IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 23–68. DOI:10.46586/tches.v2025.i2.23-68

MulLeak: Exploiting Multiply Instruction Leakage to Attack the Stack-optimized Kyber Implementation on Cortex-M4

Fan Huang[1], Xiaolin Duan[1], Chengcong Hu[1], Mengce Zheng[3] and Honggang Hu[1,2](✉)

[1] School of Cyber Science and Technology, University of Science and Technology of China, Hefei, China
{fanh2022,duanxl,hcc2016}@mail.ustc.edu.cn
[2] Hefei National Laboratory, Hefei, China hghu2005@ustc.edu.cn
[3] Zhejiang Wanli University, Ningbo, China mczheng@zwu.edu.cn

Abstract. CRYSTALS-Kyber, one of the NIST PQC standardization schemes, has garnered considerable attention from researchers in recent years for its side-channel security. Various targets have been explored in previous studies; however, research on extracting secret information from stack-optimized implementations targeting the Cortex-M4 remains scarce, primarily due to the lack of memory access operations, which increases the difficulty of attacks.

This paper shifts the focus to the leakage of multiply instructions and present a novel cycle-level regression-based leakage model for the following attacks. We target the polynomial multiplications in decryption process of the stack-optimized implementation targeting the Cortex-M4, and propose two regression-based profiled attacks leveraging known ciphertext and chosen ciphertext methodologies to recover the secret coefficients individually. The later one can also be extended to the protected implementation.

Our practical evaluation, conducted on the stack-optimized Kyber-768 implementation from the *pqm4* repository, demonstrates the effectiveness of the proposed attacks. Focusing on the leakage from the pair-pointwise multiplication, specifically the macro `doublebasemul_frombytes_asm`, we successfully recover all secret coefficients with a success rate exceeding 95% using a modest number of traces for each attack. This research underscores the potential vulnerabilities in PQC implementations against side-channel attacks and contributes to the ongoing discourse on the physical security of cryptographic algorithms.

Keywords: Post-quantum Cryptography · Kyber · Linear Regression · Profiled Attack · Cycle-level Power Leakage

1 Introduction

The National Institute of Standards and Technology (NIST) has very recently announced approval of three Federal Information Processing Standards (FIPS) for Post-Quantum Cryptography [NIS24b]: FIPS 203, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, FIPS 204, *Module-Lattice-Based Digital Signature Standard* and FIPS 205, *Stateless Hash-Based Digital Signature Standard*. Of them, FIPS 203 [NIS24a] is derived from the CRYSTALS-Kyber [SAB+22] submission which is the sole candidate for the Key Encapsulation Mechanism (KEM) schemes selected for standardization after the third round of the Post-Quantum Cryptography (PQC) standardization process. As PQC will be

Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

deployed on a wide range of computing platforms, it is extremely important that candidates are also scrutinized for their resistance against physical cryptanalysis, e.g., Side-Channel Analysis (SCA).

Since Kocher et al. [KJJ99] proposed the Differential Power Analysis (DPA) attack in 1999 to reveal the secret key of DES through power consumption, SCA has become an important approach of cryptoanalysis. SCA attacks are the ones that target the weaknesses in implementations rather than algorithm specifications, by collecting side information such as power consumption [KJJ99, KJJR11], electromagnetic radiation [AARR03], or running time [Koc96] that leaks sensitive (intermediate) information during the execution of the targeted cryptographic operation. One class of side-channel attacks known as Correlation Power Analysis (CPA) attack was introduced by Brier et al. [BCO04] in 2004. In this case, attackers model the power consumption of the device under test and measures the correlation of the model with real-world data to test secret value hypotheses. The power leakage of the device/implementation is often modeled with the Hamming Weight (HW)/Hamming Distance (HD) of/between the intermediate data. The significance of side-channel security in PQC schemes, especially for implementations on embedded devices, cannot be overstated. In the case of Kyber, several SCA or SCA-assisted attacks have emerged, underscoring its vulnerabilities [QCZ+21, XPR+22, RBRC22, RRD+23, DNGW23, MKK+23].

The polynomial multiplication is the core operation for practical constructions of lattice-based cryptography. Number Theoretic Transform (NTT) (including its inverse) and pointwise multiplication are used to implement the multiplication of polynomials in Kyber. A technique referred as incomplete NTT is introduced to handle rings of special structures as well as to improve efficiency in implementations of Kyber. It leads to the polynomial multiplications of *degree one*, which we refer to as *pair-pointwise multiplication*, following the terminology in [ABB+24]. Our study targets the *pair-pointwise multiplication* in the decryption processing of Kyber specifically.

Many research teams utilize the ARM Cortex-M4 device [ARMa] for PQC algorithm implementations, primarily due to its balanced performance and power efficiency. The *pqm4* repository [KRSS] leverages the Cortex-M4's features, including its memory and processing capabilities, to provide a robust platform for PQC algorithms. Within the *pqm4* repository, the speed-optimized and stack-optimized implementations are two optimized Kyber implementations with distinct goals: speed and stack efficiency. They have optimized operations such as the NTT (including its inverse), pair-pointwise multiplication, and modular reductions (Montgomery reduction and Barrett reduction) using assembly code. For instance, both schemes store two polynomial coefficients in a 32-bit word and then utilize vectorized instructions like `uadd16` to execute two half-word additions concurrently.

In the specific function of pair-pointwise multiplication, the two implementations adopt different strategies, as shown in Figure 16. The speed-optimized implementation employs a *lazy reduction* strategy to minimize the number of modular reduction operations, thus enhancing the speed. This approach provides a store instruction for each result coefficient that has not yet been reduced by modulo q (cf. line 9, 11 and line 20, 22 in Figure 16a). On the other hand, the stack-optimized implementation uses a *register allocation* strategy that reduces the number of load/store instructions. This allows two result coefficients that have been reduced by modulo q to be stored in a single 32-bit word, requiring only one store instruction to save both coefficients into memory (cf. line 15 and 30 in Figure 16b).

1.1 Related Works

Table 1 summarizes the related side-channel attacks against Kyber from the literature. Primas et al. [PPM17] present a notable study by combining the side-channel leakage of NTT computation with the belief propagation algorithm to conduct a single-trace profiled attack. The attack from [PP19] can be seen as a significant improvement over

Table 1: Related attacks on Kyber polynomial multiplication. LR represents the linear regression-based leakage model. HW denotes the (noisy) Hamming weight model.

Attacks	Class	Implementation(s)	Model	Target Operation
This work	Profiled	M4 stack (Masked)	LR	Pair-pointwise multiplication
[TS24]	Non-profiled	M4 speed (Masked)	HW	Pair-pointwise multiplication
[MWK+24]	Non-profiled	M4 stack	HW	Pair-pointwise multiplication
[ABB+24]	Profiled	M4 [SAB+] (Masked [HKL+, HKL+22]/Shuffled)	HW	Pair-pointwise multiplication
[HSST23]	Profiled+BP	M4 Simulation (Shuffled)	HW	NTT
[HHP+21]	Profiled+BP	M4 Simulation (Masked)	HW	NTT
[PP19]	Profiled+BP	M4 [SAB+] (Masked)	HW	NTT
[PPM17]	Profiled+BP	R-LWE Encryption [dCRVV15] (Masked [RRd+16])	HW	NTT

[PPM17] in terms of practicality. Hamburg et al. [HHP+21] present an attack using a chosen ciphertext that is decompressed to a vector containing a large amount of zeros. The attack is likewise assisted by belief propagation for template matching. In response to the shuffling countermeasures proposed by Ravi et al. [RPBC20], Hermelink et al. [HSST23] provide analysis of the shuffled NTT and give a Belief Propagation (BP) based attack. Bock et al. [ABB+24] proposed a template attack that allows attackers to reveal the secret coefficients in Kyber directly from the pair-pointwise multiplication in the decapsulation process. Tosun and Savas [TS24] present several non-profiled side-channel attacks targeting the pair-pointwise multiplication exploiting the zero-valued coefficients of the known operand, with the application to the ARM Cortex-M4 speed-optimized implementation of Kyber [impa]. Mujdei et al. [MWK+24] attack the ARM Cortex-M4 stack-optimized implementation of Kyber [impb] and show that the secret coefficients must be predicted in pairs since the incomplete NTT algorithm is used in the polynomial multiplication of the targeted implementation.

1.2 Motivation

Previous work has predominantly utilized the HW model to simulate the power leakage of intermediate or hypothetical values (as shown in Table 1). Moreover, most of these studies opt for memory operations, assuming they yield greater signal-to-noise ratios (SNR) in power leakage than register updates or combinational logic. As previously mentioned, there are two types of Kyber implementations in the *pqm4* repository: speed-optimization and stack-optimization. When attacking these implementations using the HW model with memory operation configurations, there is a significant gap in attack complexity.

In the speed-optimized implementation, each result coefficient is stored using a individual store instruction (cf. line 9, 11 or line 20, 22 in Figure 16a). Thus, it allows attackers to obtain information about individual result coefficients. For example, by employing a zero-value attack [TS24] or a CPA attack, attackers can recover the coefficients of a single secret polynomial one by one leading to an attack complexity of $O(q(n/2))$ or $O(qn)$. Conversely, the stack-optimized implementation uses only one instruction to store a pair of result coefficients (cf. line 15 or 30 in Figure 16b). It means that while an adversary may be able to obtain the Hamming weight information of this 32-bit word, they cannot discern the Hamming weight of each individual 16-bit coefficient. [MWK+24] presents a CPA attack that predicts secret coefficients in pairs, bringing the attack complexity to $O(q^2(n/2))$. Particularly, when targeting a first-order masked implementation, the attack complexity can be notably high, even reaching up to $O(q^4(n/2))$. Overall, it is more challenging for adversaries to attack the stack-optimized implementation than the speed-optimized implementation when targeting the pair-pointwise multiplication.

To overcome the difficulties when targeting the stack-optimized implementation, we shift our focus to the leakage from arithmetic instructions, particularly those pertaining to multiplication. The presence of modular reduction operations in the pair-pointwise

multiplication brings a series of multiply instructions related to the secret coefficients. Accurately modeling multiply instructions is one of the core endeavors of this paper.

McCann et al. [MOW17] conduct a modeling study focused on the instructions of Cortex-M0 and M4, including arithmetic, logical, and memory access instructions. They cluster similar instructions and terms related to instruction interactions with data states, transitions, and interactions. Then, they construct a regression-based leakage model. We expand upon their leakage model by additionally utilizing the power leakage originating from registers and combinational logic over an entire clock cycle. Then, we construct a novel cycle-level regression-based leakage model which is designed to characterize the power leakage during the execute stage of multiply instructions.

Exploiting the leakage from multiply instructions, we are able to predict the secret coefficients individually. This is another core focus of this paper and the source of the word "MulLeak" in title ("Multiply"+"Leak"). The approach yields an attack complexity of $O(qn)$ for recovering a single polynomial in an unprotected implementation. In a first-order masked implementation, the complexity increases to $O(q^2n)$. The regression-based profiled attacks proposed by Schindler et al. [SLP05] provide an adaptable improvement method for the purpose to predict the coefficients individually in our work. These attacks can be seen as a variant of template attacks introduced by Chari et al. [CRR03], where the deterministic part of the leakage function is represented as a linear combination of basis functions, in order to reduce the number of profiling traces.

In this work, we target the stack-optimized Kyber implementation running on the ARM Cortex-M4. Utilizing the proposed cycle-level leakage model, we conduct several regression-based profiled attacks on the pair-pointwise multiplication during the decryption process. A known ciphertext attack and a chosen ciphertext attack are proposed to counteract the unprotected implementation. The chosen ciphertext attack is also applied to the masked implementation.

1.3 Main Contributions

The main contributions of this paper are as follows:

- We develop a novel cycle-level regression-based leakage model targeting multiply instructions and provide a systematic method for selecting explanatory variables in the model.

  The model characterizes the power consumption of multiply instructions within a single clock cycle, offering a detailed view on the power dissipation related to the execute stage of these instructions. Our research delves into the power characteristics of the MAC circuit used by the 16×16 multiply instruction on the Cortex-M4 device, shedding light on the type of multiplier circuit involved. Based on the identified multiplier type, we simulate the MAC circuit structure within the DSP extension and emulate intermediate values during computation, enabling more accurate power consumption predictions.

  In addition to the significance hypothesis test, we offer a systematic method for selecting explanatory variables. This approach is designed to optimize the model while avoiding meaningless or redundant variables at both the instruction and algorithm levels.

- We demonstrate the application of regression-based profiled attacks targeting the pair-pointwise multiplication, specifically the macro `doublebasemul_frombytes_asm` in the stack-optimized Kyber implementation. Our approaches allow for the individual prediction of secret coefficients, which is pivotal for mounting effective side-channel attacks.

To enhance the efficiency of the attacks, we additionally utilize the Hamming weight information leaked by the store instruction, where two result coefficients are stored in a single 32-bit word. The Hamming weight relations, including two equations between the two coefficients and the word, can be leveraged to filter candidates when predicting the secret coefficient. Thereby they reduce the complexity and the number of traces required to successfully mount the attacks.

- We conduct real-world attacks on the stack-optimized Kyber implementation using the STM32F303 device. Our attacks can successfully recover secret coefficients with a high success rate using a limited number of traces. Our experiments demonstrate that only 12 traces are required to achieve a 99.993% success rate when recovering a pair of secret coefficients in an unprotected implementation.
- We also experimentally demonstrate that our chosen ciphertext attack methodology is applicable to protected a Kyber implementation with masking. 201 traces are required to recover the secret coefficient pair with success rate 99.990% when targeting a first-order masked implementation. Thus, the work extends the scope of our research to include countermeasures against side-channel attacks.

2 Preliminaries

In this section, we aim to provide a comprehensive background to make our manuscript accessible to readers unfamiliar with key areas. These include the details of the lattice-based Kyber algorithm (Section 2.1), the methods underlying regression-based leakage modeling (Section 2.2.1), and the regression-based profiled attacks (Section 2.2.2). Further, we delve into the architectural specifics of the ARM Cortex-M4 processor, focusing on the details of instruction processing (Section 2.3). This analysis is essential for understanding the side-channel leakages that are the central concern of this study. Lastly, we provide an overview of the standard experimental configurations typically used in SCA (Section 2.4).

Notations In this manuscript, we employ lowercase letters to denote variables or polynomials, and boldface letters to signify column vectors or matrices. The notation $x[b]$ is utilized to represent the b^{th} bit of the variable x. Subscripts of the form (t) are used to index time or instructions, while superscripts such as (p) denote the index of traces for profiling, and (a) signifies the index of traces for attack. The variables with hat symbols indicate that they are in the NTT domain, such as $\hat{f}$. Throughout this paper, indexing is initialized at zero.

2.1 Kyber

As mentioned above, Kyber [SAB+22] is a post-quantum KEM scheme whose security relies on the hardness of the Module Learning with Errors (M-LWE) problem [BGV12, LS15], with the ring dimension $n = 256$ and the coefficient modulus $q = 3329$ over the a polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(x^n + 1)$. The module dimension $k = 2, 3$ or 4, and parameters of central binomial distributions $\mathcal{B}_{\eta_1}$,$\mathcal{B}_{\eta_2}$, $\eta_1 = 3, 2$ or 2, $\eta_2 = 2, 2$ or 2 corresponds to Kyber-512, -768 or -1024, respectively. As described in Algorithm 1, the key generation of Kyber uses the M-LWE equation $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$, where $\mathbf{s} \in \mathcal{R}_q^k$ is the secret key, $\hat{\mathbf{t}} \in \mathcal{R}_q^k$ and $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k}$ forms the public key and $\mathbf{e} \in \mathcal{R}_q^k$ is the noise vector. $\mathbf{s}$ and $\mathbf{e}$ are short polynomials, whose coefficients are sampled from $\mathcal{B}_{\eta_1}$.

The Kyber.CPAPKE.Encryption is shown in Algorithm 2. It generates the ciphertext c which consists of two compressed parts $\mathbf{c}_1$ and c_2. Algorithm 3 presents a simplified version of the decryption progress of Kyber.CPAPKE. The ciphertexts c is decompressed to two parts, $\mathbf{u} \in \mathcal{R}_q^k$ and $v \in \mathcal{R}_q$, which can be used to recover the message m. The

Algorithm 1: Kyber.CPAPKE.KeyGen (simplified)

Output: Public key: $(\hat{\mathbf{t}}, \rho)$, secret key: $\hat{\mathbf{s}}$

Choose uniform seeds ρ, σ

$\hat{\mathbf{A}} \leftarrow \text{Sample}_U(\rho) \in \mathcal{R}_q^{k \times k}$ `// Generate uniform` $\hat{\mathbf{A}}$ `in NTT domain`

$\mathbf{s}, \mathbf{e} \leftarrow \text{Sample}_{\mathcal{B}_{\eta_1}}(\sigma) \in \mathcal{R}_q^k$ `// Sample from binomial distribution`

$\hat{\mathbf{s}} := \text{NTT}(\mathbf{s})$, $\hat{\mathbf{e}} := \text{NTT}(\mathbf{e}) \in \mathcal{R}_q^k$ `// NTT for efficient multiplication`

$\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}} \in \mathcal{R}_q^k$

return $((\hat{\mathbf{t}}, \rho), \hat{\mathbf{s}})$

Algorithm 2: Kyber.CPAPKE.Enc (simplified)

Input: Public key: $(\hat{\mathbf{t}}, \rho)$, message: m, seed: τ

Output: Ciphertext $(\mathbf{c}_1, c_2)$

$\hat{\mathbf{A}} \leftarrow \text{Sample}_U(\rho) \in \mathcal{R}_q^{k \times k}$ `// Regenerate uniform` $\hat{\mathbf{A}}$ `in NTT domain`

$\mathbf{r} \leftarrow \text{Sample}_{\mathcal{B}_{\eta_1}}(\tau) \in \mathcal{R}_q^k$ `// Sample` $\mathbf{r}$ `from` $\mathcal{B}_{\eta_1}$

$\mathbf{e}_1 \leftarrow \text{Sample}_{\mathcal{B}_{\eta_2}}(\tau) \in \mathcal{R}_q^k$, $e_2 \leftarrow \text{Sample}_{\mathcal{B}_{\eta_2}}(\tau) \in \mathcal{R}_q$ `// Sample` $\mathbf{e}_1$, e_2 `from` $\mathcal{B}_{\eta_2}$

$\mathbf{u} := \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \text{NTT}(\mathbf{r})) + \mathbf{e}_1 \in \mathcal{R}_q^k$

$v := \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \text{NTT}(\mathbf{r})) + e_2 + \text{Decompress}_q(m, 1) \in \mathcal{R}_q$

$\mathbf{c}_1 := \text{Compress}_q(\mathbf{u}, d_u)$, $c_2 := \text{Compress}_q(v, d_v)$

return $(\mathbf{c}_1, c_2)$

Algorithm 3: Kyber.CPAPKE.Dec (simplified)

Input: Secret key: $\hat{\mathbf{s}}$, ciphertext: $(\mathbf{c}_1, c_2)$

Output: Message m

$\mathbf{u} := \text{Decompress}_q(\mathbf{c}_1, d_u) \in \mathcal{R}_q^k$

$v := \text{Decompress}_q(c_2, d_v) \in \mathcal{R}_q$

$m := \text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1)$

return m

function Compress_q maps from $\mathbb{Z}_q$ to $\{0,1\}^d$, while another function Decompress_q maps from $\{0,1\}^d$ to $\mathbb{Z}_q$. Both functions are operated on the coefficients of the polynomial(s). When applying them to polynomials, they are applied coefficient-wise.

$$\text{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d,$$
$$\text{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor,$$

where x is the element of $\mathbb{Z}_q$, d is the compression size.

