1 q_2 (q_2^{-1} \bmod q_1) + c_2 q_1 (q_1^{-1} \bmod q_2) \bmod q$. Similar to the reference implementation of Raccoon [PEK+23], the CRT computation is performed together with the multiplication of $n^{-1} \bmod q_i$ at the end of the two 32-bit INTT computations. Specifically, we precompute $\zeta_1 = \zeta \cdot (q_2 n)^{-1} \cdot (2^{32})^4 \bmod q_1$ and $\zeta_2 = \zeta \cdot (q_1 n)^{-1} \cdot (2^{32})^4 \bmod q_2$ to merge the multiplication of q_i^{-1}, n^{-1}, and the normalization factor $(2^{32})^4$ into a single Montgomery multiplication at the final layer of the INTT, i.e., $c_i' = c_i \cdot \zeta_i$. The final modular multiplications in the INTT reduce all coefficients back to the range $(-q_i, q_i)$. The subsequent CRT computation, $c = c_1' q_2 + c_2' q_1$, is efficiently implemented using the `smull` and `smmla` instructions, producing coefficients in the range $(-2q, 2q)$. To prepare the coefficients for use in subsequent masking gadgets or signature serialization, we further reduce them to the range $[0, q)$ using one Barrett reduction followed by a conditional addition.

3.1.2 Double modular reductions

The polynomial splitting is required to perform two consecutive 32-bit modular reductions modulo q_1 and q_2 for each 64-bit polynomial coefficient $a = a_l + a_h \cdot 2^{32}$ before the NTT/INTT implementation, which we refer to as double modular reductions. The state-of-the-art approach for 32-bit modular reduction on Cortex-M4 relies on Montgomery reduction [Mon85], which will be the primary focus of this discussion.

In this section, we highlight the effectiveness of Seiler's Montgomery reduction in efficiently implementing the double modular reductions. By leveraging Seiler's approach, we achieve a more register-efficient and in-place implementation, reducing the instruction count and intermediate register usage compared to the state-of-the-art Montgomery reduction. This optimization is particularly beneficial for the Cortex-M4 platform, where register constraints are a critical consideration. It should be noted that this optimization can also be extended to Cortex-M7 with similar instructions.

Original Montgomery Reduction. The state-of-the-art 32-bit Montgomery reduction on Cortex-M4, as detailed in Algorithm 4, was presented in [GKS20]. However, this implementation modifies the input values a_l and a_h after Step 2 in Algorithm 4, preventing the second reduction from being performed efficiently. Consequently, to execute the second Montgomery reduction for $a = a_l + a_h \cdot 2^{32}$, it is necessary to pre-save the inputs a_l and a_h into two additional registers (as seen in Steps 2 and 3 in Algorithm 5) before performing the Montgomery reduction modulo q_1. As shown in Algorithm 5, performing the double modular reductions with the original Montgomery reduction requires six instructions and three intermediate registers.

Algorithm 5 Double modular reductions with original Montgomery reduction

Input: $a = a_l + a_h \cdot 2^{32}$
Output: $a \cdot 2^{-32} \bmod q_1, a \cdot 2^{-32} \bmod q_2$
1: `mul` $t_3, a_l, -q_2'$
2: `mov` t_1, a_l
3: `mov` t_2, a_h
4: `smlal` a_l, a_h, t_3, q_2
5: `mul` $t_3, t_1, -q_1'$
6: `smlal` t_1, t_2, t_3, q_1
7: **return** t_2, a_h

Algorithm 6 Double modular reductions with Seiler's Montgomery reduction

Input: $a = a_l + a_h \cdot 2^{32}$
Output: $a \cdot 2^{-32} \bmod q_1, a \cdot 2^{-32} \bmod q_2$
1: `mul` t_1, a_l, q_1'
2: `mul` t_2, a_l, q_2'
3: `smmul` t_1, t_1, q_1
4: `smmul` t_2, t_2, q_2
5: `sub` a_l, a_h, t_1
6: `sub` a_h, a_h, t_2
7: **return** a_l, a_h

Seiler's Montgomery Reduction. This limitation motivates the adoption of Seiler's variant of Montgomery reduction (Seiler's Montgomery reduction), as proposed in [Sei18], for implementing double modular reductions. Seiler's variant, illustrated in Algorithm 6, offers an advantage: the final subtraction in Montgomery reduction depends solely on a_h. This eliminates the need to rely on a_l after Step 1 and 2 in Algorithm 6. In this context, during Steps 3 through 6, only the most significant halves of $t_1 \times q_1$ and $t_2 \times q_2$ need to be processed, allowing subtraction from a_h independently. This sequence can be executed entirely in-place. As a result, Seiler's Montgomery reduction enables double modular reductions with only six instructions and two intermediate registers, yielding a register-efficient and implementation-friendly variant of the double modular reductions.

`SMMLS vs. SMMUL and SUB.` During the optimization process of this algorithm, we identified a subtle difference between the ARM Cortex-M4 instructions `smmls` and the combination of `smmul` and `sub`. This distinction is documented here to prevent future researchers from encountering a similar challenge.

According to the "Cortex-M4 Devices Generic User Guide"[1], the description of `smmls` aligns with performing `smmul` followed by `sub`. For example, Steps 3-6 of Algorithm 6 should theoretically be further optimized by replacing these steps with `smmls` a_l, t_1, q_1, a_h and `smmls` a_h, t_2, q_2, a_h, which could reduce the cycle counts down to 4. However, we observe that the results obtained using `smmls` are smaller by 1 compared to those produced by executing `smmul` and `sub` separately, when the low half of the product $t_1 \times q_1$ is non-zero. Upon further investigation, we discover on the ARM developer website[2] that the `smmls` a_l, t_1, q_1, a_h instruction computes:

$$a_l = \left\lfloor \frac{a_h \cdot 2^{32} - t_1 \times q_1}{2^{32}} \right\rfloor, \tag{1}$$

rather than

$$a_l = a_h - \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor, \tag{2}$$

which is the desired result for Seiler's Montgomery reduction and needs to be achieved by separately using `smmul` followed by `sub`. The discrepancy arises because the low half of the product $t_1 \times q_1$ is generally non-zero while the lower 32 bits of $a_h \cdot 2^{32}$ are all zero, causing the low half of the 64-bit subtraction $a_h \cdot 2^{32} - t_1 \times q_1$ in Equation 1 to produce a borrow bit to the upper 32 bits. This explains why results from `smmls` are mostly smaller by 1 compared to those computed with `smmul` and `sub`. Consequently, `smmls` and the combination of `smmul` and `sub` are not interchangeable on Cortex-M4, and Cortex-M7 with

[1] https://developer.arm.com/documentation/dui0553/latest/
[2] https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/multiply-instructions/smmul--smmla--and-smmls

Algorithm 7 The proposed negative double Montgomery reductions

Input: The 64-bit coefficient $a = a_l + a_h \cdot 2^{32}$, 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_l, q_1'
2: `mul` t_2, a_l, q_2'
3: `neg` a_h, a_h
4: `smmla` a_l, t_1, q_1, a_h
5: `smmla` a_h, t_2, q_2, a_h
6: **return** a_l, a_h

the same instructions. The design of `smmls` renders it unsuitable for further optimizing Seiler's Montgomery reduction.