2.1.1 Incomplete NTT and Pair-pointwise Multiplication

Unlike a full NTT, Kyber skips the last layer. This modification stems from the use of only n-th primitive roots of unity, where the modulus polynomial $X^n + 1$ factors into polynomials of degree 2. Therefore, in NTT domain, multiplication is not purely pointwise, but multiplications of polynomials of degree one (*pair-pointwise*). Hence,

$$ct = (ct_0, ct_1, ct_2, ct_3, ..., ct_{254}, ct_{255}),$$
$$\hat{ct} = \text{NTT}(ct) = (\hat{ct}_0 + \hat{ct}_1 X, \hat{ct}_2 + \hat{ct}_3 X, ..., \hat{ct}_{254} + \hat{ct}_{255} X),$$

with

$$\hat{ct}_{2i} = \sum_{j=0}^{127} ct_{2j}\zeta^{(2i+1)j},\ \hat{ct}_{2i+1} = \sum_{j=0}^{127} ct_{2j+1}\zeta^{(2i+1)j},$$

where i ranges from 0 to $n/2-1$.

Using NTT and its inverse NTT^{-1} we can compute the product $ct \cdot sk$ of two elements $ct, sk \in \mathcal{R}_q$ very efficiently as $\mathrm{NTT}^{-1}(\mathrm{NTT}(ct) \circ \mathrm{NTT}(sk))$ where $\mathrm{NTT}(ct) \circ \mathrm{NTT}(sk) = \hat{ct} \circ \hat{sk} = \hat{r}$ denotes the basecase multiplication consisting of the 128 products

$$\hat{r}_{2i} + \hat{r}_{2i+1}X = (\hat{ct}_{2i} + \hat{ct}_{2i+1}X)(\hat{sk}_{2i} + \hat{sk}_{2i+1}X) \bmod (X^2 - \zeta^{2i+1})$$

of linear polynomials, as presented in Algorithm 4.

Algorithm 4: Pair-pointwise Multiplication in Kyber.CPAPKE.Dec (simplified)

Input: Secret polynomial $\hat{sk} \in \mathcal{R}_q$, ciphertext polynomial $\hat{ct} \in \mathcal{R}_q$
Input: Twiddle factors $(\zeta^1, \zeta^3, \cdots, \zeta^{n-1})$
Output: Result polynomial $\hat{r}$
1 **for** $i \leftarrow 0$ to $n/2-1$ **then**
2 $\quad \hat{r}_{2i} = \hat{ct}_{2i} \cdot \hat{sk}_{2i} + \hat{ct}_{2i+1} \cdot \hat{sk}_{2i+1} \cdot \zeta^{2i+1}$
3 $\quad \hat{r}_{2i+1} = \hat{ct}_{2i} \cdot \hat{sk}_{2i+1} + \hat{ct}_{2i+1} \cdot \hat{sk}_{2i}$
4 **return** $\hat{r}$

2.1.2 Masked implementations

Masking [CJRR99] is a well-established technique used to protect cryptographic algorithms against SCA attacks such as DPA and CPA. It splits sensitive data into multiple shares, and then processes these shares through carefully devised functions. Following the masked computation, the shares are recombined to yield the actual output, thereby preserving the confidentiality of the cryptographic operation.

Since the polynomial multiplication and addition are linear transformations of secret coefficients, the masking technique is a natural fit for lattice-based cryptography. Several studies have been conducted on the masking of lattice-based schemes [RRVV15, RRd+16, OSPG18, BDK+21]. Furthermore, Bos et al. [BGR+21] and Heinz et al. [HKL+22] have introduced concrete masking techniques specifically tailored for the Kyber algorithm. Figure 1 provides a simplified depiction of a first-order masked polynomial multiplication in Kyber decryption processing. The secret key $\hat{\mathbf{s}}$ is partitioned into two shares, $\hat{\mathbf{s}}_{s0}$ and $\hat{\mathbf{s}}_{s1}$, such that $\hat{\mathbf{s}} = \hat{\mathbf{s}}_{s0} + \hat{\mathbf{s}}_{s1} \bmod q$. Subsequently, pair-pointwise multiplications, polynomial additions, and NTT^{-1} are computed on each share individually.

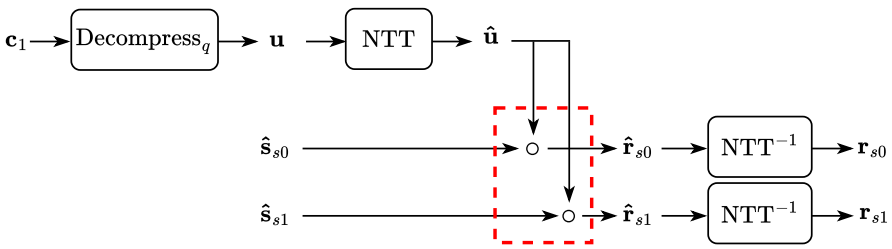


Figure 1: A simplified depiction of a first-order masked Kyber decryption process. Parts that are unnecessary for our analysis are omitted. The side channel leakage, which originates from the pair-pointwise multiplications, is highlighted by a red dashed box.

2.2 Regression-based leakage Profiling

2.2.1 Regression-based leakage Modeling

McCann et al. [MOW17] opt for a 'grey box' approach that does not require detailed hardware descriptions but assumes access to assembly code to construct models at the instruction level. They concentrate on predicting variables that can be derived from assembly sequences (i.e. input data, register locations). Then, they fit models to the predicted leakage y' of instructions via ordinary least squares estimation.

$$y' = \mathcal{Y}'(o_0, o_1, \tilde{o}_0, \tilde{o}_1) = \left(\mathbf{O}_0^T, \mathbf{O}_1^T, \mathbf{T}_0^T, \mathbf{T}_1^T\right) \cdot \boldsymbol{\beta} + \delta. \tag{1}$$

In this equation, o_i represents the i^{th} 32-bit operand of the current instruction, while $\tilde{o}_i$ is the corresponding operand from the previous instruction. $\mathbf{O}_i = (o_i[31], o_i[30], \cdots, o_i[0])^T$ denotes the vector of bits from the i^{th} operand o_i, and $\mathbf{T}_i = (o_i[31] \oplus \tilde{o}_i[31], o_i[30] \oplus \tilde{o}_i[30], \cdots, o_i[0] \oplus \tilde{o}_i[0])^T$ represents the vector of bit transitions in the i^{th} pipeline register. The scalar intercept δ and the vector of coefficients $\boldsymbol{\beta} = (\beta_{127}, \beta_{126}, \cdots, \beta_0)^T$ are the model's parameters to be estimated.

To ascertain the most influential variables in the model, McCann et al. employ the F-test, a statistical method designed to evaluate the significance of groups of variables in regression-based models. This test is particularly effective in scenarios where the potential explanatory power of numerous variables needs to be rigorously assessed.

They initially construct a full model A incorporating all potential predictors. Then, they construct a reduced model B excluding a subset of variables, such that $p_B < p_A$. The formula for the F-test is based on the ratio of the reduction in residual sum of squares (RSS) between the full and reduced models to the increase in the residual degrees of freedom, normalized by the residual mean square error of the full model.

The F-test is calculated using the following formula:

$$F = \frac{\left(\frac{RSS_B - RSS_A}{p_A - p_B}\right)}{\left(\frac{RSS_A}{n - p_A - 1}\right)}, \tag{2}$$

where p_A, p_B are number of parameters of the two models, respectively, and n is the total number of observations. In essence, the numerator of the F-test represents the per-parameter increase in the residual sum of squares due to excluding $p_A - p_B$ parameters, while the denominator represents the average residual sum of squares per degree of freedom in the full model.

To determine the statistical significance, the calculated F-statistic is compared against a critical value from the F-distribution with $(p_A - p_B, n - p_A - 1)$ degrees of freedom. If the F-statistic is larger than the critical value at a chosen significance level (such as 5%), the *null hypothesis* that the reduced model provides an adequately similar fit to the data as the full model is rejected. This implies that the variables excluded in the reduced model are collectively significant and should be included in the model for a more accurate representation.

2.2.2 Regression-based Profiled Attack

Regression-based profiled attacks have been introduced by Schindler et al. [SLP05]. They can be viewed as a variant of Chari et al.'s template attacks [CRR03], where the deterministic part of the leakage function is expressed as a linear combination of intermediate value bits, in order to reduce the number of profiling traces. The adversary measures physical observation $y_{(t)}$ at time t with the intermediate value $x = \sum_{i=0}^{b-1} x[i] \cdot 2^i \in$

$\mathbb{Z}_{2^b}$, where $t = 0, 1, 2, ..., n_t - 1$. The predicted power consumption $y'_{(t)}(x)$ is modeled as follow:

$$y'_{(t)} = \mathcal{Y}'_{(t)}(x) = \mathbf{X}^T \cdot \boldsymbol{\beta}_{(t)} + \delta_{(t)}, \tag{3}$$

where $\mathbf{X} = (x[b-1], x[b-2], \cdots, x[0])^T$. The vector $\boldsymbol{\beta}_{(t)} = \left(\beta_{b-1,(t)}, \beta_{b-2,(t)}, \cdots, \beta_{0,(t)}\right)^T$ are the weights of the models learned by profiling. $\delta_{(t)}$ is the scalar intercept to be estimated.

The profiling step collects a set of n_p traces $\mathbf{Y}^{n_p} = \left(\mathbf{y}^{(0)}, \mathbf{y}^{(1)}, \cdots, \mathbf{y}^{(n_p-1)}\right)^T$ and their corresponding intermediate values $\mathbf{x} = \left(x^{(0)}, x^{(1)}, \cdots, x^{(n_p-1)}\right)^T \in \mathbb{Z}_{2^b}^{n_p}$, where $\mathbf{y}^{(p)} = \left(y^{(p)}_{(0)}, y^{(p)}_{(1)}, \cdots, y^{(p)}_{(n_t-1)}\right)^T$ and $x^{(p)} = \sum_{i=0}^{b-1} x^{(p)}[i] \cdot 2^i$ for $p = 0, 1, ..., n_p - 1$. Let $\mathbf{y}_{(t)} = \left(y^{(0)}_{(t)}, y^{(1)}_{(t)}, \cdots, y^{(n_p-1)}_{(t)}\right)^T$ be the t^{th} column of $\mathbf{Y}^{n_p}$, and $\mathcal{Y}'_{(t)}(\mathbf{x}) = \Big(\mathcal{Y}'_{(t)}(x^{(0)}), \mathcal{Y}'_{(t)}(x^{(1)}), \cdots, \mathcal{Y}'_{(t)}(x^{(n_p-1)})\Big)^T$ be the predicted power consumption vector at time t. Thus, the residual terms $\boldsymbol{\varepsilon}_{(t)}$ is showed as follows:

$$\boldsymbol{\varepsilon}_{(t)} = \mathbf{y}_{(t)} - \mathcal{Y}'_{(t)}(\mathbf{x}).$$

The minimization of $\|\boldsymbol{\varepsilon}_{(t)}\|$ is an ordinary least squares problem.

Once $\boldsymbol{\beta}_{(t)}$ and $\delta_{(t)}$ are computed for each time $t = 0, 1, ..., n_t - 1$, $\mathbf{y}' = \Big(\mathcal{Y}'_{(0)}(x), \mathcal{Y}'_{(1)}(x), \cdots, \mathcal{Y}'_{(n_t-1)}(x)\Big)^T$ is used as the mean for a pooled template [CK13, CK18], and the approximate covariance matrix $\boldsymbol{\Sigma}$ for all traces can be computed as the empirical covariance of the residual terms,

$$\boldsymbol{\Sigma} = \frac{1}{n_p - 1} \left(\boldsymbol{\varepsilon}_{(0)}, \boldsymbol{\varepsilon}_{(1)}, \cdots, \boldsymbol{\varepsilon}_{(n_t-1)}\right)^T \cdot \left(\boldsymbol{\varepsilon}_{(0)}, \boldsymbol{\varepsilon}_{(1)}, \cdots, \boldsymbol{\varepsilon}_{(n_t-1)}\right).$$

The intermediate value x is related to the secret key k that can be expressed as $x = \mathcal{G}(m, k)$, where m is the value that attackers can know or choose, $\mathcal{G}$ is a function that acts on m and k. The modeled probability density Pd for a new leakage trace $\mathbf{y}^* = \left(y^*_{(0)}, y^*_{(1)}, \cdots, y^*_{(n_t-1)}\right)^T$ with a known $m = m^*$ is

$$\begin{aligned} \mathrm{Pd}(\mathbf{y}^* | k, m = m^*) &= \mathrm{Pd}(\mathbf{y}^* | x = \mathcal{G}(m, k)) \\ &= \frac{1}{\sqrt{(2\pi)^{n_t} |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2} \left(\mathbf{y}^* - \mathbf{y}'\right)^T \boldsymbol{\Sigma}^{-1} \left(\mathbf{y}^* - \mathbf{y}'\right)\right), \end{aligned}$$

The conditional distribution $\mathrm{Pd}(k|\mathbf{y}^*)$ can in turn be derived from the probability density function $\mathrm{Pd}(\mathbf{y}^*|k)$ using Bayes' rule,

$$\mathrm{Pd}(k|\mathbf{y}^*) = \frac{\mathrm{Pd}(\mathbf{y}^*|k) \cdot \mathrm{Pr(k)}}{\sum_{k' \in \mathcal{K}} \mathrm{Pd}(\mathbf{y}^*|k') \cdot \mathrm{Pr(k')}},$$

where $\mathcal{K}$ is the key space, the denominator is the same for each k. Assuming a uniform prior probability $\mathrm{Pr}(k) = |\mathcal{K}|^{-1}$, the likelihood $\ell(k|\mathbf{y}^*)$ [CK13, CK18] can be computed as

$$\ell(k|\mathbf{y}^*) = C \cdot \mathrm{Pd}(k|\mathbf{y}^*) = \mathrm{Pd}(\mathbf{y}^*|k),$$

where $C = \frac{\sum_{k' \in \mathcal{K}} \mathrm{Pd}(\mathbf{y}^*|k') \cdot \mathrm{Pr(k')}}{\mathrm{Pr(k)}}$ is a constant. Finally, the likelihood values are computed for each candidate k given a trace $\mathbf{y}^*$. The attackers sort them, get ranks of the candidates and identify the most likely candidate.

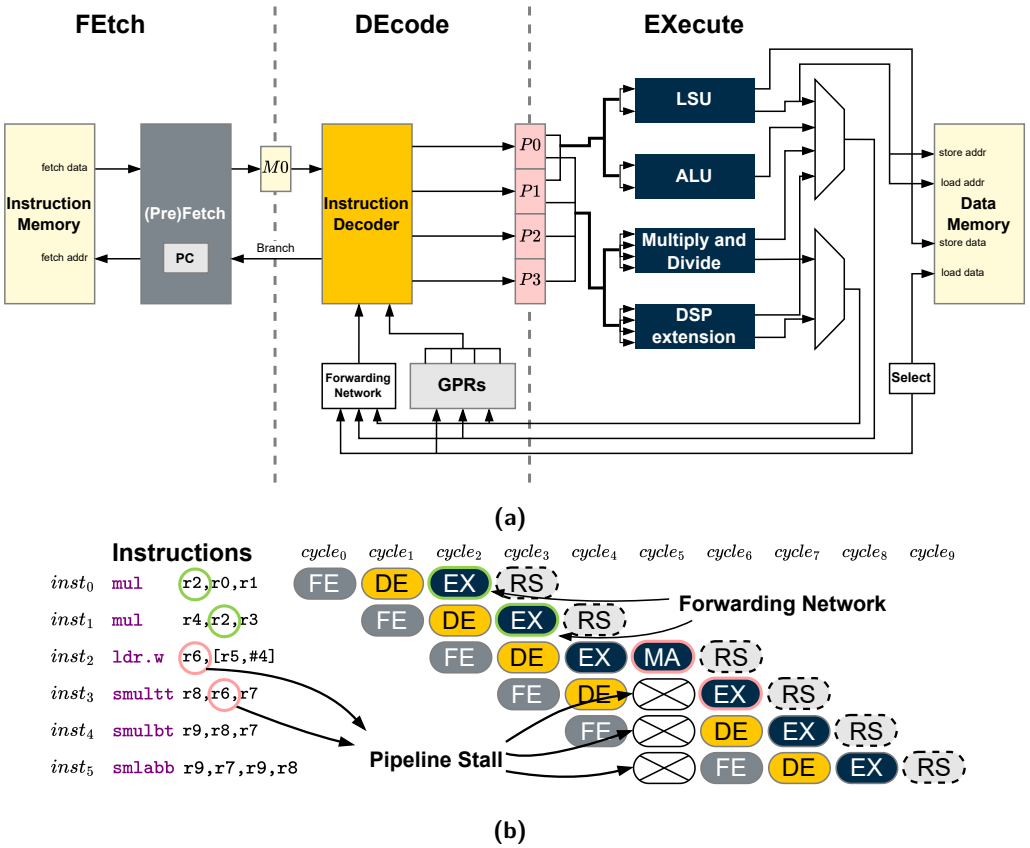


Figure 2: Architecture of the ARM Cortex-M4 microcontroller. (a) The three-stage pipeline architecture of the ARM Cortex-M4. The dashed lines serve to distinguish between the distinct stages of the pipeline. Registers positioned along these dashed lines are pipeline registers, which conventionally have a width of 32 bits. (b) The processing of instructions within the pipeline. The forwarding path mitigate the occurrence of pipeline stalls due to RAW conflicts, thereby ensuring that data dependencies are efficiently managed.