Proposed negative double Montgomery reductions. To mitigate the issue with `smmls` and further optimize double Montgomery reductions, we propose a faster implementation, referred to as negative double Montgomery reductions. This approach computes the negative of the desired result:

$$a_l' = -a \cdot 2^{-32} \bmod^{\pm} q_1, a_h' = -a \cdot 2^{-32} \bmod^{\pm} q_2. \tag{3}$$

The negative variant of modular reduction does not impact the subsequent NTT or Raccoon computations, as will be demonstrated later. Recall that in the `smmls`, the 64-bit subtraction $a_h \cdot 2^{32} - t_1 \times q_1$ introduces a borrow bit that compromises the correctness of the operation. This occurs because the low 32 bits of $a_h \cdot 2^{32}$ are all zeros, making the subtraction sensitive to the borrow. To circumvent this issue, we reverse the operation by subtracting $a_h \cdot 2^{32}$ from $t_1 \times q_1$, avoiding the errors caused by the borrow bit. In the proposed method, we first negate a_h (Step 3 of Algorithm 7), followed by using the `smmla` instruction: $a_l = \texttt{smmla}(t_1, q_1, -a_h)$. This instruction effectively computes the addition of the negated a_h and yields:

$$a_l = \left\lfloor \frac{t_1 \times q_1 + (-a_h \cdot 2^{32})}{2^{32}} \right\rfloor = \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor + (-a_h) = -\left(a_h - \left\lfloor \frac{t_1 \times q_1}{2^{32}} \right\rfloor \right). \tag{4}$$

This demonstrates that the sequence is equivalent to the negative of the desired result of Seiler's Montgomery reduction (Equation 2). Compared to the two variants in Algorithm 5 and Algorithm 6, the proposed algorithm (Algorithm 7) is one instruction faster. Moreover, similar to Algorithm 6, it requires only two intermediate registers.

3.1.3 Other adaptations for negative polynomials

Consider a negative polynomial $-a$. Since the Number Theoretic Transform (NTT) is a ring isomorphism [AB74], we have $\mathrm{NTT}(-a) = -\mathrm{NTT}(a)$, which implies that negating a polynomial results in the negation of its NTT representation. This also applies to the INTT.

Recall that the NTT for the polynomial multiplication of a and b in R_q is computed as $c = \mathrm{INTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$, where $\circ$ denotes pointwise multiplication of the NTT-transformed elements. For two negative polynomials $-a$ and $-b$, the polynomial multiplication using NTT is performed as:

$$c = \mathrm{INTT}(\mathrm{NTT}(-a) \circ \mathrm{NTT}(-b)) = \mathrm{INTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b)).$$

In this case, the pointwise multiplication of $\mathrm{NTT}(-a)$ and $\mathrm{NTT}(-b)$ naturally cancels out the negative signs, and the subsequent INTT computation is performed as usual.

During Raccoon's key generation and signature generation, we also need to handle the polynomial multiplication of a negative polynomial and a normal polynomial. Consequently, it is essential to keep track of negative polynomials in the signature generation and utilize the associated components in Raccoon to properly manage the negative signs.

For example, during the computation of $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$ in Step 16 of Algorithm 2, the use of the proposed double modular reductions (Algorithm 7) and NTT for negative polynomials results in the NTT representations $-\hat{c}_{\mathsf{poly}}$, $-[\![\hat{\mathbf{s}}]\!]$ and $-[\![\hat{\mathbf{r}}]\!]$, where $[\![x]\!]$ denotes the mask form of the sensitive data x. While the pointwise multiplication of $(-\hat{c}_{\mathsf{poly}}) \circ (-[\![\hat{\mathbf{s}}]\!])$ cancels out the negative signs, it still requires subtracting $-[\![\hat{\mathbf{r}}]\!]$ to obtain the correct results. Instead of directly subtracting $-[\![\hat{\mathbf{r}}]\!]$, we observe that the Refresh() component is required to refresh $-[\![\mathbf{s}]\!]$ and $-[\![\mathbf{r}]\!]$ in Steps 12 and 13 of Algorithm 2. Because this component mainly consists of the addition of $[\![\hat{\mathbf{0}}]\!]$, we find it perfect to handle the negative signs properly. Therefore, we propose two variants of the Refresh() component in Raccoon for the computation of $[\![\mathbf{z}]\!] := c_{\mathsf{poly}} \cdot [\![\mathbf{s}]\!] + [\![\mathbf{r}]\!]$.

The first variant is computed as $\mathsf{Refresh}_1(-[\![\hat{\mathbf{r}}]\!]) = (-[\![\hat{\mathbf{0}}]\!]) + (-[\![\hat{\mathbf{r}}]\!]) = -([\![\hat{\mathbf{0}}]\!] + [\![\hat{\mathbf{r}}]\!])$, where the NTT-domain element $-[\![\hat{\mathbf{0}}]\!]$ is also optimized with the proposed double modular reductions described in Algorithm 7. This approach maintains the negative signs for the follow-up computations. The second variant is computed as $\mathsf{Refresh}_2(-[\![\hat{\mathbf{s}}]\!]) = [\![\hat{\mathbf{0}}]\!] - (-[\![\hat{\mathbf{s}}]\!]) = [\![\hat{\mathbf{0}}]\!] + [\![\hat{\mathbf{s}}]\!]$, where the NTT-domain element $[\![\hat{\mathbf{0}}]\!]$ is implemented using the double modular reductions with Seiler's Montgomery reduction, as shown in Algorithm 6. This variant converts the negative elements into positive elements. Using the above methods, the computation proceeds as: $-[\![\hat{\mathbf{z}}]\!] := -\hat{c}_{\mathsf{poly}} \cdot [\![\hat{\mathbf{s}}]\!] + (-[\![\hat{\mathbf{r}}]\!])$, and then in Step 15 of Algorithm 2, $\mathsf{Refresh}_2(-[\![\hat{\mathbf{z}}]\!])$ is used to eliminate the negative signs. It should be noted that these adaptations do not add extra cost to Raccoon's signature generation. On the contrary, the integration of the proposed double modular reductions with Seiler's Montgomery reduction (Algorithm 7 and Algorithm 6) contributes to a further speedup of the Refresh() function.

3.2 Optimized polynomial sampling

Polynomial sampling is the most time-consuming component in lattice-based cryptography, including Raccoon. Therefore, many efforts have been devoted to optimizing the underlying Keccak permutation for efficient polynomial sampling [BK22, HAZ+24, ZYHK25]. However, the implementation efficiency of high-level extendable output functions (XOFs) like SHAKE128 and SHAKE256 has been largely overlooked [KCP16]. Moreover, since NIST's introduction of incremental SHAKE in 2025 for applications with unknown input or output lengths[3], a systematic evaluation of its performance has been absent. Consequently, schemes such as Raccoon [PEK+23], HAWK [DPPvW22], PERK [ABB+23], and AIMer [LCH+24] have extensively adopted incremental SHAKE under the assumption of comparable performance.

In this section, we invalidate this assumption through a comprehensive performance analysis of incremental SHAKE256 on the ARM Cortex-M4 platform. Our findings provide critical insights for future scheme designers, highlighting the need for caution when employing the incremental SHAKE to avoid unnecessary performance degradation.

3.2.1 Incremental vs. One-Shot SHAKE256: A Systematic Performance Evaluation

Raccoon mainly relies on the SHAKE256 XOF for various sampling operations. Therefore, we only conduct the performance evaluation of SHAKE256 in this section, while the same insights and optimizations can be applied to SHAKE128. The SHAKE XOF operates in two main stages: the absorb phase, where input seeds are incorporated into the 1600-bit internal state, and the squeeze phase, where the required volume of pseudo-random data is extracted.

[3] https://csrc.nist.gov/News/2025/decision-to-update-fips-202-and-revise-sp-800-185

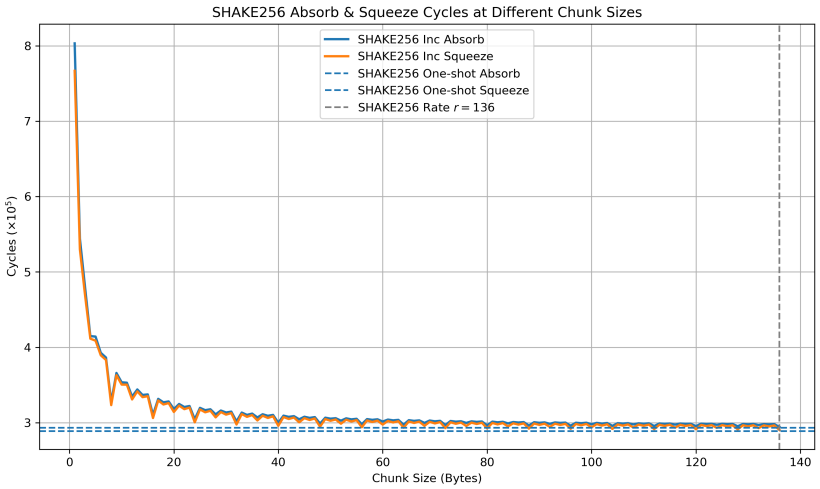


Figure 1: Comparison of cycles for one-shot vs. incremental SHAKE256 at varying chunk sizes for 136-byte input/output data on ARM Cortex-M4

In the Raccoon reference implementation [PEK+23], the sampling of each coefficient invokes the SHAKE256 absorb and squeeze phases independently using the incremental variant of SHAKE256. This approach results in at least N distinct SHAKE invocations for polynomial sampling. In contrast, Dilithium [BDK+21] adopts the one-shot approach, absorbing or squeezing the total required pseudo-random bytes in a single SHAKE invocation and caching them for subsequent operations. While the theoretical number of underlying Keccak permutations remains identical for both methods when the total number of bytes absorbed or squeezed are fixed, we observed that the computational overhead of frequent, small-scale incremental invocations is substantial on the Cortex-M4 platform, leading to severe performance degradation.

To quantify this overhead, we measure the cycle counts required to absorb or squeeze 136 bytes of data (the SHAKE256 output rate $r = 136$) using both one-shot and incremental SHAKE256 across varying chunk sizes. Because the performance degradation patterns on the ARM Cortex-M4 remain consistent for input/output lengths exceeding 136 bytes, we present the 136-byte scenario as a representative example. For lengths smaller than 136 bytes, the incremental approach with larger chunk sizes marginally outperforms the one-shot variant; the latter incurs additional overhead to process a full block, whereas the incremental method can terminate early. Nevertheless, since most of the sampling operations in Raccoon require input/output lengths much larger than 136 bytes, the remainder of our analysis focuses exclusively on this larger-length regime.

As illustrated in Figure 1, a substantial performance disparity exists at low granularities. When the chunk size is smaller than 8 bytes, the performance drops significantly. Especially for a 1-byte incremental chunk size, the absorb phase for 136 bytes requires 32 246 cycles, with a 184% increase in computational overhead compared to the one-shot baseline of 11 338 cycles. Furthermore, multiple marked reductions in cycle counts are observed when the chunk size reaches 8 bytes or a multiple of 8 bytes. We attribute this non-linear performance behavior to two primary architectural factors:

- **Expensive context switching**: The cumulative latency of repeated context switching between the rejection sampling and the Keccak state is too high because the total Keccak state cannot be held entirely in the Cortex-M4's limited register file.
- **Unaligned memory access**: Since the input and output lengths in Raccoon are frequently not multiples of the 4-byte or 8-byte word size, incremental SHAKE

invocations often necessitate unaligned memory accesses or shifting logic to align data with the Keccak state.

Based on the above analysis, we universally apply the one-shot approach across all sampling routines in our optimized Raccoon implementation, including uniform ($\texttt{Sample}_q$), secret/noise ($\texttt{Sample}_{u_t}$, $\texttt{Sample}_{u_w}$), and challenge (`ChalHash, ChalPoly`) sampling. By buffering inputs for a single SHAKE256 absorb and squeezing multiple blocks of output, we substantially reduce the frequency of SHAKE256 subroutine invocations. This strategy effectively mitigates the computational burden associated with frequent context switching and unaligned memory accesses, overheads that are particularly expensive on register-constrained architectures. As a result, this optimization leads to significant performance improvement for the sampling routines and overall Raccoon scheme. Detailed performance improvements can be found in Section 5.

Discussions. We emphasize that this optimization maintains full compatibility with existing test vectors and does not affect the security properties of Raccoon. The primary trade-off for this optimization is the increased memory footprint required to cache the pseudo-random bytes. However, as shown in Figure 1, processing data in chunk sizes larger than 40 bytes yields performance comparable to caching the entire stream while requiring significantly less RAM. This allows for flexible memory-performance trade-offs tailored to specific hardware constraints. Given the substantial performance gains, this optimization is highly favorable for resource-constrained environments like the Cortex-M4.

While this evaluation was conducted on the ARM Cortex-M4, the underlying performance bottlenecks are inherent to the interaction between the SHAKE logic and any resource-constrained architecture. On platforms such as RISC-V (RV32IM) or AVR (8-bit), the overhead of Keccak state management might be even more pronounced due to smaller register files and less efficient 64-bit emulation. Moreover, the applicability of this optimization extends beyond Raccoon. These findings establish a critical design and implementation principle for using incremental SHAKE256 on resource-constrained architectures: to efficiently process large volumes of data, designers must avoid frequent SHAKE invocations for small data granules, especially for chunk sizes smaller than 8 bytes.

3.3 Tailored implementations for Raccoon

Apart from the optimizations discussed earlier, we also introduce additional improvements specifically tailored for Raccoon's parameters.

Polynomial shifting. We observe that the shifting lengths in Raccoon are limited to $v_t = 42$ and $v_w = 44$. This allows us to optimize these operations explicitly for v_t and v_w. The left-shift operation by an explicit v_t or v_w can be accomplished using just two instructions, which offers a significant performance improvement over the reference implementation that uses variable shifting lengths. Similar speedups can be obtained for right-shift operations by 42 and 44 in Raccoon.

Coefficient reduction. Due to the use of lazy reduction in Raccoon, coefficient reduction is still required to reduce the coefficients of polynomials in $\mathcal{R}_q$ or in $\mathcal{R}_{q_i}$ down to a small range so that the subsequent computations will not overflow. We adopt the modular reduction presented in the reference implementation of Dilithium [BDK+21] and tailor it for the moduli q and q_i of Raccoon. For 32-bit modular reduction modulo q_i, the instruction sequence is similar to the modular reduction for Dilithium's modulus, which only takes three cycles on Cortex-M4. The instruction sequence of the 64-bit modular reduction modulo q is shown in Algorithm 8. It requires only six instructions and outputs results in $(-q, q)$. A 64-bit conditional addition by q is required to reduce it to $[0, q)$ for the following computations in Raccoon.