When combining the multiple individual leakage traces $\mathbf{y}^{(a)}$ from $\mathbf{Y}^{n_a}$, $a = 0, 1, ..., n_a - 1$, the joint likelihood can be expressed as

$$\ell(k|\mathbf{Y}^{n_a}) = \prod_{a=0}^{n_a-1} \ell(k|\mathbf{y}^{(a)}).$$

By applying the logarithm to both sides we have the joint log-likelihood

$$\log \ell(k|\mathbf{Y}^{n_a}) = \sum_{a=0}^{n_a-1} \log \ell(k|\mathbf{y}^{(a)}). \tag{4}$$

2.3 ARM Cortex-M4

2.3.1 Architecture

The Cortex-M4 is a 32-bit RISC processor from ARM, noted for its use in the NIST PQC project as a recommended evaluation platform, as shown in Figure 2a. It operates on the Harvard architecture, utilizing the Armv7E-M instruction set and supporting a 32-bit data path [ARMa, ARMc]. The Cortex-M4 core consists of the following components :

General-Purpose Registers (GPRs) The Cortex-M4 features a core register bank comprising 16 32-bit registers ($r0$-$r15$), with the first 13 ($r0$-$r12$) serving general-purpose functions such as holding intermediate variables and results. The remaining three are reserved for specific functions: the Stack Pointer (SP, $r13$), Link Register (LR, $r14$), and Program Counter (PC, $r15$).

Functional Units (FUs) The processor includes multiple FUs like the Arithmetic Logic Unit (ALU) for basic arithmetic and logic operations, and a dedicated Multiply and Divide unit capable of handling complex arithmetic operations efficiently. The Cortex-M4 core has a DSP extension [ARMd], enhancing its capabilities with DSP-specific instructions like saturating, multiply and accumulate, and Single Instruction Multiple Data (SIMD) operations. These instructions are seamlessly integrated into the existing Instruction Set Architecture (ISA), utilizing the same 32-bit GPR bank, and execute in a single cycle. A Load/Store Unit (LSU) manages data movements between the core and memory, using base addresses and offsets to calculate memory addresses. It loads/stores data from/into the data memory via the data bus between the processor core and RAM.

Pipeline Architecture The processor adopts a three-stage pipeline design — **FE**tch, **DE**code, and **EX**ecute. The **FE** stage retrieves instructions from memory, **DE** decodes them and passes operands to the **EX** stage where the FUs process the operations. Results are then written back to the GPRs.

Pipeline Registers Between each stage, pipeline registers store operational data to ensure smooth data flow through the pipeline. Specific registers like $M0$ (instruction register between **FE** and **DE**) and registers $P0$ to $P3$ (operand registers between **DE** and **EX**) facilitate the transfer of data and control signals across stages.

2.3.2 The Journey of Instructions

In this subsection, we demonstrate how instructions are processed in the pipeline as shown in Figure 2b. Let $inst_i$ and $cycle_i$ denote the i^{th} instruction and the i^{th} clock cycle, respectively.

Consider the processing flow of an example instruction $inst_0$, as depicted in Figure 2b. The `mul` instruction is fetched from the instruction memory during $cycle_0$. The opcode for `mul` is then stored in the $M0$ register at the rising edge of the clock in $cycle_1$, as shown in Figure 2a. The decoder reads the opcode from $M0$, generates the necessary control signals, and retrieves operands from the GPRs. At the rising edge of the clock in $cycle_2$, these operands are subsequently stored in pipeline registers $P0$ and $P1$ which hold the two 32-bit operands of `mul`. Unlike operations such as division, which may require multiple cycles, the Cortex-M4's arithmetic and logic instructions, including `mul`, typically complete within a single cycle. Therefore, in $cycle_2$, the Multiply and Divide unit processes the operands and computes the result. Although the Cortex-M4 architecture does not include a dedicated Write Back stage, the result of $inst_0$ is effectively written to the destination register at the rising edge of the clock in $cycle_3$. This action, while not an official stage of the pipeline, is illustrated using a dotted box in Figure 2b and referred to as the Result Store (**RS**) stage. The three-stage pipeline architecture enables the processor to manage 3 instructions concurrently within the duration of a single clock cycle (or 4 instructions if the **RS** stage is included). It is essential to consider the impact of surrounding instructions when modeling a specific instruction.

If the result of the preceding instruction is the operand of the current instruction, a Read-after-Write (RAW) conflict may occur, leading to an incorrect operand read or a pipeline stall. $inst_1$ gets operands from GPRs and puts them into pipeline registers in $cycle_3$, while $inst_0$ stores the result into the GPR, leading to an unprepared operand. In the ARM Cortex-M4, the forwarding network enables outputs from FUs and memory to

bypass storage in GPRs, allowing direct transfer, as shown in Figure 2a. Thus, the incorrect operand reading and pipeline stalls are mitigated, ensuring that the data dependencies are efficiently managed. However, pipeline stalls may still occur, for example, as a result of memory access instructions [ARMb, Chapter 3.3.3]. A concrete example is depicted in Figure 2b. In $cycle_5$, $inst_2$ `ldr.w` loads the data from memory[1], which serves as the operand for $inst_3$ `smultt`. A stall happens when the operand of $inst_3$ should be prepared already but the data has not yet been loaded. In $cycle_6$, the loaded data is stored into GPRs and sent to the pipeline register using the forwarding network simultaneously. We need to be aware of these possible pipeline stalls when locating instructions in the power trace.

2.4 Experimental Setups

Our experimental setup utilizes the ChipWhisperer CW308 platform, hosting an STM32F303 target with a Cortex-M4 core. The microcontroller is configured to operate at a clock frequency of 5 MHz [2]. Signal acquisition is conducted using a PicoScope 5444B digital oscilloscope, sampling the signal at 250 MSamples/s with 10-bit resolution, yielding 50 samples per clock. A CW501 differential probe is used in conjunction with a CW502 low-noise amplifier, boosting the signal by 20 dB to ensure high-quality measurements. The analysis runs on a computer equipped with 8GB RAM and AMD Ryzen 7 4800HS Processor clocked at 2.90 GHz.

We use the stack-optimized Kyber-768 implementation of *pqm4* [impb], with the only addition being a trigger to simplify the recording of traces. The focus of our experiments is on the function **frombytes_mul_asm** (more concretely, the macro **doublebasemul_frombytes_asm**), which is specifically chosen because it handles the polynomial multiplication of the secret and the ciphertext, the critical operations in our analysis. In terms of power trace compression, unlike McCann et al. [MOW17], who employ the method of maximum extraction by selecting the point of interest that is the maximum peak in the clock cycles during which instruction leaks, we adopt the raw integration approach [MOP07, Chapter 4.5]. This approach calculates the sum of all sampling points within each cycle. These sums are then used as the compressed power traces. Typically, this approach tends to be more robust than maximum extraction.

In order to reflect a real-world scenario, we use two physical instances of the ChipWhisperer device, separately for profiling and attacking.

3 A Novel Cycle-level Power Leakage Model

In this section, we quantify the single-cycle power consumption of the Cortex M4 core based on its pipeline architecture and analyze the sources of data-related power leakage. Then we present a fundamental cycle-level leakage model using linear regression (Section 3.1). In our attack scenario, we focus on multiply instructions that directly leak secret information, executed on the MAC circuit within the DSP extension. Therefore, we further analyze the power consumption characteristics of the MAC circuit by examining the structure of the multipliers (Section 3.2). Building on this insight, we simulate the structural components of the multiplier and emulate the intermediate values generated during the multiplication process (Section 3.3). Finally, we present a comprehensive leakage model and propose a systematic method for selecting explanatory variables targeting deterministic instruction sequences (Section 3.4).

[1] Memory Access is not a dedicated stage in Cortex-M4 pipeline architecture. It is part of the **EX** stage, but cost additional clock cycle(s). In $cycle_5$ of Figure 2b, we use **MA** to present this operation and paint

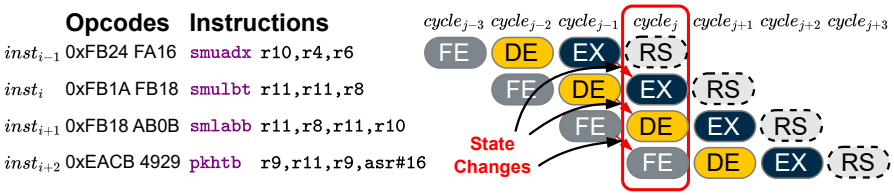


Figure 3: State changes that occur within an instruction sequence. We observe the state changes that occur within a clock cycle, which are highlighted with a red solid box.

3.1 Cycle-level Power Consumption

The dynamic power dissipation in the ARM Cortex-M4 microcontrollers is predominantly attributed to instruction execution, memory access, and data processing, and is influenced by factors such as supply voltage, clock frequency, and instruction types. The microcontroller features a three-stage pipeline architecture, which is designed to balance efficiency and power consumption. It consists of the **FE**, **DE**, and **EX** stages, along with an additional RS stage, defined in section 2.3.2, which accounts for power during result storage.

Assuming no stalls or interrupts, and ignoring branch prediction effects, the power consumption in a clock cycle is calculated by summing the power changes across the pipeline stages, inclusive of the power of storing result. The total power E_{total} is given by:

$$E_{\text{total}} = E_{\text{FE}} + E_{\text{DE}} + E_{\text{EX}} + E_{\text{RS}} + E_{\text{other}},$$

where $E_{\text{FE}}, E_{\text{DE}}, E_{\text{EX}}$ and E_{RS} represent the power consumption of each stage, respectively. E_{other} covers any additional variations due to environmental or supply conditions.

In side-channel analysis, the **EX** stage is pivotal as it processes confidential data, where power variations potentially revealing information, so as the RS stage. Cycle-level power consumption allows to simplify the analysis of operations observed in a determined sequence. As shown in Figure 3, at the onset of $cycle_j$, register $r10$ is updated, and $inst_{i-1}$ stores its result. The power consumption at this moment is influenced by the Hamming distance between the new and previous values in $r10$. During the same cycle, $inst_i$ executes a multiplication, with power consumption varying based on the operands involved. Since no system interrupts or branching occurs in this sequence, the power consumption for $inst_{i+1}$ in the **DE** stage and $inst_{i+2}$ in the **FE** stage can be assumed *constant* due to their predetermined post-compilation.

Given these observations, the total power consumption for $cycle_j$ can be simplified to:

$$E_{\text{total}} \approx E_{\text{EX}} + E_{\text{RS}} + E'_{\text{other}}, \tag{5}$$

where E'_{other} accounts for the relatively constant power consumption of the **FE** and **DE** stages combined with other minor variations.

The **EX** stage's power consumption is influenced by operand-related variations, including:

Pipeline Register Activities: Power consumption affected by data replacement in pipeline registers. In this work, the multiply instructions in the target function only use 2 or 3 operands, so we only focus on the first three pipeline registers.

---

it the same color as **EX**.

[2]The maximum sampling frequency of our oscilloscope is only 250 MSamples/s. In order to obtain as many sampling points as possible within a single clock cycle, we use a lower operating frequency.

Results Storage of the Previous Instruction: Power consumption during writing results back to registers is influenced by bit flips. The target multiply instructions only produce 32-bit results, indicating that the destination register is one of the GPRs.

Execution of Instructions: Power consumption patterns vary with the type and operands of instructions, affecting the FUs' logic. Our work focuses on 16-bit multiply instructions, executed by the multiplier in the Cortex-M4 DSP extension. Therefore, we discuss the FU of the Cortex-M4 DSP, specifically, the multiplier in DSP extension.

A foundational cycle-level power leakage model for the **EX** and the **RS** stages is developed, and a linear regression approach is used to quantify power consumption associated with each instruction. The model is formulated as

$$y'_{(t)} = \mathcal{Y}'_{(t)}(o_{0,(t)}, o_{1,(t)}, o_{2,(t)}, r_{(t)}, \tilde{o}_{0,(t)}, \tilde{o}_{1,(t)}, \tilde{o}_{2,(t)}, \tilde{r}_{(t)}) = \mathbf{X}^T_{(t)} \cdot \boldsymbol{\beta}_{(t)} + \delta_{(t)}, \tag{6}$$

where the subscript (t) represents the index of the instruction. The vector of weights $\boldsymbol{\beta}_{(t)}$ and the scalar intercept $\delta_{(t)}$ are the model's parameters to be estimated. o_i is the i^{th} 32-bit operand of the current instruction, $\tilde{o}_i$ is the i^{th} 32-bit operand corresponding to the previous instruction, $i = 0, 1$ or 2. $r_{(t)}$ denotes the result of the previous instruction, $\tilde{r}_{(t)}$ denotes the value from the earlier instruction in the GPR which is the destination register of the previous instruction.

$$\mathbf{X}^T_{(t)} = \Big(\mathbf{O}^T_{0,(t)}, \mathbf{O}^T_{1,(t)}, \mathbf{O}^T_{2,(t)}, \mathbf{R}^T_{(t)}, \mathbf{DM}^T_{(t)}, \mathbf{TO}^T_{0,(t)}, \mathbf{TO}^T_{2,(t)}, \mathbf{TO}^T_{2,(t)}, \mathbf{TR}^T_{(t)}, \mathbf{TDM}^T_{(t)}\Big), \tag{7}$$

where $\mathbf{O}_i = (o_i[31], o_i[30], \cdots, o_i[0])^T$ denotes the vector of bits from the i^{th} operand o_i, and $\mathbf{T}_i = (o_i[31] \oplus \tilde{o}_i[31], o_i[30] \oplus \tilde{o}_i[30], \cdots, o_i[0] \oplus \tilde{o}_i[0])^T$ represents the vector of bit transitions in the i^{th} pipeline register. $\mathbf{R_{(t)}} = \big(r_{(t)}[31], r_{(t)}[30], \cdots, r_{(t)}[0]\big)^T$ is the vector of bits from $r_{(t)}$, $\mathbf{TR_{(t)}} = \big(r_{(t)}[31] \oplus \tilde{r}_{(t)}[31], r_{(t)}[30] \oplus \tilde{r}_{(t)}[30], \cdots, r_{(t)}[0] \oplus \tilde{r}_{(t)}[0]\big)^T$ is the vector of bit transitions in the GPR. $\mathbf{DM}_{(t)}$ and $\mathbf{TDM}_{(t)}$ represent the state of the multiplier generated by the t^{th} instruction and the state transition between the previous instruction and the current instruction, respectively. The specific composition of $\mathbf{DM}_{(t)}$ and $\mathbf{TDM}_{(t)}$ is explored in subsequent sections.

3.2 Simple Reverse Engineering of MAC Circuit in DSP Extension

We embark on an empirical study to ascertain the type of MAC circuit implemented within the DSP extension of the Cortex-M4. This investigation is grounded in the analysis of cycle-level power consumption patterns, specifically focusing on the distinct roles played by the multiplicands and multipliers in the multiplication process. The primary objective is to discern whether the MAC circuit aligns with a (Radix-2) Booth multiplier, a (Radix-4) Modified Booth multiplier, or a Baugh-Wooley array multiplier architecture.

To discern the type of MAC circuit, it is essential to understand the differences of power consumption among these three multipliers. In the Booth encoder, either the multiplicand or multiplier is encoded, leading to a difference in how these inputs contribute to the overall power consumption of the multiplication operation. The Booth multiplier groups two adjacent bits of the operand at a time (with each group overlapping by one bit from the previous), generating n encoded values and n partial products for an n-bit multiplication. On the other hand, the Modified Booth multiplier encodes three bits at a time, (with each group still sharing one overlapping bit with the adjacent group), thereby halving the number of required encoding steps and partial products for an n-bit multiplication. The Baugh-Wooley array multiplier is characterized by its symmetric handling of multiplicands and multipliers. Unlike the Booth multipliers, the symmetry leads to a more uniform contribution of both inputs to the overall power consumption.

To distinguish between the three types of multipliers, we design the following experiments with two input sets: one with non-zero multiplicands and zero multipliers, and the other with zero multiplicands and non-zero multipliers. The Hamming weight of the non-zero inputs is 1. Then we collect the power traces from these inputs and analyze the cycle-level power consumption of the multiply instructions.

1. We distinguish between Booth and Baugh-Wooley multipliers by comparing the power traces from the input sets. Minimal power differences suggest that the multiplicand and multiplier have similar effects on power consumption, pointing to a Baugh-Wooley multiplier. Conversely, significant differences indicate distinct contributions, identifying the Booth multiplier. As illustrated in Figure 4, the variation in cycle-level power traces between different multiplicands and multipliers indicates that their roles in power consumption are not identical, suggesting that the multiplier is unlikely a Baugh-Wooley array multiplier.

2. We further differentiate between Radix-2 and Radix-4 multipliers by analyzing their distinct encoding methods. As shown in Figure 4, the first set exhibits higher overall energy compared to the second, suggesting that the multiplicand is involved in the encoding process. Notably, negative encoding tends to increase power consumption, as it results in partial products derived from the 2's complement of the multiplier multiple. This 2's complement is created through bit flipping, which leads to higher power usage. In a Radix-2 Booth multiplier, each bit of the multiplicand has the potential to induce negative encoding, whereas in a Radix-4 Booth multiplier, this effect is restricted to the odd-indexed bits of the multiplicand. The significant differences observed between odd- and even-indexed bits in the multiplicand, as depicted in Figure 4a, b, c, suggest that the MAC unit in the DSP extension may be using a modified Booth encoder [Boo51] for partial product generation.

Figure 17 illustrates the code snippet we meticulously designed. The target multiply (-accumulate) instruction is surrounded by a series of `nop.w` instructions to isolate the power leakage specifically attributable to the target instruction, thereby enabling a more precise and uncontaminated measurement.

In summary, our experiments suggest that the MAC unit within the DSP extension employs a modified Booth encoding strategy, as evidenced by the distinct power consumption patterns corresponding to various bit positions in the multiplicand. This insight sheds light on the underlying architecture of the MAC unit, supporting our subsequent simulation on the power consumption of the multiplication.

3.3 Simulation of Intermediate Variables

In the previous subsection, through a process akin to reverse engineering, we establish the type of multiplier used in the DSP extension as a modified Booth multiplier. Due to the absence of public documentation on the underlying circuit logic of the Cortex-M4 DSP extension, we adopt an approach that employs a classic design of the modified Booth multiplier to simulate the intermediate variables generated during the execution of a single-cycle 16-bit multiplication (and accumulation) in the Cortex-M4. This simulation aimed to approximate the power consumption of the MAC unit of the DSP extension.

A modified Booth multiplier generally consists of a modified Booth [Boo51] encoder, partial product generator (PPG), a carry-save adder (CSA), partial product reduction tree (PPRT), and a carry-propagate adder. The block diagram of the MAC unit is shown in Figure 5a.

The multiplicand md and multiplier mr serve as the inputs to the multiplication process. They are represented as sums of their individual bits: $md = \sum_0^{15} md[i] \cdot 2^i$, $mr = \sum_0^{15} mr[i] \cdot 2^i$, where, $md[i]$ and $mr[i]$ denote the i^{th} bit of the multiplicand

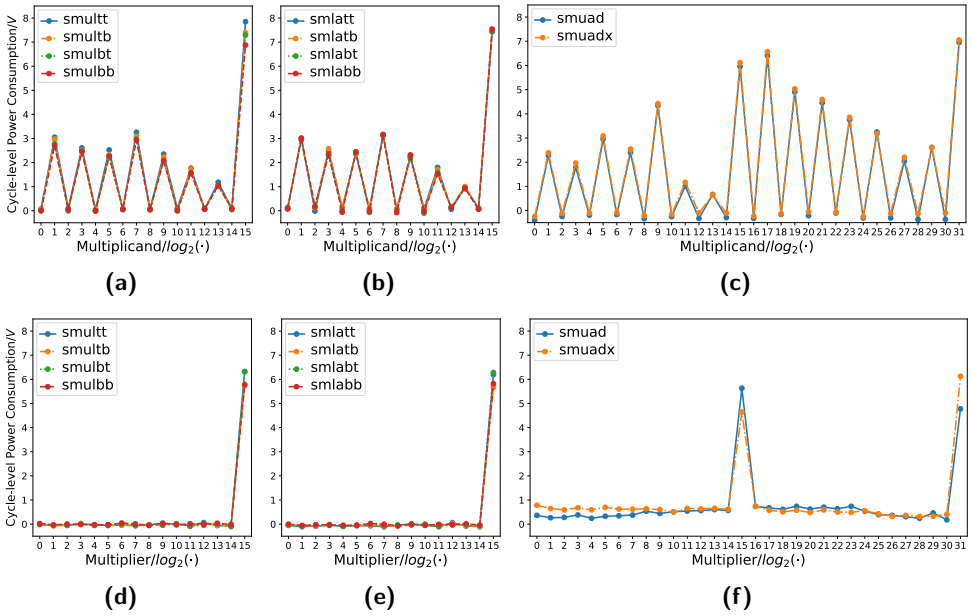


Figure 4: Experiments to discern the type of the MAC circuit in the DSP extension of the Cortex-M4. (a) and (d) use the signed 16-bit multiply instructions. (b) and (e) use the signed 16-bit multiply-accumulate instructions. (c) and (f) use the signed dual 16-bit multiply instructions.

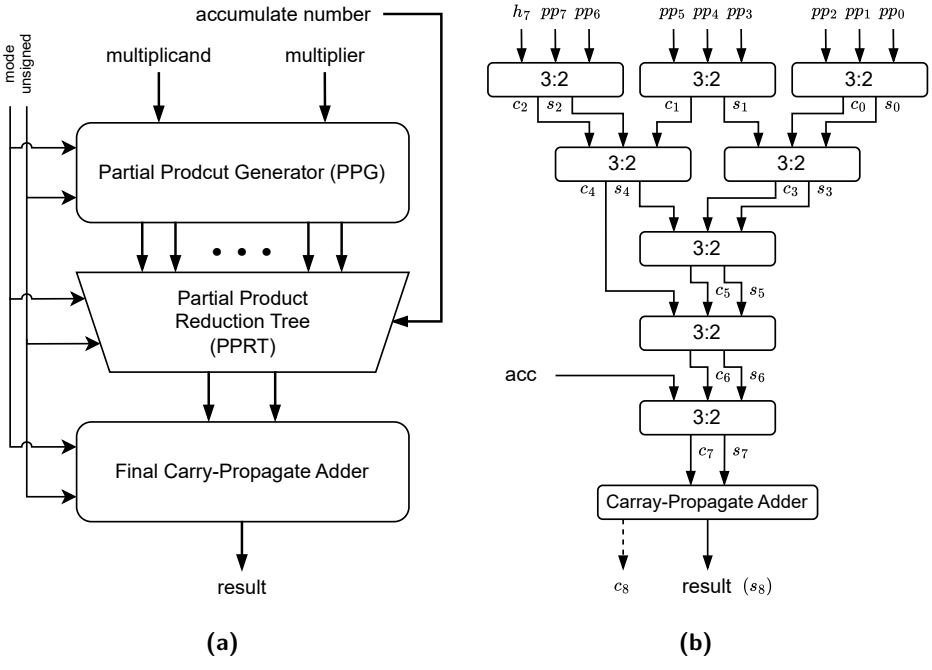


Figure 5: A typical modified Booth multiplier design. (a) MAC Architecture block diagram. (b) Partial product reduction tree and carry-propagate adder for 16-bit MAC.

and multiplier, respectively. The multiplier bits are encoded using the modified Booth algorithm. Let $mr[-1]$ be 0, the Booth encoding for each group of bits is calculated as

Table 2: Width of variables of 16×16 modified Booth multiplier. pp_i and h_i are reasonably abbreviated to indicate that all partial products and hot-ones have the same width, where $i = 0, ..., 7$.

Variable	pp_i	h_i	s_0 c_0	s_1 c_1	s_2 c_2	s_3 c_3	s_4 c_4	s_5 c_5	s_6 c_6	s_7 c_7	s_8 c_8
Width	18	1	20	20	18	24	20	26	22	32	32

$Code_i = -2 \times mr[2i+1] + mr[2i] + mr[2i-1]$, for $i = 0, ..., 7$. For a 16-bit multiplication, the PPG creates 8 partial products, denoted as pp_0 to pp_7. Each partial product is the corresponding multiplicand multiple $Code_i \times md$, contributing to the cumulative result of the multiplication. It should be noticed that forming a negative multiple of the multiplicand can be done by shifting the multiplicand to form the corresponding positive multiple, then negating by the "take the 1's complement and add 1" - except that instead of adding 1 at this phase, it should be postponed to the next phase. We denote these "hot-ones" bits corresponding to partial products by h_0 to h_7.

The partial product reduction phase is commonly implemented using a Wallace CSA tree [Wal64]. The Wallace tree employs 3:2 compressors as fundamental building blocks, with each compressor taking three input bits and producing two output bits - a sum and a carry. Figure 5b illustrates the Wallace CSA tree, highlighting a detailed representation of the output from all 3:2 compressors in their steady state. The sums and carries are labeled s_0 to s_8 and c_0 to c_8, respectively, representing the results of the compression process at different stages in the tree. The number of the sums and carries directly impact the efficiency of the multiplication operation. For instance, a higher number of carries may suggest more complex bit manipulations, potentially leading to increased power consumption.

The outputs from the last compressors (s_8 and c_8) are then fed into a conventional adder to obtain the final result of the multiplication (and accumulation). According to Figure 5b, we can obtain the widths of all intermediate variables, and we record them in Table 2. Thus, the variables $\mathbf{DM}_{(t)}$ of the model can be instantiated as

$$\mathbf{DM}_{(t)} = \left(\mathbf{pp}_{0,(t)}^T, \cdots, \mathbf{pp}_{7,(t)}^T, \mathbf{h}_{0,(t)}^T, \cdots, \mathbf{h}_{7,(t)}^T, \mathbf{s}_{0,(t)}^T, \cdots, \mathbf{s}_{8,(t)}^T, \mathbf{c}_{0,(t)}^T, \cdots, \mathbf{c}_{8,(t)}^T\right)^T,$$

so as the corresponding transition variables $\mathbf{TDM}_{(t)}$.

3.4 Leakage Model and Explanatory Variables Selection

In this subsection, we are committed to completing and refining our cycle-level model to more precisely represent the target instructions shown in Figure 6a, which consist of 10 multiply instructions spanning Lines 4 to 13. This refinement requires careful consideration of the specific characteristics of these instructions to ensure the model's accuracy and relevance.

The **`smuadx`** instruction performs two 16×16 multiplications, necessitating the inclusion of predictor variables for both multipliers. This can be implemented by introducing

- $\mathbf{DMb}_{(t)}$ for the multiplier operating on the lower bits of the multiplicand register,
- $\mathbf{DMt}_{(t)}$ for the multiplier operating on the upper bits of the multiplicand register,

so as the corresponding transition variables $\mathbf{TDMb}_{(t)}$ and $\mathbf{TDMt}_{(t)}$. Thus, the complete cycle-level leakage model is given as follows,

$$y'_{(t)} = \mathcal{Y}'_{(t)}(o_{0,(t)}, o_{1,(t)}, o_{2,(t)}, r_{(t)}, \tilde{o}_{0,(t)}, \tilde{o}_{1,(t)}, \tilde{o}_{2,(t)}, \tilde{r}_{(t)}) = \mathbf{X}_{(t)}^T \boldsymbol{\beta}_{(t)} + \delta_{(t)}, \tag{8}$$

with

$$\mathbf{X}_{(t)} = \left(\mathbf{O}^T_{0,(t)}, \mathbf{O}^T_{1,(t)}, \mathbf{O}^T_{2,(t)}, \mathbf{R}^T_{(t)}, \mathbf{DM}^T_{(t)}, \mathbf{TO}^T_{0,(t)}, \mathbf{TO}^T_{2,(t)}, \mathbf{TO}^T_{2,(t)}, \mathbf{TR}^T_{(t)}, \mathbf{TDM}^T_{(t)}\right)^T,$$

$$\mathbf{DM}_{(t)} = \left(\mathbf{DMb}_{(t)}, \mathbf{DMt}_{(t)}\right)^T, \mathbf{TDM}_{(t)} = \left(\mathbf{TDMb}_{(t)}, \mathbf{TDMt}_{(t)}\right)^T,$$

$$\mathbf{DMb}_{(t)} = \left(\mathbf{ppb}^T_{0,(t)}, \cdots, \mathbf{ppb}^T_{7,(t)}, \mathbf{hb}^T_{0,(t)}, \cdots, \mathbf{hb}^T_{7,(t)}, \mathbf{sb}^T_{0,(t)}, \cdots, \mathbf{sb}^T_{8,(t)}, \mathbf{cb}^T_{0,(t)}, \cdots, \mathbf{cb}^T_{8,(t)}\right)^T,$$
$$\mathbf{DMt}_{(t)} = \left(\mathbf{ppt}^T_{0,(t)}, \cdots, \mathbf{ppt}^T_{7,(t)}, \mathbf{ht}^T_{0,(t)}, \cdots, \mathbf{ht}^T_{7,(t)}, \mathbf{st}^T_{0,(t)}, \cdots, \mathbf{st}^T_{8,(t)}, \mathbf{ct}^T_{0,(t)}, \cdots, \mathbf{ct}^T_{8,(t)}\right)^T.$$

We delineate a systematic process for selecting appropriate predictor variables, integral for constructing a statistically robust model. The variable selection process, designed atop the framework used by McCann et al. [MOW17] and leveraging the F-test, involves analyzing variables converted from single factors as a unit to assess their contribution to the model variable. The selection process is approached from two levels: instruction-level and algorithm-level. At the instruction-level, the focus is on the number of operands, while at the algorithm-level, more emphasis is placed on factors like operand size limitations, particularly as the model targets specific algorithms for attacks.

- **Instruction-Level Selection**

  1. **Unused Operand:** The foundational model covers all multiply instructions in the target function. If a given instruction involves only two operands, $\mathbf{O}_2$ should be excluded. It is also important to note that $\mathbf{TO}_2$ not only relate to the current instruction but also to the number of operands in the previous instruction.
  2. **Same Value Variables:** Frequently in instruction sequences, the result of the previous instruction serves as an operand for the current instruction. In such cases, these variables, which remain same across instructions, can be merged or one can be eliminated.
  3. **Constant Value Variables:** For instructions where the second operand is a constant, this operand's contribution to the dependent variable is fixed and can be merged with the intercept, leading to its exclusion.

- **Algorithm-Level Selection**

  1. **Same Value Variables:** Algorithms may also generate situations where, for example, a multiply instruction computes the square of a number, making the first and second operands identical. In such cases, retaining only one of these operands for modeling is sufficient.
  2. **Constant Value Variables:** Moreover, if an operand is a constant across the program, such a variable should also be excluded from the model.
  3. **Unused Bit Variables:** Typically, operands in instructions are 32-bit, but in some algorithms, the operand range might be limited, e.g., 0-3329, using only the lower 12 bits while the upper 20 bits remain constant at zero. These constant high bits should be excluded from the model.

When modeling the various instructions in the algorithm, different explanatory variables may be excluded. A detailed example is provided in Section 5.2.

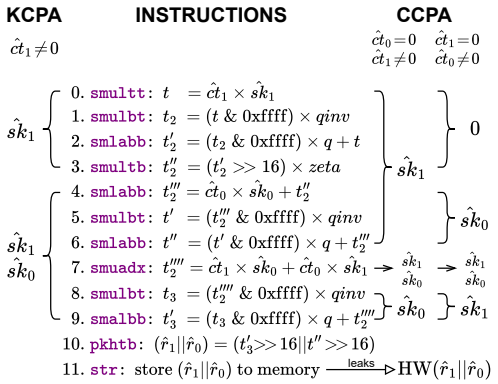
4 Proposed Attacks

Kyber employs an incomplete NTT (cf. 2.1) to convert a single 256-degree polynomial multiplication into 128 first-degree polynomial multiplications, as shown in Algorithm 4. In this section, we focus on a single first-degree polynomial and demonstrate various attacks targeting the recovery of the secret coefficient pairs.

Firstly, we describe the computation process of the first-degree polynomial in the target function in detail, which aids in comprehending the attack strategies (Section 4.1). Then, we propose two profiled attacks against unprotected implementations: one using known ciphertext and the other using chosen ciphertext methods (Section 4.2 and 4.3). Additionally, we present a second-order chosen ciphertext profiled attack against the first-order masked implementation (Section 4.4). Furthermore, the Hamming weight leakage from the **str** instruction can be utilized to assist in filtering candidates, leading us to propose two candidate filtering conditions (Section 4.5).

```
1  ldr.w poly2, [bptr, #4]
2  ldr poly0, [bptr], #8
3
4  smultt t, poly0, poly1
5  smulbt t2, t, qinv       // mont
6  smlabb t2, q, t2, t      // mont
7  smultb t2, t2, zeta
8  smlabb t2, poly0, poly1, t2
9  smulbt t, t2, qinv       // mont
10 smlabb t, q, t, t2       // mont
11 smuadx t2, poly0, poly1
12 smulbt poly0, t2, qinv   // mont
13 smlabb poly0,q,poly0,t2  // mont
14 pkhtb t, poly0, t, asr#16
15 str t, [rptr], #4
```

(a)



(b)

Figure 6: (a) Targeted Thumb instructions: the initial first-degree polynomial multiplication of the macro **doublebasemul_frombytes_asm**. (b) Proposed profiled attacks: a known ciphertext profiled attack is depicted on the left side, and two distinct scenarios of chosen ciphertext profiled attacks are presented on the right side.

4.1 Macro doublebasemul_frombytes_asm

The *pqm4* library [KRSS] provides various implementations of Kyber specifically optimized for the ARM Cortex-M4, with critical operations optimized using assembly codes. We focus on the assembly function **frombytes_mul_asm** in the stack-optimized Kyber implementation [impb], which is based on the work of [AHKS22]. Specifically, we attack the macro **doublebasemul_frombytes_asm** which involves two first-degree polynomial multiplications. Figure 6a shows the Thumb instructions for the initial first-degree polynomial multiplication in the macro **doublebasemul_frombytes_asm**.

Our targeted macro uses Montgomery reduction, which reduces the multiplication product (or accumulated sum) by q, resulting in a range of $(-q, q)$. Under the setting $qinv = 3327 = -q^{-1} \bmod 2^{16}$ (instead of $q^{-1} \bmod 2^{16}$), the subtraction in Montgomery reduction is transformed into an addition, thereby reducing the number of instructions. Notably, $qinv$ and q are stored in a single 32-bit register ($r12$), with the most significant 16 bits representing $qinv$.

As shown in Figure 6a, two pairs of ciphertext coefficients are loaded from the memory. The 32-bit register $poly0$ stores the public coefficient pair $\hat{ct}_1||\hat{ct}_0$, and the register $poly2$ stores $\hat{ct}_3||\hat{ct}_2$ required for the next first-degree polynomial multiplication. Secret coefficient

pairs $\hat{sk}_1||\hat{sk}_0$ and $\hat{sk}_3||\hat{sk}_2$, stored in registers *poly*1 and *poly*3, are loaded using the macro **deserialize** in the function **frombytes_mul_asm**. Therefore, line 4 performs $\hat{sk}_1 \cdot \hat{ct}_1$ and the result is reduced by q in line 5 and 6. Then, the reduced result is multiplied by ζ in line 7, added to the product $\hat{sk}_0 \cdot \hat{ct}_0$ in line 8, and finally reduced in line 9 and 10 to obtain $\hat{r}_1$. In line 11, **smuadx** calculates the sum of products of dual 16-bit signed multiplications in a single cycle. The sum $\hat{ct}_1 \cdot \hat{sk}_0 + \hat{ct}_0 \cdot \hat{sk}_1$ is reduced via Montgomery reduction to get the result $\hat{r}_0$. Line 14 packs the two 16-bit results into the form $\hat{r}_1||\hat{r}_0$ and stores them in the 32-bit register t. Finally, the instruction **str** stores the coefficient pairs $\hat{r}_1||\hat{r}_0$ into the data memory.

4.2 Known Ciphertext Profiled Attack

According to Figure 6b, the first 4 instructions involve the secret coefficient $\hat{sk}_1$, and the following 6 multiply instructions involve both $\hat{sk}_1$ and $\hat{sk}_0$. First, we recover $\hat{sk}_1$, then use the recovered $\hat{sk}_1$ to determine $\hat{sk}_0$.

When recovering $\hat{sk}_1$, it is important to note that $\hat{ct}_1 \neq 0$. Given the public ct which is in the normal domain, we need to use the following expression to verify whether $\hat{ct}_1$ is equal to 0. If $\hat{ct}_1 = 0$, we discard the polynomial.[3]

$$\hat{ct} = (\hat{ct}_0, \hat{ct}_1, ..., \hat{ct}_{255}) = \mathrm{NTT}(\mathrm{Decompress}_q(ct, d)),$$

where $ct = (ct_0, ct_1, ..., ct_{255})$, $ct_i \in \mathbb{Z}_q, i = 0, 1, ..., 255$.

We estimate the parameters $\boldsymbol{\beta}_{(t)}$, $\delta_{(t)}$ of models (Equation 8) for multiply instructions in Figure 6b. As explained in section 4.1, the operands and results of each multiplication (i.e., the independent variables in Equation 8) depend solely on $\hat{sk}_1, \hat{sk}_0, \hat{ct}_1, \hat{ct}_0$. In other words, $y'_{(t)}$ can be fully expressed in terms of $\hat{sk}_1, \hat{sk}_0, \hat{ct}_1, \hat{ct}_0$, thus we rewrite them accordingly.

$$y'_{(t)} = \mathbf{X}^T_{(t)}\boldsymbol{\beta}_{(t)} + \delta_{(t)} = \mathcal{Y}''_{(t)}(\hat{sk}_1, \hat{sk}_0, \hat{ct}_1, \hat{ct}_0) \tag{9}$$

Then, we get the predictors $y'_{(t)}$ and residual vectors $\boldsymbol{\varepsilon}_{(t)}$, where t denotes the index of the instruction that ranges from 0 to 9.