Algorithm 8 The 64-bit modular reduction modulo q of Raccoon

Input: The 64-bit coefficient $a = a_l + a_h \cdot 2^{32}$ with $|a| \leq 2^{63} - 2^{48} - 1$ and 64-bit modulus $q = q_l + q_h \cdot 2^{32}$.
Output: $r = a \mod q \in (-q, q)$
1: `add` $t_0, a_h, \#2^{16}$
2: `asr` $t_0, t_0, \#17$
3: `smull` t_1, t_2, t_0, q_l
4: `mla` t_2, t_0, q_h, t_2
5: `subs` a_l, t_1
6: `sbc` a_h, t_2
7: **return** $r = a_l + a_h \cdot 2^{32}$

4 Enabling high-order Raccoon on IoT devices

Raccoon provides high-order masked implementations with up to 32 shares. These implementations require splitting each secret polynomial into d-share polynomials, which can lead to significant memory consumption. As a result, high-order masked implementations are not feasible on memory-constrained devices. For instance, the reference implementations of Raccoon-128 with $d \geq 16$, Raccoon-192 with $d \geq 8$, and Raccoon-256 with $d \geq 4$ cannot run on the selected platform with 640 KiB of RAM. Therefore, memory optimizations for high-order Raccoon are essential to ensure both protection against side-channel attacks and the practicality of Raccoon on such devices.

In this section, we explore several optimizations to reduce the memory footprint of Raccoon, thereby enabling high-order variants to provide robust side-channel protection on memory-constrained devices. Because the memory consumption of the `sign` algorithm poses the primary barrier to practical deployment on such platforms, our analysis mainly focuses on optimizing `sign`. We also extend some of the techniques to the `keygen` and `verify` for certain parameter sets. While analogous optimizations have been proposed for Dilithium [GKS20, BRS22], those works do not address masked implementations. Optimizing memory in the context of masking introduces significant challenges: masked polynomials account for the vast majority of memory usage, necessitating tailored strategies to efficiently manage them.

4.1 Streaming the matrix-vector multiplication

For lower-order variants of Raccoon, we explore several memory optimizations that improve efficiency. One of the most memory-consuming operations in Raccoon is the matrix-vector multiplication with the d-share polynomial vector. In Raccoon's reference implementation, the polynomial matrix and masked vector account for a large proportion of memory during the matrix-vector multiplications. Each n-degree polynomial in $\mathcal{R}_q$ consumes 4 KiB of stack memory. The polynomial matrix $\mathbf{A}$ therefore accounts for $4kl$ KiB; the secret polynomial vector $[\![\mathbf{r}]\!]$ consumes $4ld$ KiB; and the d-share polynomial $[\![\mathbf{w}]\!]$ accounts for $4d$ KiB. The polynomial matrix and the d-share polynomial vector are the main reasons for the large stack usage at masking orders smaller than 16. Both of them need to be used twice in Algorithm 2. Consequently, time-memory trade-offs are required to enable high-order Raccoon on memory-constrained devices.

Streaming the matrix A. Instead of retaining the entire matrix, one can generate the matrix $\mathbf{A}$ on-the-fly and store only a single entry at a time, following a similar approach to [GKS20, BRS22]. This requires generating the matrix twice, in Steps 6 and 17 of Algorithm 2, but it can save 80 KiB, 140 KiB, and 252 KiB of memory for Raccoon-128,

Raccoon-192, and Raccoon-256, respectively. We apply this strategy to Raccoon-128 with $d \geq 16$, Raccoon-192 with $d \geq 8$, and Raccoon-256 with $d \geq 4$.

Streaming the d-share vector $[\![\mathbf{r}]\!]$. As d increases, the stack usage of $[\![\mathbf{r}]\!]$ and $[\![\mathbf{w}]\!]$ grows linearly. When $d > k$, the stack usage of the d-share polynomial vector $[\![\mathbf{r}]\!]$ surpasses that of $\mathbf{A}$. Consequently, streaming $[\![\mathbf{r}]\!]$ becomes essential to reduce memory consumption in Raccoon. We generate only one d-share polynomial at a time, which reduces memory usage by $4(l-1)d$ KiB. It is important to note that the `AddRepNoise` subroutine requires generating κ random bytes to sample a secret polynomial noise and add it to $[\![\mathbf{r}]\!]$ in Steps 5 and 7 of Algorithm 2. The reference implementation generates these bits using random bit generators within each iteration of the `AddRepNoise` subroutine. However, this approach does not guarantee the same random-byte sequence when we want to stream and regenerate $[\![\mathbf{r}]\!]$. To address this, we pre-generate the required random bytes and reuse them to regenerate the unmasked noise polynomial $\mathbf{r}$ in Step 14. We then utilize fresh masking randomness to refresh $[\![\mathbf{r}]\!]$ to ensure its probing security. This strategy is employed for Raccoon-128 when $d = 8, 16$, Raccoon-192 when $d = 8$, and for Raccoon-256 when $d = 4$.

Note that the randomness used in Raccoon has two distinct roles. The first type is sampler randomness, which is used to deterministically reconstruct algebraic values such as the public matrix, secret/noise polynomials. The second type is masking randomness, which is used to generate zero-sharings and refresh masked values for probing security. The randomness seed that we store for regenerating $[\![\mathbf{r}]\!]$ is used to reconstruct the same underlying unmasked noise polynomial $\mathbf{r}$. We use fresh masking randomness to generate a new d-sharing $[\![\mathbf{r}]\!]$. This is equivalent to the refresh operation in the `sign` algorithm when $[\![\mathbf{r}]\!]$ is reused: the unmasked value remains the same, while the sharing is re-randomized with fresh zero-sharing randomness. Therefore, the optimization changes does not affect the probing security argument. Furthermore, we note that storing seed material for deterministic reconstruction is a standard implementation technique as long as the seed is protected appropriately. For example, FIPS 204 [NIS24] allows the ML-DSA `keygen` seed ξ to be stored for later expansion, while requiring that it be protected like a private key.

4.2 On-the-fly serialization

In addition to the matrix-vector multiplication, the internal structure of the secret key, public key, and signature in Raccoon also contributes significantly to memory usage. The public key consists of a k-dimensional polynomial vector $\mathbf{t}$, which occupies $4k$ KiB of memory. The signature consists of a k-dimensional hint $\mathbf{h}$ and an l-dimensional vector $\mathbf{z}$, requiring a total of $4(k+l)$ KiB of memory. The d-share secret key mainly consists of a $4ld$ KiB secret polynomial vector, which will be optimized by the mask compression technique described in Subsection 4.3.

On-the-fly public key serialization. We propose to serialize the public key on-the-fly. This approach allows us to free up memory as soon as each polynomial is generated, rather than waiting for the entire public key to be constructed. By streaming the serialization of the public key, we can reduce $4k$ KiB memory usage in `keygen` and `verify`.

On-the-fly signature serialization. Similarly, we can serialize the signature on-the-fly during `sign`. However, when we tried to stream the signature serialization, we find that the sequence of Raccoon signature steps is not suitable for on-the-fly serialization due to the variable length of $\mathbf{h}$. The hint $\mathbf{h}$ is encoded before $\mathbf{z}$ but it is computed after $\mathbf{z}$. Without knowing the actual index of $\mathbf{z}$ in the signature bytes, it would be difficult to serialize them on-the-fly. Therefore, we propose to encode $\mathbf{z}$ before $\mathbf{h}$, which is the same order as they are computed in `sign`. Even though this change modifies the signature format, it does

not affect the security of Raccoon. The verifier can still decode $\mathbf{z}$ and $\mathbf{h}$ correctly, as they are encoded independently. This modification allows for on-the-fly serialization of the signature, reducing $4(k+l)$ KiB of memory consumption in `sign` and `verify`.

4.3 Mask compression

The primary challenge in deploying high-order Raccoon on memory-constrained devices is the substantial memory footprint of the d-share polynomial vectors. Because the vectors within the secret key are essential for maintaining the scheme's side-channel security, any memory optimization must strictly preserve their side-channel security. To mitigate this, Saarinen and Rossi [SR23] introduced (Strong)-Non Interference secure (NI/SNI-secure) mask compression techniques to reduce the memory consumption of the masked implementation. They applied the NI-secure mask compression technique to a preliminary version of Raccoon on FPGA. However, its efficacy on resource-constrained IoT platforms remains largely unexplored. In this work, we also apply the NI-secure mask compression algorithms ([SR23, Algorithm 1,2,6]) and evaluate their performance on ARM Cortex-M4.

The mask compression technique condenses a d-share polynomial into a single polynomial and $d-1$ κ-bit cryptographic seeds, where $\kappa \in \{128, 192, 256\}$ for Raccoon-128, Raccoon-192, and Raccoon-256, respectively. Each seed corresponds to one subsequent $d-1$ shares. Only one single share of the compressed polynomial will be regenerated on-the-fly during computations, yielding significant memory savings. Specifically, one masked polynomial originally requiring $4d$ KiB can be compressed down to just 4 KiB, plus $(d-1)\kappa/8$ bytes for the seeds, which is a significant reduction at high masking order.

By employing this technique in `sign` and to ensure compatibility with the Raccoon reference implementation, we need to maintain one single d-share masked polynomial for the intermediate computations in masked gadgets, while storing the polynomial vectors $[\![\mathbf{s}]\!]$ and $[\![\mathbf{r}]\!]$ in Algorithm 2 in their compressed forms. This significantly reduces the memory footprint and enables the deployment of high-order Raccoon on IoT devices. However, this memory reduction incurs significant computational penalty due to the on-the-fly regeneration and refreshing of the masked polynomials. Consequently, we selectively apply this technique only to parameter sets that would otherwise exceed memory limits despite our prior memory optimizations: specifically, Raccoon-128 with $d = 32$, Raccoon-192 with $d \geq 16$, and Raccoon-256 with $d \geq 16$, which cannot be deployed with the aforementioned memory optimizations. One can also apply this technique to other parameter sets to further reduce memory usage, albeit with a performance trade-off.

5 Results and comparisons

The experimental setup is the same as the pqm4 [KRSS19]. All experiments are compiled using arm-none-eabi-gcc version 10.3.1 with the `-O3` optimization level enabled. We do not enable link-time optimization (`-flto`), since preliminary evaluation shows that it can degrade the performance of critical operations, such as NTT and INTT. This issue has been previously discussed in pqm4[4], where `-flto` is likewise not enabled by default due to its negative impact on the performance of certain cryptographic primitives.

The target Cortex-M4 board is the latest platform supported in pqm4, the NUCLEO_L4R5 board, with 2 MB of flash and 640 KiB of RAM. We use the optimized Keccak implementation presented in [HAZ+24] for benchmarking the proposed assembly implementations. The baseline is the reference C implementation of Raccoon provided by the Raccoon team [PEK+23]. The masked random number generator we used is the fast LFSR-based generator provided in the reference implementation. The leakage evaluation of Raccoon is conducted on a CW308T-STM32F4 with an ARM Cortex-M4 processor.

[4] https://github.com/mupq/pqm4/pull/36

Table 1: Cycle counts (cc) of the polynomial arithmetic of Raccoon-128 on Cortex-M4.

	`Split`	`Join`	`NTT`	`INTT`	`LeftShift`	`RightShift`	`Add`	`Addq`[a]
Ref. [PEK+23]	9 795	22 613	117 423	175 778	12 371	14 420	7 236	10 835
This work	5 718	13 908	41 671	48 146	3 801	4 942	5 970	9 047
Ref/This work	1.71×	1.63×	2.82×	3.65×	3.25×	2.93×	1.21×	1.20×

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

Table 2: Cycle counts (cc) of the polynomial sampling of Raccoon-128 on Cortex-M4.

Sampling type	$\texttt{Sample}_q$	$\texttt{Sample}_{u_t}$	$\texttt{Sample}_{u_w}$	`ChalHash`	`ChalPoly`
Ref. [PEK+23]	673 184	143 513	562 787	627 699	29 363
This work	312 928	59 212	261 460	235 439	16 282
Ref./This work	2.15×	2.42×	2.15×	2.67×	1.80×

5.1 Performance of polynomial arithmetic and sampling

Table 1 presents the performance results of the polynomial arithmetic for Raccoon-128 on the Cortex-M4. Compared to the reference implementation in [PEK+23], we achieve a comprehensive improvement in the performance of polynomial arithmetic. The polynomial splitting operation involves the double modular reductions discussed in Subsubsection 3.1.2. As shown, the proposed double modular reductions in Algorithm 7 outperform the implementation based on the state-of-the-art Montgomery reduction in [GKS20] by factors of 1.71×. The polynomial joining operation, which includes the CRT computation, outperforms the reference implementation by a factor of 1.63×. The performance improvements for NTT and INTT are even more significant, with speedups of 2.82× and 3.65×, respectively, because we thoroughly leverage the layer merging and lazy reduction strategies to enhance efficiency. We also conducted an experiment on the NTT implementation using Barrett multiplication from [BHK+22] on Cortex-M4. Although Barrett multiplication by a twiddle factor can be implemented with three instructions on Cortex-M4 (the same as Montgomery multiplication) and no additional modular reductions are required in the NTT implementation, the resulting NTT implementation requires 47 607 cycles, which is 14.23% slower than the corresponding implementation using Montgomery multiplication. For each 32-bit NTT modulo q_1 or q_2, the Barrett-based NTT implementation requires 20 718 cycles, whereas the Montgomery-based implementation requires only 18 031 cycles. This slowdown is mainly caused by the additional memory-access overhead introduced by the doubled twiddle-factor table, demonstrating the performance advantage of Montgomery arithmetic on the ARM Cortex-M4. The 64-bit left-shift and right-shift operations are 3.25× and 2.93× faster, thanks to tailored implementations for Raccoon's parameters. The assembly implementation of polynomial addition and subtraction is also slightly faster than the reference implementation.

The performance of the polynomial sampling routines is detailed in Table 2. Our results demonstrate that optimizing the sampling process yields significant computational speedups across all variants. Specifically, both uniform sampling and secret/noise sampling achieve a performance increase of approximately 2×, while the challenge sampling routines `ChalHash` and `ChalPoly` are 2.67× and 1.80× faster than the reference implementation. We further show that these speedups transfer to matrix generation, masking gadgets, and the Raccoon scheme.

5.2 Performance of masking components and matrix generation

Table 3 provides a comparative performance analysis of the core operations within Raccoon, specifically focusing on the four primary masking components, secret key serialization, and matrix generation. The performance gains for ZeroEncoding, Refresh, and Decode are relatively modest, as these routines primarily benefit from the enhancements in polynomial

Table 3: Cycle counts (cc) of the masking gadgets, `SKEncode`, and matrix generation of Raccoon-128 on Cortex-M4. Note that the reported entries for matrix generation correspond to the unmasked operation, and thus only the case $d = 1$ is considered.