Given a new leakage trace with $\hat{ct}_1 = \hat{ct}^*_1, \hat{ct}_0 = \hat{ct}^*_0$, we get the corresponding cycle-level consumption $y^*_{(0)}, y^*_{(1)}, ..., y^*_{(9)}$ of each multiply instruction. Although $\hat{sk}_0$ is part of the operand in the first multiply instruction **smultt**, it does not participate in the multiplication operation, so we set the value of $\hat{sk}_0$ to 0 in the models when recovering $\hat{sk}_1$. Let

$$\mathbf{y}^*_{\hat{sk}_1} = \left(y^*_{(0)}, \cdots, y^*_{(3)}\right)^T, \tag{10}$$

$$\mathbf{y}'_{\hat{sk}_1} = \left(\mathcal{Y}''_{(0)}(\hat{sk}_1, 0, \hat{ct}^*_1, \hat{ct}^*_0), \cdots, \mathcal{Y}''_{(3)}(\hat{sk}_1, 0, \hat{ct}^*_1, \hat{ct}^*_0)\right)^T, \tag{11}$$

then

$$\boldsymbol{\Sigma}_{\hat{sk}_1} = \frac{1}{n_p - 1}\left(\boldsymbol{\varepsilon}_0, \boldsymbol{\varepsilon}_1, ..., \boldsymbol{\varepsilon}_3\right)^T \cdot \left(\boldsymbol{\varepsilon}_0, \boldsymbol{\varepsilon}_1, ..., \boldsymbol{\varepsilon}_3\right),$$

$$\mathrm{Pd}_{\hat{sk}_1}(\mathbf{y}^*_{\hat{sk}_1}|\hat{sk}_1) = \frac{1}{\sqrt{(2\pi)^4|\boldsymbol{\Sigma}_{\hat{sk}_1}|}}\exp\left(-\frac{1}{2}(\mathbf{y}^*_{\hat{sk}_1} - \mathbf{y}'_{\hat{sk}_1})^T\boldsymbol{\Sigma}^{-1}_{\hat{sk}_1}(\mathbf{y}^*_{\hat{sk}_1} - \mathbf{y}'_{\hat{sk}_1})\right),$$

and the likelihood $\ell(\hat{sk}_1|\mathbf{y}^*_{\hat{sk}_1}) = \mathrm{Pd}_{\hat{sk}_1}(\mathbf{y}^*_{\hat{sk}_1}|\hat{sk}_1)$. Ranks of the candidates $\hat{sk}_1$ are obtained by sort the likelihood values of all candidates.

Given n_a individual leakage traces $\mathbf{Y}^{n_a}$, the joint log-likelihood can be calculated using Equation 4: $\log \ell(\hat{sk}_1|\mathbf{Y}^{n_a}) = \sum_{a=0}^{n_a-1} \ell(\hat{sk}_1|\mathbf{y}^{(a)}_{\hat{sk}_1})$. The candidate with the highest joint log-likelihood value is most likely the correct secret coefficient $\hat{sk}^\star_1$.

[3] For all coefficients in the polynomial, if $\hat{ct}_{2i+1} = 0, i = 0, 1, ..., 127$, we discard the generated polynomial.

We next reconstruct $\hat{sk}_0$ with the recovered $\hat{sk}_1^*$ using the last 6 multiply instructions. The approach is very similar to $\hat{sk}_1$ recovery, i.e.,

$$\mathbf{y}'_{\hat{sk}_0} = \left(\mathcal{Y}''_{(5)}(\hat{sk}_1^*, \hat{sk}_0, \hat{ct}_1^*, \hat{ct}_0^*), \cdots, \mathcal{Y}''_{(9)}(\hat{sk}_1^*, \hat{sk}_0, \hat{ct}_1^*, \hat{ct}_0^*)\right)^T. \tag{12}$$

The remaining steps are not described in detail. It should be noted that the traces used to recover $\hat{sk}_1$ and $\hat{sk}_0$ overlap. To put it another way, a trace can be used to restore both $\hat{sk}_1$ and $\hat{sk}_0$. This applies to all all subsequent attacks.

4.3 Chosen Ciphertext Profiled Attack

By choosing specific ciphertexts, we can predict $\hat{sk}_1$ and $\hat{sk}_0$ individually. Figure 6b illustrates two cases for ciphertext selection.

- Let $\hat{ct}_0 = 1$ and $\hat{ct}_1 \neq 0$, $\hat{sk}_1$ is related to the first 7 instructions, while $\hat{sk}_0$ is related to the last 2 multiply instructions. In this scenario, the specific model for the `smuadx` instruction, which is associated with both $\hat{sk}_1$ and $\hat{sk}_0$, is not used for the attack, as its entangled leakage could complicate the individual recovery of each coefficient.
- Let $\hat{ct}_1 = 1$ and $\hat{ct}_0 \neq 0$, $\hat{sk}_0$ is relevant to the 4^{th} to 6^{th} instructions and $\hat{sk}_1$ is relevant to the last 2 multiply instructions. Compared to case 1, this scenario provides 4 fewer instructions for the attack, potentially reducing the effectiveness of the attack. Thus, we opt to abandon this case in favor of the first one.

The Chosen Ciphertext Profiled Attack (CCPA) shares a similar methodology with the Known Ciphertext Profiled Attack (KCPA) in terms of recovering secret coefficients, therefore we do not provide an elaborate description here. CCPA requires specific ciphertext where $\hat{ct}_0 = 1$ and $\hat{ct}_1 \neq 0$. Compared to the method presented in [HHP+21], we offer a straightforward approach to generate polynomials that meet this requirements. Firstly, we initialize a set $\mathcal{C}$ whose elements remain unchanged after consecutive applications of the Compress_q and Decompress_q functions:

$$\mathcal{C} = \{x | x = \text{Compress}_q(\text{Decompress}_q(x, d), d), x \in \mathbb{Z}_q\}.$$

Next, we generate a ciphertext polynomial ct:

$$ct = (0, ct_1, 0, 0, ..., 0, 0),$$

where $ct_1 \in \mathcal{C}$. Finally, we apply NTT to the ciphertext polynomial $\mathbf{ct}$ to obtain

$$\hat{ct} = \text{NTT}(ct) = (0, ct_1, 0, ct_1, ..., 0, ct_1).$$

4.4 The Second-order Chosen Ciphertext Profiled Attack

In proposed KCPA, recovering $\hat{sk}_0$ depends on the previously recovered $\hat{sk}_1$. This dependency makes it challenging to be applied to masked implementations. For instance, in a first-order masked scheme, each secret coefficient is split into two shares, such as $\hat{sk}_1 = \hat{sk}_{s0,1} + \hat{sk}_{s1,1} \bmod q$, $\hat{sk}_0 = \hat{sk}_{s0,0} + \hat{sk}_{s1,0} \bmod q$, where $\hat{sk}_{s0,1}$, $\hat{sk}_{s1,1}$, $\hat{sk}_{s0,0}$ and $\hat{sk}_{s1,0}$ are refreshed for each decryption. In a single-trace key recovery scenario, reconstructing the secret coefficients from these shares is straightforward. However, in multiple-trace attacks, recovering $\hat{sk}_0$ or $\hat{sk}_1$ from their respective shares ($\hat{sk}_{s1,0}$ and $\hat{sk}_{s0,1}$ for $\hat{sk}_0$, or $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$ for $\hat{sk}_1$) is significantly more complex. Furthermore, the dependency between $\hat{sk}_0$ and $\hat{sk}_1$ indicates that information from all four shares must be integrated when recovering $\hat{sk}_0$, making the attack even more challenging.

Therefore, we opt for CCPA to target the masked implementation, as it is capable of predicting the two coefficients individually. Taking the recovery of $\hat{sk}_1$ in the first-order masked implementation as an example, given a target trace $\mathbf{y}^* = (\mathbf{y}^*_{s0}, \mathbf{y}^*_{s1})^T$ with the chosen ciphertext, we obtain the cycle-level power consumption for share 0: $\mathbf{y}^*_{s0} = \left(y^*_{s0,(0)}, y^*_{s0,(1)}, \cdots, y^*_{s0,(9)}\right)^T$ and share 1: $\mathbf{y}^*_{s1} = \left(y^*_{s1,(0)}, y^*_{s1,(1)}, \cdots, y^*_{s1,(9)}\right)^T$, respectively. We calculate the likelihood for each share of $\hat{sk}_1$, $\ell(\hat{sk}_{s0,1}|\mathbf{y}^*_{s0})$ and $\ell(\hat{sk}_{s1,1}|\mathbf{y}^*_{s1})$. Then, the likelihood of $\hat{sk}_1$ can be computed as follows:

$$\ell(\hat{sk}_1|\mathbf{y}^*) = \prod_{\substack{\hat{sk}_{s0,1}+\hat{sk}_{s1,1} \bmod q = \hat{sk}_1 \\ \hat{sk}_{s0,1}, \hat{sk}_{s1,1} \in \mathbb{Z}_q}} (\ell(\hat{sk}_{s0,1}|\mathbf{y}^*_{s0}) \cdot \ell(\hat{sk}_{s1,1}|\mathbf{y}^*_{s1})), \tag{13}$$

and the log-likelihood of $\hat{sk}_1$ is

$$\log \ell(\hat{sk}_1|\mathbf{y}^*) = \sum_{\substack{\hat{sk}_{s0,1}+\hat{sk}_{s1,1} \bmod q = \hat{sk}_1 \\ \hat{sk}_{s0,1}, \hat{sk}_{s1,1} \in \mathbb{Z}_q}} (\log \ell(\hat{sk}_{s0,1}|\mathbf{y}^*_{s0}) + \log \ell(\hat{sk}_{s1,1}|\mathbf{y}^*_{s1})). \tag{14}$$

By accumulating, we obtain the joint log-likelihood $\log \ell(\hat{sk}_1|\mathbf{Y}^{n_a})$ for given n_a leakage traces $\mathbf{Y}^{n_a}$ using Equation 4.

The recovery of $\hat{sk}_0$ follows a similar procedure, hence the steps are not reiterated. This approach allows for the individual estimation of the likelihoods of the shares, which is essential for the recovery of secret coefficients in a first-order masked implementation.

In a real-world attack, employing Equation 14 for the log-likelihood of $\hat{sk}_1$ presents challenges due to the accumulation of log-likelihoods across q^2 pairs of $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$. Only one pair corresponds to the correct secret coefficient shares, while the remaining $q^2 - 1$ are incorrect, significantly increasing the rank of the correct coefficient complicating its recovery.

To mitigate this, one strategy is to prune some of the incorrect pairs, thereby increasing the relative proportion of the correct one. Typically, with a single leakage trace, the correct share of the secret coefficient ranks relatively high within the candidate set. Thus, a threshold can be set to eliminate candidates whose ranks exceed this threshold for each share.

This pruning process involves the following steps:

1. **Rank Calculation**: Determine the ranks of each candidates $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$ based on the computed log-likelihood from a single trace.

2. **Threshold Setting**: Set a rank threshold th_{rank}, below which candidate pairs are considered unlikely to be correct.

3. **Pruning**: Eliminate candidate pairs whose ranks are lower than the set threshold for both shares $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$.

By implementing this pruning strategy, the attack can focus on a smaller set of more probable candidates. This increases the likelihood of successfully recovering the secret coefficients. This approach narrows down the search space and improve the efficiency of our attack in practical scenarios.

4.5 Assistance with the Store Instruction

As we described in Section 1, using a single store instruction to save two result coefficients presents challenges for the attack. However, the Hamming weight of the result coefficient pair can assist in reducing the number of target traces required. Assuming that $\hat{r}^*_1$ and

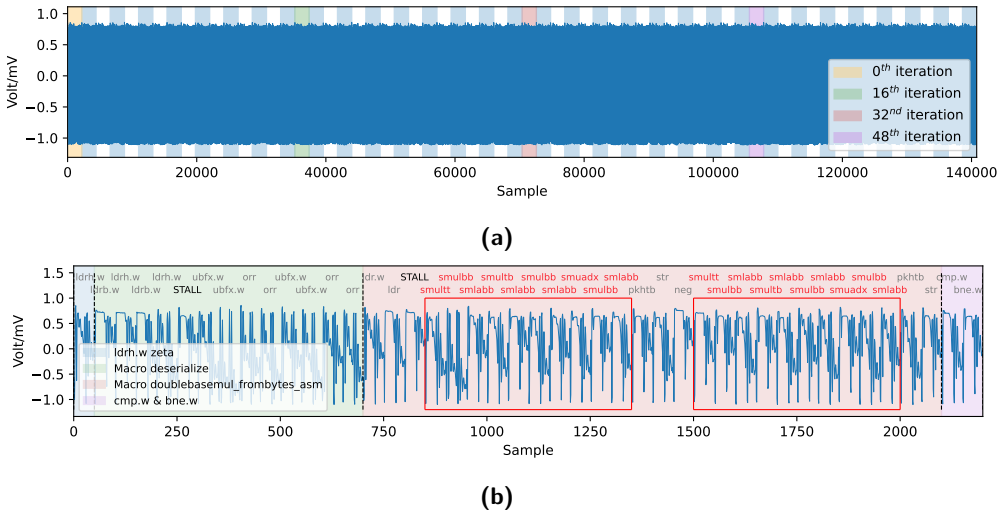


Figure 7: (a) The power trace of the function **frombytes_mul_asm** in the unprotected stack-optimized Kyber implementation running on the Cortex-M4. (b) The power traces of a single iteration which contains a target macro **`doublebasemul_frombytes_asm`**.

$\hat{r}_0^*$ are the real result coefficients, the Hamming weight of $(\hat{r}_1||\hat{r}_0)$ can be extracted using the **`str`** instruction via a template attack, as described in [CRR03, RO04]. The template matching results are demonstrated next section.

We propose two conditions to utilize $\mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$:

- Cond**1** : $\mathrm{HW}(\hat{r}_0), \mathrm{HW}(\hat{r}_1) \leq \mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$;
- Cond**2** : $\mathrm{HW}(\hat{r}_0) + \mathrm{HW}(\hat{r}_1) = \mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$.

When recovering any secret coefficient, the value of either $\hat{r}_0$ or $\hat{r}_1$ can only be hypothesized, thereby limiting the application of Cond2 to the recovery of a single coefficient within a secret coefficient pair, rather than both at the same time. Cond2 is only applicable to the recovery of the second coefficient once the first has been successfully retrieved.

These conditions are applied to select candidates that satisfy the specified criteria. By filtering the candidates based on the Hamming weight conditions, the attacker can more efficiently identify the correct secret coefficients. This reduces both the complexity and the number of traces required for a successful attack.

5 Experiments and Results

In this section, we define and declare the notations used throughout this chapter. n_c denotes the number of traces for profiling. n_p represents the number of trace fragments for profiling. n_{hw} is the number of HW candidates ordered by ranks. n_a is the number of traces required for recovering secret coefficients. th_{rank} is the rank threshold value used in the second-order attacks, as described in Section 4.4. These notations will be consistently applied to describe the experimental settings and results presented in this section.

5.1 Preprocessing

Traces Collection The traces used for model construction are collected from the profiling device using random ciphertext and secret polynomials. The number of traces for

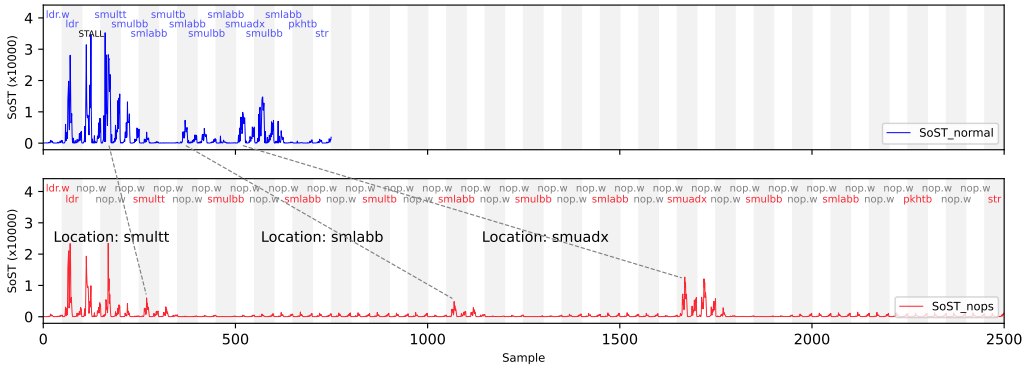


Figure 8: Instruction location via SoST. The upper figure displays the SoST values (indicated by the blue trace) at each sampling point generated by the normal instruction sequence shown in Figure 6a. The lower figure presents the SoST values (indicated by the red trace) resulting from the instruction sequence with inserted **`nop.w`** operations. The alternating gray and white regions represent individual clock cycles.

constructing accurate models is discussed in section 5.2. The traces for recovering secret coefficients are captured from the target device.

Reference Trace A reference trace, used as a baseline, is needed to align the collected power traces and ensure that they are properly aligned. Clock signals are instrumental in understanding the cycle information of instructions; hence, we use a power trace carrying a clock signal as the reference.

Trace Cutting The Kyber polynomial multiplication function **`frombytes_mul_asm`** consists of 64 iterations (128 iterations for the first-order protected Kyber), with each iteration calling the macro **`doublebasemul_frombytes_asm`** to performs two first-degree polynomial multiplication operations. Therefore, we split the power trace of **`frombytes_mul_asm`** into 64 segments (128 segments), as shown in Figure 7a. Figure 7b presents the power trace of a single iteration, with our targeted macro highlighted in pink. In each macro, the multiplication instruction parts are marked with red solid boxes. The benefits of trace cutting are twofold: firstly, by locating the instructions within a single macro, we can determine the instruction locations for all macros; secondly, when modeling the power consumption of the instructions, the derived 64 (128) trace segments can be used for profiling, separately, which significantly reduces the number of traces required for profiling.

Traces Alignment We employ cross-correlation to calculate the offset between two traces. The Python NumPy package [HMvdW+20] provides a function called *correlate*, which is used to compute the cross-correlation of two one-dimensional sequences of length n. The function outputs correlation values for offsets ranging from $-n+1$ to $n-1$. By identifying the offset corresponding to the maximum correlation value, we can properly align the trace by shifting it accordingly.