		ZeroEncoding	AddRepNoise	Refresh	Decode	SKEecode	MatrixGen
$d = 1$	Ref. [PEK+23]	3643	4592k	54	7226	281k	15547k
	This work	3642	2166k	53	7227	168k	6962k
$d = 2$	Ref. [PEK+23]	19375	4755k	40912	10836	3122k	-
	This work	16167	2303k	34169	9046	1533k	-
$d = 4$	Ref. [PEK+23]	100744	4879k	143992	32422	8813k	-
	This work	86285	2411k	122331	27055	4268k	-
$d = 8$	Ref. [PEK+23]	325951	17187k	412358	75588	20200k	-
	This work	280562	8732k	352616	63066	9731k	-
$d = 16$	Ref. [PEK+23]	900507	17708k	1087332	161901	43074k	-
	This work	777047	9174k	926768	135065	20684k	-

arithmetic. In contrast, AddRepNoise and secret key serialization, both of which rely heavily on polynomial sampling in addition to arithmetic, exhibit substantial improvements, achieving a speedup of approximately 2× over the reference implementation. Furthermore, matrix generation significantly benefits from the assembly Keccak implementation in [HAZ+24] and our optimized one-shot sampling strategy, resulting in a speedup exceeding 2.23×.

Table 4: Cycle counts (cc) and stack usage (bytes) of `keygen`, `sign`, and `verify` of Raccoon on Cortex-M4. For implementations without mask compression, cycle counts were obtained by averaging 100 executions. Conversely, variants with mask compression were evaluated using an average of only two executions. Furthermore, because stack usage is constant, it was measured using a single execution.

	Implementation	Raccoon-128			Raccoon-192			Raccoon-256		
		keygen	sign	verify	keygen	sign	verify	keygen	sign	verify
$d = 1$	Ref.[PEK+23]	28931k[a] 83232	65251k 230752	21730k 111960	45289k 107864	93892k 290815	35662k 144800	73480k 140704	123262k 505320	60481k 185832
	This work	12859k 86224	31044k 233728	10785k 114952	20178k 110856	44489k 336192	17352k 147752	32827k 143680	62913k 509188	29001k 189724
$d = 2$	Ref.[PEK+23]	35103k 112008	72257k 284064	21729k 111960	53488k 140744	103185k 394720	35665k 144800	84988k 181660	135412k 583208	60464k 185832
	This work	16151k 111772	34867k 286528	10788k 114952	24545k 140508	49479k 389472	17347k 147792	38982k 181640	69570k 578844	29008k 189724
$d = 4$	Ref.[PEK+23]	45905k 164328	84595k 377292	21736k 111944	67797k 201180	108393k 504332	35655k 144800	- -	- -	- -
	This work	21502k 164256	41365k 360872	10792k 114952	31631k 201184	56124k 487912	17343k 147792	49061k 192968	273524k 312056	28899k 87364
$d = 8$	Ref.[PEK+23]	111145k 262636	198379k 299007	21736k 111960	- -	- -	- -	- -	- -	- -
	This work	55758k 262640	106503k 557720	10789k 114952	77259k 266856	488002k 407296	17187k 70008	111297k 340608	786728k 504648	28901k 87364
$d = 16$	Ref.[PEK+23]	- -	- -	- -	- -	- -	- -	- -	- -	- -
	This work	78391k 422512	342402k 511999	10722k 57592	318124k 177432	7441782k 256320	17209k 70008	525817k 188104	12685267k 236300	28912k 87364
$d = 32$	Ref.[PEK+23]	- -	- -	- -	- -	- -	- -	- -	- -	- -
	This work	659222k 303920	18535522k 365672	10607k 57648	1055217k 310672	29705278k 342588	17194k 70008	1762570k 323008	48356318k 337676	28925k 87364

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

5.3 Performance of schemes

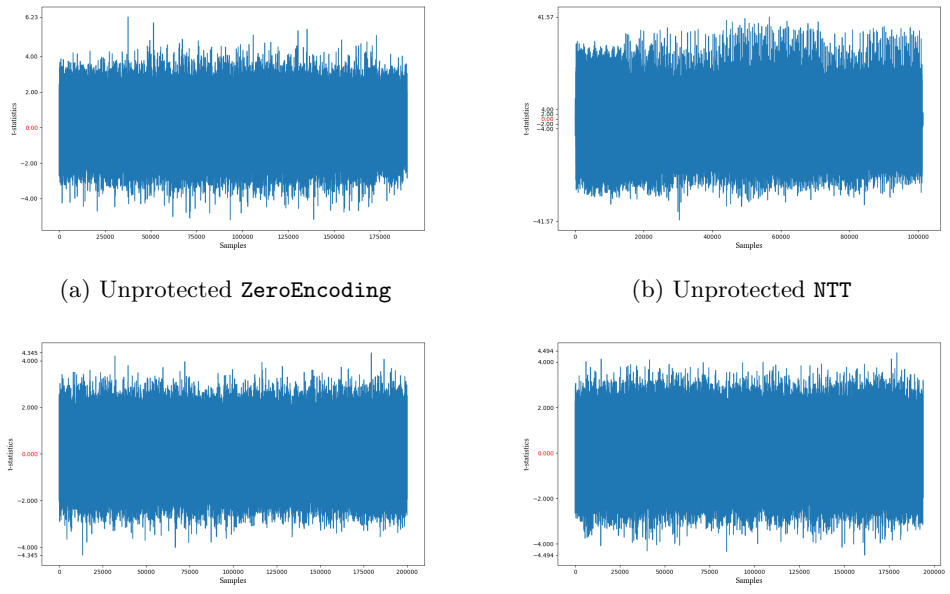
The cycle counts and stack usages of Raccoon are shown in Table 4. Combining all the aforementioned optimizations, the proposed assembly implementations are 1.86×–2.25× faster than Raccoon's reference implementation. Analysis of the stack usage for Raccoon-128 with $d = 1$ indicates that our optimized polynomial sampling routine requires an additional 3 KiB of memory to buffer pseudo-random bytes. Given that this results in a significant increase in performance, we consider this modest increase in stack a highly

Table 5: Comparison of high-order masked signature generation of Dilithium3 and Raccoon-192 on Cortex-M4.

Implementation	$d = 1$	$d = 2$	$d = 4$	$d = 8$
Raccoon-192	44489k	49479k	56124k	488002k
Dilithium3 [ABC+23]	6523k[a]	154299k	453455k	1324689k
Dilithium3/Raccoon-192	0.15×	3.11×	8.07×	2.71×

a: Unmasked Dilithium3 implementation presented in [HAZ+24].

favorable trade-off for resource-constrained environments where execution time is a critical bottleneck. Moreover, the memory optimizations in Subsection 4.1 enable the usage of high-order Raccoon, even with $d = 32$, on the selected platform. However, these optimizations also introduce some performance overhead, resulting in slower execution for high-order Raccoon. Note that the stack usage of low-order Raccoon in Table 4 is not the minimal number because we aim for better performance. Applying the memory optimizations in Subsection 4.1 to Raccoon-128 with $d = 1$ further reduces the stack usage to 44 512, 99 000, and 57 648 bytes for `keygen`, `sign`, and `verify`, respectively, yielding memory reductions of 46.52%, 56.10%, and 48.51% compared to the reference implementation. Furthermore, applying on-the-fly serialization for the public key and signature to the high-order Raccoon `verify` reduces the stack usage of `verify` by around 50% without introducing any additional performance penalty.