Instructions Location We first locate the load instructions, which exhibit significant variance at the sampling points compared to arithmetic or logic instructions, making them easily distinguishable. We set $ct_i = 0$ for i ranging from 2 to 255, and randomly generate ct_0 and ct_1, ensuring that they satisfy $ct_j = \text{Decompress}_q(\text{Compress}_q(ct_j, d), d)$, for $j = 0$ or 1. Thus, in the NTT domain, the ciphertext polynomial is $\hat{ct} = (ct_0, ct_1, ct_0, ct_1, \cdots, ct_0, ct_1)$. For example, $ct = (16, 1365, 0, 0, \cdots, 0, 0)$, and its corresponding NTT transformation is $\hat{ct} = \text{NTT}(ct) = (16, 1365, 16, 1365, \cdots, 16, 1365)$. We categorize the randomly generated ciphertexts based on $\text{HW}(ct_0||ct_1)$. Since $ct_0, ct_1 \in \mathbb{Z}_q$, we obtain 23 groups, with 100

ciphertext polynomials retained in each group. We input the ciphertext polynomials and a secret polynomial where all coefficients are set to zero, and then collect the corresponding power traces.

We calculate the mean $m_{i,(t)}$ and standard deviation $\sigma_{i,(t)}$ at the t^{th} sampling point of the i^{th} group, and apply the sum-of-squared t-values (SoST) [GLP06] for locating the load instruction.

$$\mathrm{SoST}_{(t)} = \sum_{i,j=0}^{20} \left(\frac{m_{i,(t)} - m_{j,(t)}}{\sqrt{\frac{\sigma^2_{i,(t)}}{n_i} + \frac{\sigma^2_{j,(t)}}{n_j}}} \right)^2 ,$$

where $n_i = n_j = 100$ represent the numbers of traces in the i^{th} and j^{th} group, and $\mathrm{SoST}_(t)$ represents the SoST value at the t^{th} sampling point.

It is worth noting that load/store instructions perform the memory access operation in the clock cycle following the **EX** stage, and the SoST peak is expected to occur during this clock cycle. As shown by the blue trace in Figure 8, the peak observed in the second cycle results from the `ldr.w` instruction which loads $\hat{ct}_3||\hat{ct}_2$ (the ciphertext coefficient pair for the next first-degree polynomial multiplication). Similarly, The peak observed in the third cycle is due to the `ldr` instruction which loads $\hat{ct}_1||\hat{ct}_0$.

One straightforward approach to locate the multiply instructions is to determine the instructions corresponding to the clock cycles immediately following `ldr` based on the instruction sequence. However, considering the potential pipeline stalls caused by load instructions, which can delay the execution of subsequent instructions by one clock cycle, we propose a method for locating instruction with improved accuracy. We modify the source code by adding dummy operations to the macro `doublebasemul_frombytes_asm`. In addition to consecutive load instructions, we insert three `nop.w` instructions between each pair of adjacent instructions to introduce precise delays and isolate each instruction. With the same ciphertext used for locating the load instruction, we collect power traces and calculate the SoST values. As shown in Figure 8, by analyzing the delays introduced by the `nop.w` instructions, we can accurately determine the position of each multiply instruction in the red trace and deduce their positions in the blue trace which does not include the dummy operations.

5.2 Models Construction and Evaluation for Multiply Instructions

Models Construction After preprocessing, we obtain trace fragments that include the first-degree polynomial multiplications. We employ the raw integration method [MOP07, Chapter 4.5] to compress the traces. We sum the sampling point data for each multiply instruction over its corresponding cycle to determine the cycle-level power consumption, denoted as $\mathbf{y}_{(t)}$, for $t = 0$ to 9, as displayed in Figure 9a. Using the inputs from each single polynomial multiplication, we calculate the explanatory variables of the models for instructions, as shown in Figure 9b. Utilizing the variable selection method proposed in section 3.4, we eliminate some unnecessary variables. The set of excluded explanatory variables may vary across different models. For each multiply instruction within the macro `doublebasemul_frombytes_asm`, the variable selection process are detailed in Table 9. Additionally, we apply the significant F-test for further refinement of the variable selection, which is discussed in detail in the subsequent paragraph. By solving the least squares problem, we derive the estimated coefficients $\boldsymbol{\beta}_{(t)}$, $\delta_{(t)}$ and residual terms $\boldsymbol{\varepsilon}_{(t)}$ for the attacks, as shown in Figure 9c.

Models Evaluation We use n_c to denote the number of traces for modeling, and n_p to represent the number of trace fragments for profiling. Thus, n_p is equal to $64 \cdot n_c$ and $128 \cdot n_c$ for the unprotected and the first-order masked implementations, respectively. For

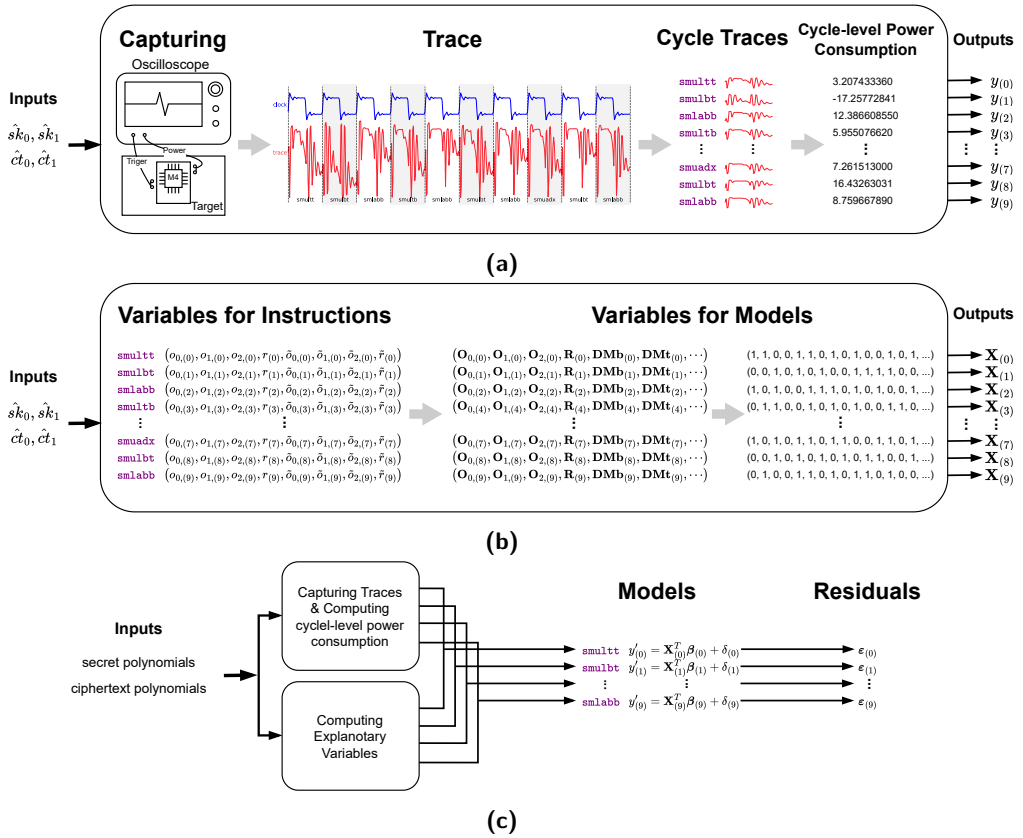


Figure 9: Preprocessing. (a) Capturing traces and computing the cycle-level consumption. (b) Calculating the variables of the models. (c) Generating the models for multiply instructions.

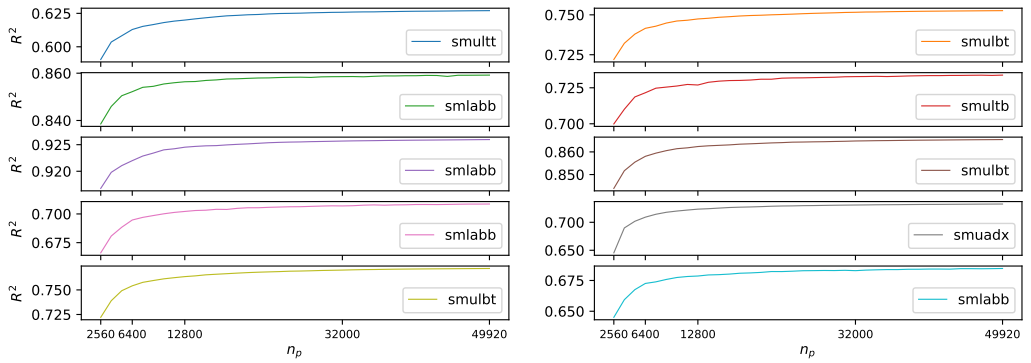


Figure 10: The coefficient of determination R^2 of each model with increasing trace fragments for profiling. The initial number of trace fragments should be larger than the number of explanatory variables.

different values of n_p, we test the coefficient of determination R^2 for each model. Figure 10 illustrates the trend of R^2 for the 10 models as n_p increases. n_p should exceed the maximum number of variables in all 10 models, and therefore, we choose $n_p = 2{,}560$ as the starting point in our tests. Between 2,560 and 32,000, the increase in R^2 is relatively pronounced, whereas from 32,000 to 49,920, there is little to no noticeable growth. Table

Table 3: The coefficient of determination R^2 of each model with different numbers of trace fragments for profiling.

	Inst. / n_p	`smultt`	`smulbt`	`smlabb`	`smultb`	`smlabb`
R^2	6,400	0.6129	0.7415	0.8522	0.7216	0.9220
	12,800	0.6200	0.7474	0.8564	0.7269	0.9245
	32,000	0.6258	0.7516	0.8586	0.7328	0.9257
	49,920	0.6270	0.7526	0.8592	0.7338	0.9260
	Inst. / n_p	`smulbt`	`smlabb`	`smuadx`	`smulbt`	`smlabb`
R^2	6,400	0.8581	0.6947	0.7092	0.7542	0.6726
	12,800	0.8624	0.7022	0.7235	0.7635	0.6786
	32,000	0.8648	0.7070	0.7309	0.7702	0.6830
	49,920	0.8654	0.7086	0.7326	0.7718	0.6847

3 provides the R^2 values for the 10 models at four different values of n_p, with the values rounded to four decimal places.

To confirm the significance of each explanatory variable in the models, an F-test is performed on all terms. Table 9 shows the F-statistic values for the terms tested in all models given $n_p = 49,920$ (780 traces for the unprotected implementation or 390 traces for the first-order masked implementation). In all cases, variables $\mathbf{DM}_{(t)}$ and $\mathbf{TDM}_{(t)}$ are statistically significant at the 5% level, indicating that our simulation of the combinational logic is effective. Variables that did not reach statistical significance (i.e., tests which fail to reject at the 5% level) are considered for exclusion from the model to maintain model statistical robustness.

5.3 Recovering $\mathbf{HW}(\hat{r}_1^*||\hat{r}_0^*)$

Subsection 4.5 has demonstrated that utilizing the HW information of the str instruction, $\mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$, can assist in KCPA or CCPA. However, the accuracy of recovering $\mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$, using template matching should be taken into account. This subsection provides a detailed analysis of the impact of the template matching on filtering secret coefficients.

The store instruction leaks the HW information of $(\hat{r}_1^*||\hat{r}_0^*)$ over two clock cycles. During the first clock cycle, the **EX** stage of the **str** instruction processes the output $(\hat{r}_1^*||\hat{r}_0^*)$ from the previous instruction, overlapping with the **RS** stage of the **pkhtb** instruction. In the second clock cycle, the **str** instruction accesses memory. For each HW value, we collect 2,000 trace fragments to construct the corresponding template[4]. Furthermore, we utilize 1,000 additional trace fragments to evaluate the matching probabilities and rank the candidates accordingly.

The HW candidates obtained through template matching are represented as $hw_0, hw_1, \ldots, hw_{32}$, ordered by their ranks. Table 4 shows the cumulative probability of the top n_{hw} ranks, representing the likelihood that the HW assumes one of these values. The correct HW value is the top-ranked candidate (rank 0) with an approximate probability of 80%, and there is a nearly 98% probability that it falls within the top five ranks. Moreover, the correct HW consistently appears among the top eight ranks.

The recovered HW value is utilized in the two conditions proposed in Subsection 4.5 to discard incorrect secret coefficients while preserving the correct one. Depending on the number of ranks employed, Cond**1** and Cond**2** can be reformulated as follows:

[4]The trace fragments used to construct the models for multiply instructions can be (partially) reused when generating the HW templates.

Table 4: The probability that the correct HW value falls in the top n_{hw} ranks.

n_{hw}	1	2	3	4	5
Pr	79.5%	86.7%	91.8%	95.4%	97.8%
n_{hw}	6	7	8	9	10
Pr	99.3%	99.9%	100%	100%	100%

- Cond**1** : $\mathrm{HW}(\hat{r}_0), \mathrm{HW}(\hat{r}_1) \leq \max\{hw_0, ..., hw_{n_{hw}-1}\}$ [5];
- Cond**2** : $\mathrm{HW}(\hat{r}_0) + \mathrm{HW}(\hat{r}_1) \in \{hw_0, \ldots, hw_{n_{hw}-1}\}$.

The probability that Cond**2** retains the correct secret coefficient is directly demonstrated in Table 4. For instance, when only the top-ranked candidate, hw_0, is considered, this probability is approximately 79.5%. As n_{hw} increases from 1 to 8, the probability that Cond**2** retains the correct secret coefficient reaches 100%. However, this comes at the expense of a diminished ability to filter out incorrect secret coefficient guesses.

Table 4 is not directly applicable to Cond**1**, as this condition may still hold even when we obtain a wrong estimation of $\mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$. For instance, we consider the top-ranked candidate hw_0. Cond**1** may remain valid in cases where either $hw_0 > \mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$ or $hw_0 < \mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$. Understanding the first case is straightforward. $\mathrm{HW}(\hat{r}_0), \mathrm{HW}(\hat{r}_1) \leq \mathrm{HW}((\hat{r}_1^*||\hat{r}_0^*)) < hw_0$ holds indicating Cond**1** will not filter out the correct secret coefficient. In case of $\mathrm{HW}(\hat{r}_0), \mathrm{HW}(\hat{r}_1) \leq hw_0 < \mathrm{HW}((\hat{r}_1^*||\hat{r}_0^*))$, the effectiveness of Cond**1** depends on the specific values of $\mathrm{HW}(\hat{r}_0)$ and $\mathrm{HW}(\hat{r}_1)$. Since the probability that Cond**1** works in case 2 cannot be precisely determined, we only presents the probability of $\max\{hw_0, \ldots, hw_{n_{hw}-1}\} \geq \mathrm{HW}(\hat{r}_1^*||\hat{r}_0^*)$ in Table 5. The actual probability should be higher since we exclude certain scenarios in case 2.

Table 5: The probability that the correct HW value is no larger than the maximum value of the top n_{hw} candidates.

n_{hw}	1	2	3	4	5
Pr	89.1%	94.2%	96.7%	97.6%	98.7%
n_{hw}	6	7	8	9	10
Pr	99.7%	100%	100%	100%	100%

Since the attacks proposed in this paper are multi-trace rather than single-trace attacks, there is no strict requirement for every target trace to satisfy Cond**1** or Cond**2**. Therefore, a tradeoff exists between the probability of retaining the correct secret coefficients and the ability to eliminate incorrect secret coefficient guesses. The performance of KCPA and CCPA enhanced with Cond**1** and Cond**2** is analyzed in detail in the following subsection.

5.4 Attacks and Performance

5.4.1 The First-order Attacks

In the evaluation of the performance of KCPA and CCPA on unprotected implementations, we assess the number of traces needed to recover a pair of secret coefficients, ensuring a success rate of at least 99.99% under different values of n_p.

Table 6 displays the data in a comparative format, with each data point consisting of two values: the number of traces required to recover $\hat{sk}_1$ on the left and $\hat{sk}_0$ on the right. Since each trace can be used to recover both $\hat{sk}_1$ and $\hat{sk}_0$, the larger value of the two

[5] We use $\max\{hw_0,...,hw_{n_{hw}-1}\}$ to minimize the probability of incorrectly discarding the correct secret coefficient.

[6] This notation is consistently used in the tables throughout the paper.

Table 6: The performance of attacks against the unprotected stack-optimized Kyber implementation. "Cond**1**" and "Cond**2**" represent that the attacks use the filtering conditions which exploit the HW information leaked from the store instruction. The symbol "-" represent that Cond**2** is inapplicable to recover the $\hat{sk}_1$.[6]Additional 67,000 trace fragments are used for the template attack targeting the store instruction.

Performance	n_c	n_p	KCPA		CCPA		
			Original	Cond**2** (n_{hw}=8)	Original	Cond**1** (n_{hw}=1)	Cond**2** (n_{hw}=6)
n_a for $\hat{sk}_1 \vert \hat{sk}_0$	100	6,400	**13**|10	−(**13**)|7	8|**34**	7|**28**	−(7)|**14**
	200	12,800	**13**|10	−(**13**)|7	8|**29**	7|**25**	−(7)|**14**
	500	32,000	**12**|9	−(**12**)|6	8|**27**	7|**24**	−(7)|**13**
	780	49,920	**12**|9	−(**12**)|6	8|**26**	7|**23**	−(7)**13**

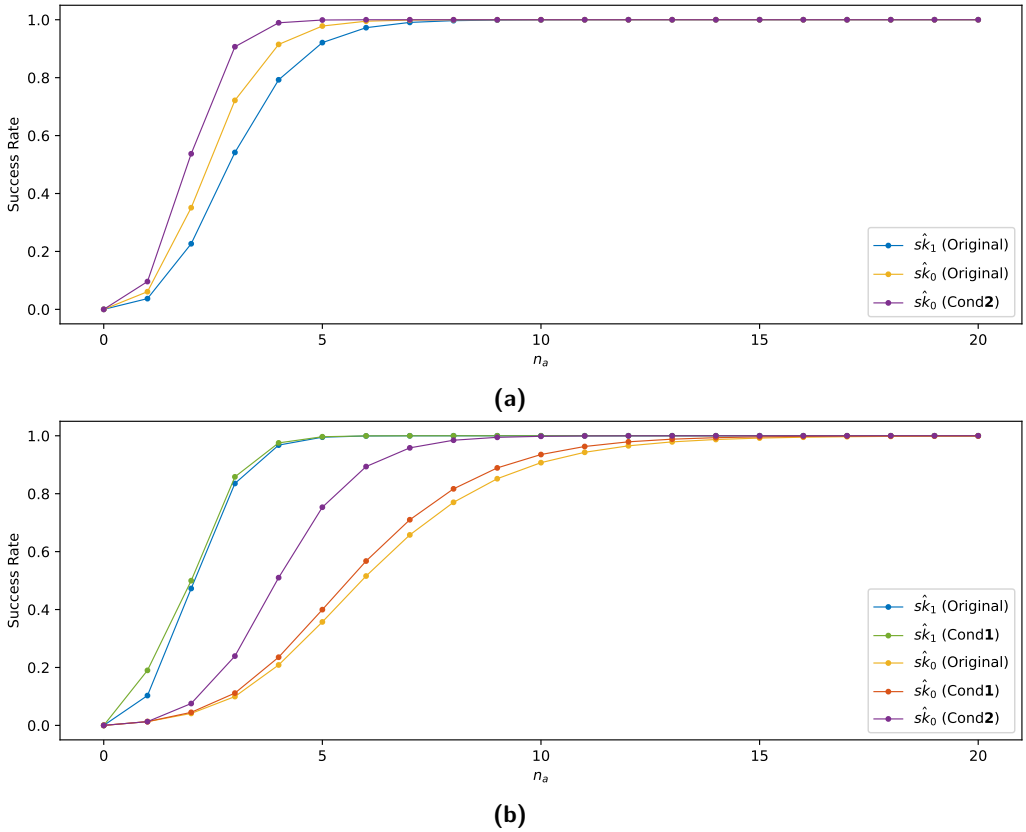


Figure 11: The success rate of the attacks against the unprotected implementation. (a) The success rate of KCPA with $n_p = 49,920$. For the Cond**2**-augmented KCPA, $n_{hw} = 8$. (b) The success rate of CCPA with $n_p = 49,920$. For the Cond**1**-/Cond**2**-augmented CCPA, $n_{hw} = 1/6$.

represents the total number of traces needed to recover the secret coefficient pair. This value is highlighted in bold in the table.

When examining the table horizontally, we provide results for both KCPA and CCPA in two scenarios: the original attack with no assistance and the attack combined with $\mathrm{HW}(\hat{r}_1||\hat{r}_0)$ as auxiliary information. In the original attack scenario, KCPA requires fewer traces compared to CCPA. For instance, when $n_p = 6,400$, KCPA requires 13 traces, while CCPA requires 34. This discrepancy arises because, in KCPA, the recovery of $\hat{sk}_1$ utilizes

leakage from 4 instructions, whereas in CCPA, the recovery of $\hat{sk}_0$ only utilizes leakage from 2 instructions, thereby increasing the attack cost.

In the enhanced attack scenario, Cond**1** utilizes the Hamming weight of $\hat{r}_1$ or $\hat{r}_0$ to filter candidates for the secret coefficients. As shown in Figure 6b, the first 7 instructions are used to generate $\hat{r}_0$. In KCPA, the recovery of $\hat{sk}_1$ only employs the first 4 instructions, making Cond**1** inapplicable to KCPA. Cond**2** is exclusively applied to the recovery of $\hat{sk}_0$. [7] The numerical values within parentheses come from the original or the Cond**1**-augmented attack of the same row to represent a complete attack. In other words, the original attack or attack enhanced with Cond**1** is utilized to recover $\hat{sk}_1$, and the Cond**2**-augmented attack is utilized to recover $\hat{sk}_0$.

As discussed in the previous subsection, the choice of n_{hw} affects both the efficacy of Cond**1** and Cond**2** in eliminating incorrect secret coefficient guesses and the probability of retaining the correct candidates. Thus, we investigate the effects of varying n_{hw} (ranging from 1 to 8) and n_p (from 6400 to 49,920) on the performance of the Cond**1**-augmented and the Cond**2**-augmented attacks, as summarized in Table 10. The results indicate that for Cond**1**, CCPA achieves its highest effectiveness with n_{hw}=1, requiring the fewest traces. In contrast, for Cond**2**, increasing n_{hw} leads to improved performance due to stricter conditions for retaining correct secret coefficients compared to Cond**1**. Specifically, when n_{hw}=8, KCPA enhanced with Cond**2** achieves the best performance with the minimum number of traces. Similarly, when n_{hw}=6, the Cond**2**-augmented CCPA achieves its optimal performance. The corresponding results are also presented in Table 6.

When examining Table 6 vertically, it is observable that as n_p increases, the number of traces required for the attack decreases gradually. The number of required traces sees only a small drop when the attack only needs few traces. For example, when $n_p = 6,400$, the original KCPA requires 13 traces to recover a pair of secret coefficients, and when $n_p = 49,920$, the required traces only decreases by only one. However, when the attack requires more traces, to a certain extent, a larger n_p lead to fewer traces required. This trend is evident when no conditions are applied for optimization or when only Cond**1** is used in the CCPA attack.

Figure 11 illustrates the relationship between the success rate of two profiled attacks and the number of traces for attack, for recovering a pair of secret coefficients. For clarity, only the success rate curves for $n_p = 49,920$ are shown. In Figure 11a, the success rate of recovering $\hat{sk}_0$ using the original KCPA is higher than that of recovering $\hat{sk}_1$ given a same n_a. We can also observe that the application of Cond**2** significantly enhances the success rate of recovering $\hat{sk}_0$. When $n_a = 7$, all three curves exceed a success rate of 99%. As shown in Figure 11b, the success rate of recovering $\hat{sk}_1$ using CCPA is significantly higher than that of recovering $\hat{sk}_0$ for the same n_a. When $n_a = 15$, the success rate for all curves exceeds 99%. Overall, the findings in Figure 11 are consistent with the results presented in Table 6.

In summary, according to the data in the table, the best performance against unprotected implementations is achieved when $n_p = 49,920$. Under this condition, KCPA and CCPA require 12 and 26 traces, respectively, for the original attacks, and 12 and 13 traces, respectively, for the Cond**2**-augmented attacks. Considering these measurements for attacks, recovering all secret coefficient Kyber-768, we have the success rates 97.15% and 96.50% for original KCPA and CCPA, 97.15% and 97.16% for Cond**2**-augmented attacks.

[7] In KCPA, $\hat{sk}_0$ recovery relies on $\hat{sk}_1$, so Cond**2** targets $\hat{sk}_0$. In CCPA, $\hat{sk}_0$ requires more traces than $\hat{sk}_1$, thus we use Cond**2** for $\hat{sk}_0$ after recovering $\hat{sk}_1$ with the original or Cond**1**-augmented attack.

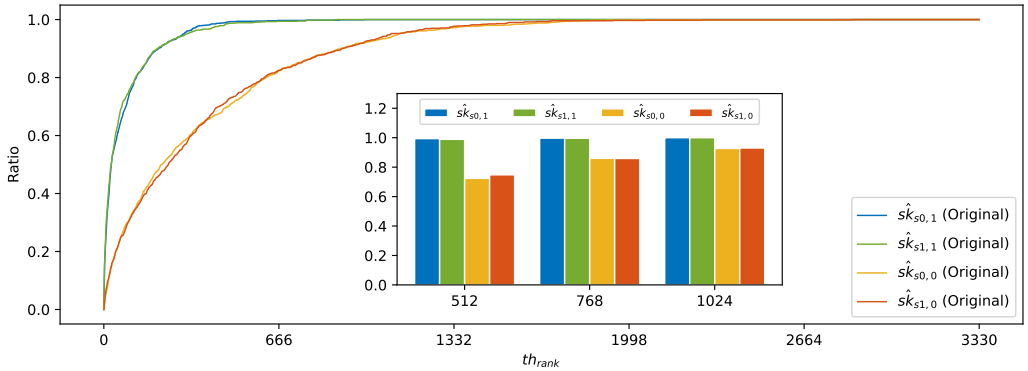


Figure 12: The value of rank ratio with the increasing rank threshold. The subplot presents the rank ratio values at three designated threshold levels, 512, 768, and 1,024.

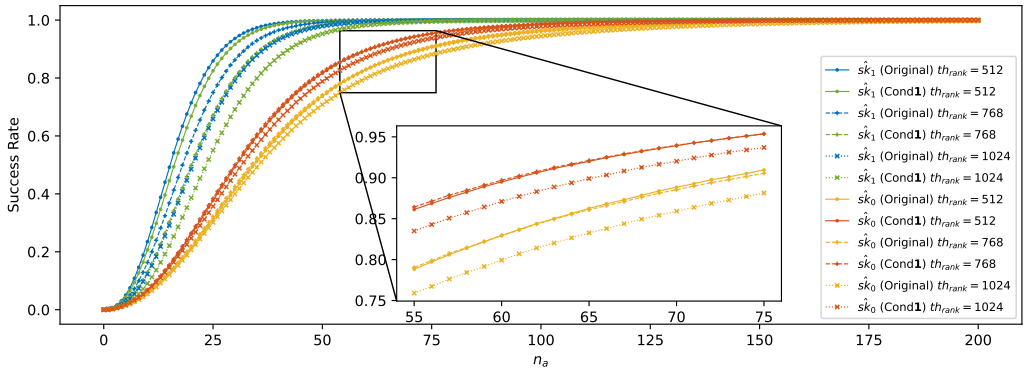


Figure 13: The success rate of a second-order attack at the designated threshold value with $n_p = 49,920$. For the Cond**1**-augmented CCPA, $n_{hw} = 1$.

Table 7: The performance of attacks against the first-order masked stack-optimized Kyber implementation. "Cond**1**" and "Cond**2**" represent that the attacks use the filtering conditions which exploit the HW information leaked from the store instruction. Additional 67,000 trace fragments are used for the template attack targeting the store instruction.

Performance	n_c	n_p	Rank<512		Rank<768		Rank<1,024	
			Original	Cond**1** (n_{hw}=1)	Original	Cond**1** (n_{hw}=1)	Original	Cond**1** (n_{hw}=1)
n_a for $\hat{sk}_1 \vert \hat{sk}_0$	50	6,400	69|**325**	73|**245**	86|**309**	103|**242**	108|**341**	131|**264**
	100	12,800	67|**264**	71|**214**	86|**271**	99|**207**	106|**299**	128|**230**
	250	32,000	66|**264**	70|**214**	85|**270**	97|**208**	104|**296**	126|**227**
	390	49,920	67|**259**	70|**201**	86|**272**	97|**202**	106|**296**	126|**222**

5.4.2 The Second-order Attack

Before evaluating the performance of CCPA against a first-order masked implementation, we demonstrate how rank threshold values are selected. Given 1000 traces, the rank of the correct candidate value is computed for each trace. The proportion of traces where this rank is higher than a given rank threshold is referred to as the rank ratio. Figure 12 illustrates the changes in rank ratio for various secret coefficient shares as the threshold increases. It can be observed that the rank ratios of $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$ reach 1.0 more quickly than those of $\hat{sk}_{s0,0}$ and $\hat{sk}_{s1,0}$. This is because the recovery of $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$

utilizes leakages from 7 instructions, which is 5 more than the number of instructions used for $\hat{sk}_{s0,0}$ and $\hat{sk}_{s1,0}$.

As mentioned in the previous section, a smaller threshold is advantageous for attacks. However, the value of the rank ratio is also crucial; if the rank ratio is too low, the attack will be more complicated. In practice, three threshold values are considered: 512, 768, and 1,024. As illustrated in the subplot of Figure 12, under the three thresholds, the rank ratios of the correct candidates for $\hat{sk}_{s0,1}$ and $\hat{sk}_{s1,1}$ consistently remain above 98%. In contrast, the ratios for $\hat{sk}_{s0,0}$ and $\hat{sk}_{s1,0}$ are comparatively lower and increase as the threshold value increases. This trend aligns with the growth patterns observed in the curves of Figure 12.

In the second-order attack, we also examine the effects of n_{hw} values (ranging from 1 to 8), n_p values (from 6400 to 49,920), and th_{rank} values (ranging from 512 to 1024) on Cond**1**'s performance. The experimental results in Table 11 indicate that when n_{hw}=1, the recovery of key coefficient pairs requires the fewest traces. The corresponding data from this column are also provided in Table 7.

Table 7 presents the evaluation of CCPA against the first-order masked implementation. As shown in Table 7, the number of traces required for the attack decreases as n_p increases, which is consistent with the effect demonstrated in the first-order attack. The value of the rank threshold does not have a direct positive or negative correlation with the number of traces required. When $n_p = 6,400$, both the original attack and the Cond**1**-augmented attack require the fewest traces at a threshold of 768, requiring 309 and 242 traces, respectively. When n_p is 12,800 or 32,000, the original attack requires the fewest number of traces at a threshold of 512, with 264 traces, while the Cond**1**-augmented attack obtains the minimum number of traces at a threshold of 768, with 207 or 208 traces.

Figure 13 illustrates the recovery success rates of $\hat{sk}_1$ and $\hat{sk}_0$ as n_a increases under different threshold values. Overall, fewer traces are needed to recover $\hat{sk}_1$ compared to $\hat{sk}_0$. Additionally, applying Cond**1** significantly improves the success rates. The impact of threshold selection on the success rate is consistent with the results shown in Table 7.

In this paper, the best performance against the first-order masked implementation is achieved when $n_p = 49,920$ and the rank threshold is set to 512. Under these conditions, 259 traces are required for the original attack and 201 traces for the Cond**1**-augmented attack. Considering these measurements for attacks, recovering all secret coefficient in Kyber-768, we have the success rates 96.31% for original attack, 96.31% for the Cond**1**-augmented attack.

5.5 Comparison

Recently, [MWK+24] and [ABB+24] target the same or similar implementation as our work. Table 8 compares our work with them on the number of traces required for profiling and attack, along with the attack success rates, in unprotected or masked implementations. We show the number of traces for profiling, the required traces for attack and successful rate when recovering all secret coefficient of Kyber-768.

Mujdei et al. [MWK+24] utilize a CPA attack to predict two secret coefficients simultaneously, targeting the pair-pointwise multiplications in the unprotected stack-optimized implementation [impb]. The authors accurately recover all secret coefficients using 200 power traces. In their work, guessing one pair of coefficients takes roughly 5 minutes on average. Our work recovers all secret coefficients of a polynomial with a success rate exceeding 99% using as few as 12 traces for the original KCPA and 13 traces for the CCPA enhanced with Cond**2**. The recovery time per coefficient pair is approximately 5 minutes for KCPA and 4.4 minutes for the enhanced CCPA.

Bock et al. [ABB+24] also target the unprotected implementation [SAB+], and give a single trace attack with 65% success rate to recover secret coefficients of three polynomials in Kyber-768. They match a secret pair once time and require 44.5 million templates that each template is exactly one trace. In contrast, we need more traces for attacks, but we only

use 780 traces for profiling (and additional 67,000 trace fragments for template attacks). For the first-order masked implementation, they utilize a template attack to recover individual shares of the secret coefficients using a single trace, thereby reconstructing the real secret coefficients. With a template database of 78 million templates, the success rate of their single-trace attack is 43%. When the template size is increased to 105 million, the success rate reaches 90%. Although our attack requires more traces, with at least 201 traces needed for the CCPA with Cond**1**, our profiling phase requires only 390 traces (and additional 67,000 trace fragments for template attacks), and our success rates are close to 100%. The time cost of [ABB+24] heavenly depends on the traces collection. They capture 1500 traces per minute, while we just need several minute for all traces acquirement. While Bock et al. [ABB+24] do not report the recovery time, our attacks take approximately 4 hours and 3 hours, respectively.

Moreover, it should be noted that the experimental setup of [ABB+24] uses the same device for both profiling and attack, which is the most favorable environment for an attacker. In contrast, we use separate devices for profiling and attack, which more closely resembles real-world scenarios.

Table 8: Comparison with the related works targeting the same or similar implementations. The successful rates of our attacks are presented for recovering all secret coefficient in Kyber-768. Additional 67,000 trace fragments are used for the template attack targeting the store instruction.

Works	Implementation	# traces for profiling	# traces for attack	SR	Across devices
Our work (KCPA)	Unprotected	**780**	12 (Original) 12 (Cond2)	97.15% 97.15%	Yes
Our work (CCPA)	Unprotected	**780**	26 (Original) 13 (Cond2)	96.50% 97.16%	Yes
Our work (CCPA)	Masked	**390**	259 (Original) 201 (Cond1)	96.31% 96.31%	Yes
[MWK+24]	Unprotected	-	200	-	-
[ABB+24]	Unprotected [SAB+]	43M	1	$\approx 100\%$	No
[ABB+24]	Masked [HKL+]	78M 105M	1 1	43% 90%	No

6 Discussion

Linear Regression-based Leakage Model The foundation of our leakage model's applicability to attacks rests on the premise that the inputs and outputs of the targeted instructions correlate exclusively with a single cryptographic secret coefficient. For instance, in a first-order CCPA aimed at recovering $\hat{sk}_1$, the first seven multiply instructions are solely associated with $\hat{sk}_1$ and are independent of $\hat{sk}_0$.

The model (Equation 3) presented in Section 2.2.2 characterizes the power consumption associated with an intermediate value at various time instances, differing from our proposed model. Our model aligns more closely with the one (Equation 1) described in Section 2.2.1. It focuses on power consumption related to each individual cycle, contingent upon inputs, outputs, and the computational processes of both preceding and current instructions. The model (Equation 1) simplify the outputs of the preceding instruction, designating them as an operand for the current instruction. In contrast, our model construction does not presuppose such simplifications; instead, it refines explanatory variables to better reflect the complex interdependencies inherent in instruction execution.

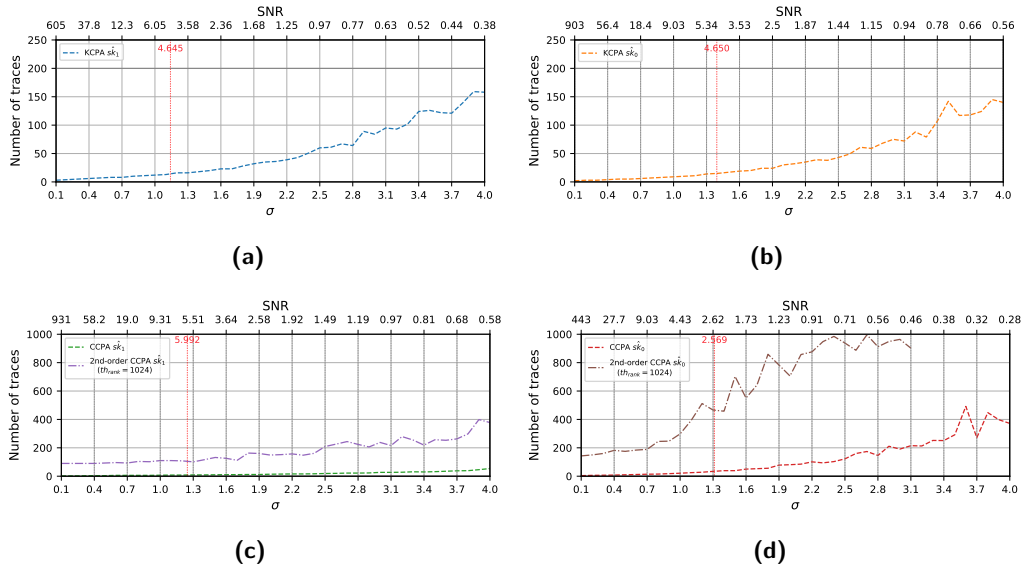


Figure 14: : Required number of traces under varying noise levels (σ) for KCPA and CCPA attacks. The rank threshold th_{rank} for the second-order attack is set to 1024. Red vertical dashed lines indicate the SNR values corresponding to real-world attacks. (a) KCPA for $\hat{sk}_1$. (b) KCPA for $\hat{sk}_0$. (c) The first-order and second-order CCPAs for $\hat{sk}_1$. (d) The first-order and second-order CCPAs for $\hat{sk}_0$.