(a) Unprotected `ZeroEncoding`

(b) Unprotected `NTT`

(c) `ZeroEncoding`

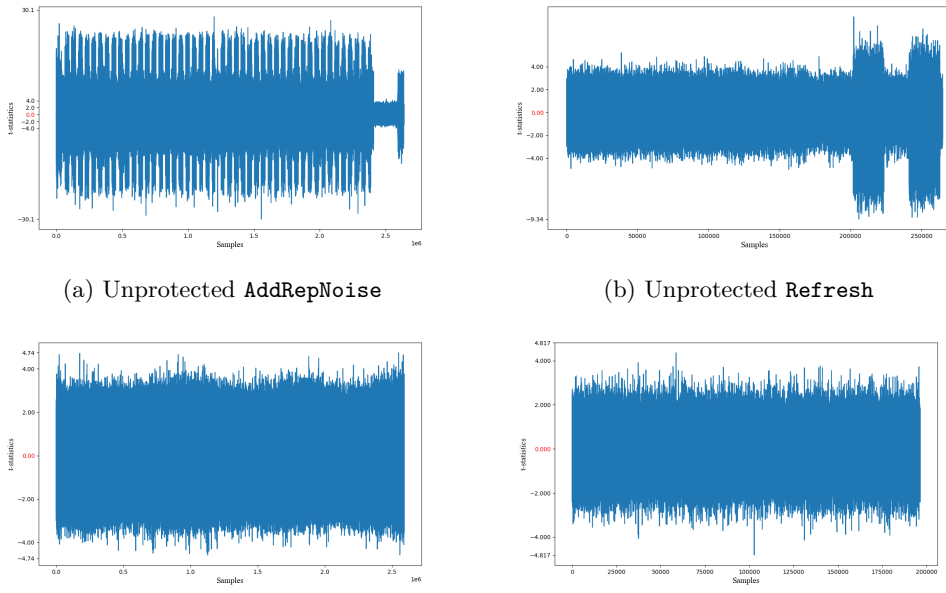
(d) `NTT`

Figure 2: Leakage assessment of `ZeroEncoding` and `NTT`. Figures (a) and (b) show the unprotected versions with random numbers disabled. The pointwise two-sided p-values for Figures (c) and (d) are 1.39×10^{-5} and 6.99×10^{-6}, respectively. Following the multiple-time-sample formula of [DZD+17], the corresponding critical value for both figures is $C = 6.57$.

5.4 Comparison with masked Dilithium

Since Raccoon closely resembles Dilithium, Table 5 compares the performance of Raccoon-192 with the state-of-the-art high-order masked (deterministic) Dilithium3 [ABC+23] on the ARM Cortex-M4. The results in [ABC+23] were obtained without accounting for the signature generation repetitions required by Dilithium3. However, due to the rejection sampling procedure, Dilithium3's signature generation typically requires approximately 5.1 repetitions [BDK+21, Table 2]. To ensure a fair comparison, we scale their results by a factor of 5.1 and compare them with Raccoon-192.

As shown in Table 5, Raccoon-192 is 3.11×, 8.07×, and 2.71× faster than masked Dilithium3 for masking orders $d = 2, 4, 8$, respectively. While the unmasked Raccoon implementation is significantly slower than the unmasked Dilithium implementation, it is vulnerable to side-channel attacks and, therefore, insecure for certain use cases. Moreover, we observe that as the masking order increases from 1 to 4, the performance of Dilithium deteriorates by a factor of 69.52×, while Raccoon with $d = 4$ experiences only a 1.26× slowdown compared to Raccoon with $d = 1$. This demonstrates that the masking overhead in Raccoon increases linearly with the masking order, making it more efficient in higher-order masking scenarios. The smaller speedup at $d = 8$ is attributed to memory optimizations that slightly reduce Raccoon's performance. These results highlight Raccoon's superior efficiency in masked implementations compared to Dilithium, making it a promising lattice-based scheme for side-channel-resistant cryptographic applications.



(a) Unprotected `AddRepNoise` (b) Unprotected `Refresh`

(c) `AddRepNoise` (d) `Refresh`

Figure 3: Leakage assessment of `AddRepNoise` and `Refresh`. Figures (a) and (b) show the unprotected versions with random numbers disabled. The pointwise two-sided p-values for Figures (c) and (d) are 2.14×10^{-6} and 1.46×10^{-6}, respectively. Following the multiple-time-sample formula of [DZD+17], the critical values for these two figures are $C = 6.94$ and $C = 6.57$, respectively.

5.5 Leakage assessment of Raccoon

To evaluate the side-channel security of Raccoon on the Cortex-M4, we followed the test procedure specified in ISO 17825:2022 [ISO23] and conducted leakage assessments on the key components of Raccoon's `sign` with $d = 2$ using the CW308T-STM32F4 platform. Power signals were collected via connectors on the CW308 board using a PicoScope 5244D oscilloscope, with the tests executed at a clock frequency of 16.624 MHz.

Due to the limited resources and slow performance of Raccoon on the Cortex-M4, we were unable to perform a leakage assessment for the entire Raccoon signature. Instead, we focused on evaluating the key operations involving secret information, $[\![\mathbf{r}]\!]$ and $[\![\mathbf{s}]\!]$, in Raccoon's `sign`. The random-vs-fixed test was employed to detect any potential side-channel leakage of $[\![\mathbf{r}]\!]$ or $[\![\mathbf{s}]\!]$. Due to the page limit, we present leakage assessments of four components in Figure 2 and Figure 3 as an illustration. The leakage assessment of the masked version was performed with $N = 200{,}000$ traces, while the version with random numbers disabled was evaluated using a smaller number of traces to verify that the leakage detection was operational. As shown in Figure 2(a,b) and Figure 3(a,b), the unprotected versions of these components exhibit significant leakage when random numbers are disabled, demonstrating the effectiveness of the masked implementation.

Despite being designed to be masking-friendly, the software implementation of masked Raccoon on real-world architectures may still be susceptible to DPA attacks, primarily due to the (micro-)architectural overwriting issue described in [SSB+21, CPW24]. We mitigated this issue by carefully reordering instructions and inserting dummy operations to clear sensitive registers, effectively preventing transitional leakage. After applying these countermeasures, we conducted additional leakage assessments on the key operations in Raccoon. The test values in Figure 2(c,d) and Figure 3(c,d) remain below the critical thresholds derived from the multiple-time-sample fomula of [DZD+17], indicating that no leakage was detected after addressing the (micro-)architectural overwriting issue. A comprehensive leakage assessment of the entire Raccoon signature generation process is left for future work, as it requires a significantly larger number of traces and more careful effort to address the (micro-)architectural overwriting issue.

6 Conclusion

This paper proposes the first efficient and compact implementation of high-order Raccoon on the ARM Cortex-M4. The proposed optimizations include efficient NTT/INTT implementations, novel double modular reductions, polynomial sampling, and optimized operations tailored for Raccoon's parameters, resulting in significant improvements in its efficiency. The memory optimizations for high-order Raccoon were also explored, addressing the challenges posed by memory-constrained platforms. These optimizations enhance the practicality of high-order Raccoon, providing robust side-channel resistance on memory-constrained platforms. Finally, leakage assessment of Raccoon's components on real-world architectures was conducted to demonstrate its resistance to DPA attacks.

Acknowledgments

This work is supported in part by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (24-SIS-SMU-052), Scientific Research Innovation Capability Support Project for Young Faculty (SRICSPYF-BS2025137), Guangdong Provincial Key Laboratory of IRADS (2022B1212010006), Guangdong and Hong Kong Universities "1+1+1" Joint Research Collaboration Scheme (2025A0505000001), Guangdong Basic and Applied Basic Research Foundation (2024A1515011274), National Natural Science Foundation of China (62372273).

References

[AB74] Ramesh C. Agarwal and Charles S. 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.

[ABB+23] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, and Jean-Pierre Tillich. PERK. 2023.

[ABC+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):58–79, Aug. 2023.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):127–151, 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 853–871. Springer, 2022.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP Lattice-Based Signature Scheme at Any Order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 2018.

[BBE+19] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. GALACTICS: Gaussian Sampling for Lattice-Based Constant- Time Implementation of Cryptographic Signatures, Revisited. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2147–2164. ACM, 2019.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium Algorithm Specifications And Supporting Documentation (Version 3.1). Technical report,

National Institute of Standards and Technology Post-Quantum Cryptography, 2021.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):221–244, 2022.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS$^+$ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology - INDOCRYPT 2022 - 23rd International Conference on Cryptology in India, Kolkata, India, December 11-14, 2022, Proceedings*, Lecture Notes in Computer Science, pages 272–293. Springer, 2022.

[BPT+25] Pierre-Augustin Berthet, Justine Paillet, Cédric Tavernier, Brice Colombier, and Lilian Bossuet. Masked Computation of the Floor Function and Its Application to the FALCON Signature. *IACR Communications in Cryptology*, 1(4):1–23, 2025.

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for Memory Constrained Devices. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18-20, 2022, Proceedings*, volume 13503 of *Lecture Notes in Computer Science*, pages 217–235. Springer Nature Switzerland, 2022.

[CC24] Keng-Yu Chen and Jiun-Peng Chen. Masking Floating-Point Number Multiplication and Addition of Falcon: First- and Higher-Order Implementations and Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):276–303, 2024.

[CGL+24] Jean-Sébastien Coron, François Gérard, Tancrède Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved High-Order Masked Generation of Masking Vector and Rejection Sampling in Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):335–354, 2024.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved Gadgets for the High-Order Masking of Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4), 2023.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[CPW24] Hao Cheng, Daniel Page, and Weijia Wang. eLIMInate: a Leakage-focused ISE for Masked Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):329–358, 2024.

[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.

[DPKN+25] Rafaël Del Pino, Shuichi Katsumata, Guilhem Niot, Michael Reichle, and Kaoru Takemure. Unmasking TRaccoon: A lattice-based threshold signature with an efficient identifiable abort protocol. In *Annual International Cryptology Conference*, pages 423–456. Springer, 2025.

[dPKPR24] Rafaël del Pino, Shuichi Katsumata, Thomas Prest, and Mélissa Rossi. Raccoon: A masking-friendly signature proven in the probing model. In *Annual International Cryptology Conference*, pages 409–444. Springer, 2024.

[dPPRS23] Rafaël del Pino, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. High-Order Masking of Lattice Signatures in Quasilinear Time. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1168–1185. IEEE, 2023.

[DPPvW22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. Hawk: Module LIP Makes Lattice Signatures Fast, Compact and Simple. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 65–94. Springer, 2022.

[DZD+17] Aidong Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards Sound and Optimal Leakage Detection Procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[EFG+22] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A Simpler, Parallelizable, Maskable Variant of Falcon. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 222–253. Springer, 2022.

[FDK20] Apostolos P. Fournaris, Charis Dimopoulos, and Odysseas G. Koufopavlou. Profiling Dilithium Digital Signature Traces for Correlation Differential Side Channel Attacks. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation - 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings*, volume 12471 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2020.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, Dec. 2020.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The Hidden Parallelepiped Is Back Again: Power Analysis Attacks on Falcon. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):141–164, 2022.

[Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[GR19] François Gérard and Mélissa Rossi. An Efficient and Provable Masked Implementation of qTESLA. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *Lecture Notes in Computer Science*, pages 74–91. Springer, 2019.

[GS66] W. Morven Gentleman and Gordon Sande. Fast Fourier Transforms: for fun and profit. volume 29 of *AFIPS Conference Proceedings*, pages 563–578, 1966.

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):1–24, 2024.

[HP23] Daniel Heinz and Thomas Pöppelmann. Combined Fault and DPA Protection for Lattice-Based Cryptography. *IEEE Trans. Computers*, 72(4):1055–1066, 2023.

[ISO23] ISO. Information technology - security techniques - testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Draft International Standard ISO/IEC DIS 17825:2022(E), 2023.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KA21] Emre Karabulut and Aydin Aysu. FALCON Down: Breaking FALCON Post-Quantum Signature Scheme through Side-Channel Attacks. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 691–696. IEEE, 2021.

[KA24] Emre Karabulut and Aydin Aysu. Masking FALCON's Floating-Point Multiplication in Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):483–508, 2024.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-Trace Side-Channel Attacks on ω-Small Polynomial Sampling: With Applications to NTRU, NTRU Prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pages 35–45. IEEE, 2021.

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. *NIST special publication*, 800:185, 2016.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. IACR Cryptol. ePrint Arch., Paper 2019/844, 2019. https://eprint.iacr.org/2019/844.

[LCH+24] Jooyoung Lee, Jihoon Cho, Jincheol Ha, Jihoon Kwon, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Sangyub Lee, Dukjae Moon, Mincheol Son, and Hyojin Yoon. The AIMer Signature Scheme. Technical report, National Institute of Standards and Technology Post-Quantum Cryptography Additional Digital Signatures, 2024. Version 2.1.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - Efficient Implementation and Side-Channel Evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, volume 11464 of *Lecture Notes in Computer Science*, pages 344–362. Springer, 2019.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[NIS24] NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/final, 2024.

[PEK+23] Rafael del Pino, Thomas Espitau, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. Raccoon: A Side-Channel Secure Signature Scheme. https://raccoonfamily.org/wp-content/uploads/2023/07/raccoon.pdf, 2023. Accessed: 2024-11-23.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch.*, page 39, 2018.

[SR23] Markku-Juhani O Saarinen and Mélissa Rossi. Mask compression: High-order masking on memory-constrained devices. In *International Conference on Selected Areas in Cryptography*, pages 65–81. Springer, 2023.

[SSB+21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[UMTS22] Vincent Quentin Ulitzsch, Soundes Marzougui, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling Side-Channel Attacks on Dilithium - A Small Bit-Fiddling Leak Breaks It All. In Benjamin Smith and Huapeng Wu, editors, *Selected Areas in Cryptography - 29th International Conference, SAC 2022, Windsor, ON, Canada, August 24-26, 2022, Revised Selected Papers*, volume 13742 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2022.

[ZLYW23a] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved Power Analysis Attacks on Falcon. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 565–595. Springer, 2023.

[ZLYW23b] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved Power Analysis Attacks on Falcon. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 565–595. Springer, 2023.

[ZYHK25] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):to appear, 2025.

[ZZ26] Jipeng Zhang and Jiaheng Zhang. Vectorized Falcon-Sign implementations using SSE2, AVX2, AVX-512F, NEON, and RVV. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2026(1), 2026.