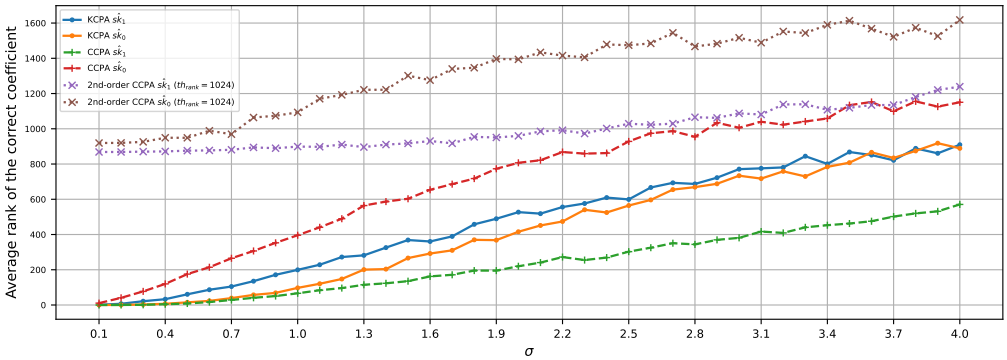


Figure 15: Average rank of the correct secret coefficients under varying noise levels (σ). The rank threshold th_{rank} for the second-order attack is set to 1024.

Noise The STM32 platform exhibits relatively low noise levels, contributing to the high success rate of our attacks. To evaluate the generalizability of our method in higher-noise settings, we conduct simulations to explore the attacks' performance under increasing noise. Specifically, we introduce additional Gaussian noise with a standard deviation σ, ranging from 0.1 to 4.0 in increments of 0.1.

We employ LR models with $n_p = 49,920$ for the simulations. For each σ, we collect target traces to achieve a 99.99% success rate in recovering secret coefficients. In the second-order CCPA, the rank threshold th_{rank} is set to 1024.

Figure 14 presents the number of traces required to recover $\hat{sk}_1$ and $\hat{sk}_0$ under varying noise levels. Due to variations in the modeling quality (R^2), the Signal-to-Noise Rate (SNR) for each instruction differs across attacks. Thus, Figure 14 is divided into four

subplots to show the average SNR for each attack. In first-order attacks, the number of required traces remains below 100 when $\sigma < 2.0$, demonstrating the robustness of these attacks. KCPA remains effective up to $\sigma = 3.2$. In contrast, the second-order CCPA requires approximately 150 traces even at low noise levels ($\sigma = 0.1, 0.2$) due to the masking countermeasures and stricter rank thresholds. Notably, for $\sigma > 3.1$, the second-order CCPA can recover $\hat{sk}_1$ but fails to recover $\hat{sk}_0$, primarily because of the limited number of instructions available for $\hat{sk}_0$.

Figure 15 shows the average rank of the correct secret coefficients with 1,000 target traces under varying noise levels. As noise increases, the average rank rises significantly across all attack methods, with the rate of increase depending on the number of instructions used. For first-order attacks, leveraging more instructions exhibit slower rank growth, with CCPA for $\hat{sk}_1$ (utilizing 7 instructions) achieving better performance than CCPA for $\hat{sk}_0$ (utilizing only 2 instructions). In second-order attacks, the high threshold ($th_{rank} = 1024$) leads to higher average ranks, even at a low noise level ($\sigma = 0.1$), with average ranks of 868 for $\hat{sk}_1$ and 919 for $\hat{sk}_0$. When the average rank approaches $q/2$, recovering secret coefficients becomes significantly more challenging, which explains the failure to recover $\hat{sk}_0$ for $\sigma > 3.1$.

Type of the MAC Circuit During model construction, we conduct a simple reverse engineering experiment to identify the MAC circuit type used in 16×16 multiply instructions, simulating the intermediate values generated during multiplication. The F-statistics for the explanatory variables related to the MAC circuit, shown in Table 3, validate the efficacy of our simulation.

We reverse-engineer the 16×16 multiplier to identify the MAC circuit type and simulate the values it computes. The F-statistics in Table 3 confirm our simulation's accuracy for MAC-related variables.

This work is performed on the STM32F303 device, manufactured by STMicroelectronics, confirming that this particular device utilizes a Modified Booth multiplier within its DSP extension. It is reasonable to assume that other Cortex-M4(F) devices from the same manufacturer are similarly configured. However, whether other vendors adopt comparable microarchitectural structures remains an open inquiry, warranting further exploration in our future works. [8]

Application to $\mathbf{NTT^{-1}}$ In addition to pair-pointwise multiplication, the NTT^{-1} operation during the Kyber decryption has drawn significant attention as a target for SCA attacks [PPM17, HHP+21, HSST23]. Both the speed-optimized and stack-optimized Kyber implementations use the same assembly code for NTT^{-1}.

The strategies for implementing the NTT^{-1} present challenges to the attacks. First, two NTT^{-1} coefficients are stored within a single 32-bit word, utilizing vectorized instructions such as `sadd16` and `ssub16` to perform two half-word additions and subtractions simultaneously. Furthermore, the *register allocation* strategy that minimizes load/store instructions to only occur every third NTT^{-1} layer, while a *lazy reduction* strategy ensures that modular reduction operations are not performed at each NTT^{-1} layer.

While addition and subtraction operations are performed on coefficient pairs which are stored in 32-bit words, a significant number of Montgomery reduction and Barrett reduction operations are applied to individual coefficients. These reductions utilize 16×16 or 32×16 multiply instructions. By leveraging our proposed model to characterize these multiply instructions, it may be possible to recover some individual NTT^{-1} coefficients. Integrating this approach with the BP algorithm could facilitate the recovery of all coefficients, which we plan to explore in the future work.

[8] Given that the DSP extension is an integral component of the Cortex-M4 core [ARMd], rather than a peripheral one, it is highly probable that a consistent microarchitectural structure is employed across devices.

Application to Plantard Implementation Huang et al. [HZZ+22] employ Plantard reduction as an alternative to Montgomery reduction to enhance algorithm efficiency, providing implementations for both speed-optimization and stack-optimization in the *pqm4* repository. By precomputing $\zeta' = \zeta q^{-1} \bmod^{\pm} 2^{32}$ and $qa = q \cdot 2^3$, the Plantard multiplication reduces the usage of one multiply instruction.

The assembly macro **doublebasemul_frombytes_asm** of the stack-optimized implementation with Plantard multiplication and reduction is illustrated in Figure 18. Within this macro, the number of multiply instructions used for the first-degree polynomial multiplication is nine, one less than the number targeted in this paper. This discrepancy does not impede the proposed profiled attacks. However, it may result in a slight decrease in the attack performance.

The Plantard multiplication employs a 32×16-bit signed multiply instruction **smulwt**, which is not included in our analysis. However, this instruction is decomposed into two 16×16-bit signed multiplications, fitting within our model. For a 32-bit a and a 16-bit b, the multiplication can be expressed as: $a\times b = ((a_H << 16)+a_L)\times b = (a_H \times b) << 16+(a_L \times b)$, where a_H, a_L are the top half and bottom half of a, respectively. Similarly, the Plantard reduction utilizes a 32×32-bit signed multiply instruction **mul** which is decomposed into two 16×32-bit signed multiplications. In our model, extending the multiplier width from 16-bit to 32-bit is straightforward, as the multiplier is not used to generate Booth codes. This modification only requires increasing the width of the partial product pp_i from 18 to 34 bits, along with adjusting the corresponding widths of s_i and c_i. Therefore, our leakage model remains applicable to stack-optimized Kyber implementation using Plantard reduction.

Masking Masking is a widely used countermeasure against side-channel power analysis and inherently affects the performance of profiled attacks. Our second-order attack successfully demonstrates the independent recovery of each secret coefficient from a first-order masked stack-optimized Kyber implementation, as shown in Section 5.4.2. While it is feasible to extend our attack to higher orders, this requires a greater number of traces and increased processing time.

It is important to note that our attack on Kyber is applicable only to schemes that mask the secret key but not the ciphertext, which is the common case [RRd+16, OSPG18, HKL+22]. If the ciphertext is masked during the Kyber decryption process, the proposed attacks would no longer be able to independently predict a single secret coefficient. Instead, it would be required to simultaneously predict a combination of coefficients, specifically two ciphertext coefficients and one secret coefficient. This significantly increases the difficulty and complexity of the attack, as the number of possible combinations grows exponentially with each additional coefficient that must be predicted.

Shuffling Within cryptographic algorithms, there are many small operations that, while functionally identical, process different data. The fundamental concept of shuffling is to randomize the order of operations in a way that is unpredictable to an attacker, thereby preventing the correlation between the physical observations and the secret data being processed.

For Kyber pair-pointwise multiplication, shuffling technique randomizes the execution order of 128 first-degree polynomial multiplications. This approach effectively nullifies the direct threat posed by our attacks. Although the order of operations is changed, the ciphertext coefficient pair and the corresponding secret coefficient pair are bound together in each first-degree polynomial multiplication. In other words, $\hat{ct}_{2i}$ and $\hat{ct}_{2i+1}$ are always multiplied with $\hat{sk}_{2i}$ and $\hat{sk}_{2i+1}$, rather than any other pair of secret coefficients.

If it becomes feasible to identify the positions of the ciphertext coefficient pairs across multiple measurements, our attacks can recover the corresponding secret coefficient pairs. As demonstrated in [ABB+24, Section 3.2], selecting a specific ciphertext polynomial with

distinct coefficient pairs for template generation allows for determining the positions of the ciphertext coefficient pairs during template matching, thereby pinpointing the associated secret coefficient pairs. Alternatively, leveraging Deep Learning-based analysis as proposed in [LZH+22] could facilitate the identification of ciphertext coefficient pairs, providing an additional avenue for locating the corresponding secret coefficient pairs.

7 Conclusion

In this paper, we target the pair-pointwise multiplication during the decryption process in the stack-optimized Kyber implementation which is running on the ARM Cortex-M4. We model the power consumption during the **EX** stage of the multiply instruction using linear regression and provide a systematic approach for selecting explanatory variables. We evaluate the coefficient of determination (R^2) of the models under different trace fragments (n_p). By combining models from multiple instructions, we perform profiled attacks to predict each secret coefficient individually.

We propose two types of profiled attacks: one based on known ciphertext and the other on chosen ciphertext. Both attacks can predict secret coefficients individually; however, KCPA requires that the recovery of even-indexed coefficients depends on the previously recovered odd-indexed coefficients, whereas CCPA imposes no such limitation. Although the single store operation in polynomial multiplication presents a challenge to previous attacks, the Hamming weight information leaked by this instruction can enhance the proposed attacks.

Using KCPA, we recover a pair of secret coefficients from an unprotected implementation with a success rate of 99.99%, using only 12 target traces. Using CCPA, we recover a secret coefficient pair with only 13 traces. The proposed CCPA can also be applied to a protected implementation. In a practical second-order attack, by setting an appropriate rank threshold, 201 traces are required to recover a secret coefficient pair from a first-order masked implementation.

Acknowledgements

The authors would like to thank the anonymous reviewers and the shepherd for their insightful suggestions and comments which improved this work. This work was supported by National Natural Science Foundation of China (Grant No. 62472397), Innovation Program for Quantum Science and Technology (Grant No. 2021ZD0302902), and Ningbo Young Science and Technology Talent Cultivation Program (Grant No. 2023QL007).

References

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 29–45. Springer, Berlin, Heidelberg, August 2003.

[ABB+24] Estuardo Alpirez Bock, Gustavo Banegas, Chris Brzuska, Lukasz Chmielewski, Kirthivaasan Puniamurthy, and Milan Sorf. Breaking DPA-protected kyber via the pair-pointwise multiplication. In Christina Pöpper and Lejla Batina, editors, *ACNS 24International Conference on Applied Cryptography and Network Security, Part II*, volume 14584 of *LNCS*, pages 101–130. Springer, Cham, March 2024.

[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, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 853–871. Springer, Cham, June 2022.

[ARMa] ARM. Arm cortex-M4 Processor Datasheet. https://developer.arm.com/documentation/102832/0100.

[ARMb] ARM. Arm cortex-M4 Processor Technical Reference Manual. https://developer.arm.com/documentation/100166/0001.

[ARMc] ARM. Armv7-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0403.

[ARMd] ARM. The DSP capabilities of Arm cortex-M4 and cortex-M7 Processors. https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, Berlin, Heidelberg, August 2004.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM J. Emerg. Technol. Comput. Syst.*, 17(2):10:1–10:26, 2021.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR TCHES*, 2021(4):173–214, 2021.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[Boo51] Andrew Donald Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4:236–240, 1951.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Berlin, Heidelberg, August 1999.

[CK13] Omar Choudary and Markus G. Kuhn. Efficient template attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2013.

[CK18] Marios O. Choudary and Markus G. Kuhn. Efficient, portable template attacks. *IEEE Transactions on Information Forensics and Security*, 13(2):490–501, 2018.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 13–28. Springer, Berlin, Heidelberg, August 2003.

[dCRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-lwe encryption. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 339–344. ACM, 2015.

[DNGW23] Elena Dubrova, Kalle Ngo, Joel Gärtner, and Ruize Wang. Breaking a fifth-order masked implementation of crystals-kyber by copy-paste. In Masayuki Fukumitsu and Shingo Hasegawa, editors, *Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop, APKC 2023, Melbourne, VIC, Australia, July 10-14, 2023*, pages 10–20. ACM, 2023.

[GLP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In Louis Goubin and Mitsuru Matsui, editors, *CHES 2006*, volume 4249 of *LNCS*, pages 15–29. Springer, Berlin, Heidelberg, October 2006.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR TCHES*, 2021(4):88–113, 2021.

[HKL+] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. mkm4: A First-order Masked Kyber Implementation on ARM Cortex-M4. https://github.com/masked-kyber-m4/mkm4.

[HKL+22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Report 2022/058, 2022.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[HSST23] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Katharina Thieme. Adapting belief propagation to counter shuffling of NTTs. *IACR TCHES*, 2023(1):60–88, 2023.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022.

[impa] m4fspeed: The speed version implementation of Kyber-768 on ARM Cortex-m4. https://github.com/mupq/pqm4/tree/3bfbbfd30401bd1dce3c497feb2a152713f2e735/crypto_kem/kyber768/m4fspeed.

[impb] m4fstack: The stack version implementation of Kyber-768 on ARM Cortex-m4. https://github.com/mupq/pqm4/tree/3bfbbfd30401bd1dce3c497feb2a152713f2e735/crypto_kem/kyber768/m4fstack.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Berlin, Heidelberg, August 1999.

[KJJR11] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, April 2011.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Berlin, Heidelberg, August 1996.

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

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[LZH+22] Yanbin Li, Jiajie Zhu, Yuxin Huang, Zhe Liu, and Ming Tang. Single-trace side-channel attacks on the toom-cook: The case study of Saber. *IACR TCHES*, 2022(4):285–310, 2022.

[MKK+23] Soundes Marzougui, Ievgen Kabin, Juliane Krämer, Thomas Aulbach, and Jean-Pierre Seifert. On the feasibility of single-trace attacks on the gaussian sampler using a CDT. In Elif Bilge Kavun and Michael Pehl, editors, *COSADE 2023*, volume 13979 of *LNCS*, pages 149–169. Springer, Cham, April 2023.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 199–216. USENIX Association, August 2017.

[MWK+24] Catinca Mujdei, Lennert Wouters, Angshuman Karmakar, Arthur Beckers, Jose Maria Bermudo Mera, and Ingrid Verbauwhede. Side-channel analysis of lattice-based post-quantum cryptography: Exploiting polynomial multiplication. *ACM Trans. Embed. Comput. Syst.*, 23(2):27:1–27:23, 2024.

[NIS24a] NIST. FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard, 2024. https://csrc.nist.gov/pubs/fips/203/final.

[NIS24b] NIST. Three Federal Information Processing Standards (FIPS) for Post-Quantum Cryptography, 2024. https://csrc.nist.gov/News/2024/postquantum-cryptography-fips-approved.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure masked Ring-LWE implementations. *IACR TCHES*, 2018(1):142–174, 2018.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *LATINCRYPT 2019*, volume 11774 of *LNCS*, pages 130–149. Springer, Cham, October 2019.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, Cham, September 2017.

[QCZ+21] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based NIST candidate KEMs. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 92–121. Springer, Cham, December 2021.

[RBRC22] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) nist pqc candidates for practical message recovery attacks. *IEEE Transactions on Information Forensics and Security*, 17:684–699, 2022.

[RO04] Christian Rechberger and Elisabeth Oswald. Practical template attacks. In Chae Hoon Lim and Moti Yung, editors, *WISA 04*, volume 3325 of *LNCS*, pages 440–456. Springer, Berlin, Heidelberg, August 2004.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT - A performance evaluation study over kyber and dilithium on the ARM cortex-m4. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 2020.

[RRd+16] Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking ring-LWE. *Journal of Cryptographic Engineering*, 6(2):139–153, June 2016.

[RRD+23] Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'Anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on LWE-based KEMs - parallel PC oracle attacks on Kyber KEM and beyond. *IACR TCHES*, 2023(2):418–446, 2023.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 683–702. Springer, Berlin, Heidelberg, September 2015.

[SAB+] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehle, and Jintai Ding. Algorithm Information of CRYSTALS-Kyber. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 30–46. Springer, Berlin, Heidelberg, August / September 2005.

[TS24] Tolun Tosun and Erkay Savas. Zero-value filtering for accelerating non-profiled side-channel attack on incomplete ntt-based implementations of lattice-based cryptography. *IEEE Trans. Inf. Forensics Secur.*, 19:3353–3365, 2024.

[Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, 1964.

[XPR+22] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of Kyber. *IEEE Transactions on Computers*, 71(9):2163–2176, 2022.

A Montgomery Reduction

The coefficients of the product polynomial need to be modulo q. Kyber represents elements in Montgomery representation in order to avoid expensive division by q in the computation mod q. It replaces the modulo q operation with the division by 2^{16} (taking the top half of a register) and the computation $\bmod^{\pm} 2^{16}$ (taking the bottom half of a register), as shown in Algorithm 5.

Algorithm 5: Signed Montgomery reduction in Kyber (simplified)

Input: Product $a \in \mathbb{Z}$, modulus $q = 3329$, $R = 2^{16} > q, q^{-1} \bmod R$
Output: $t \equiv aR^{-1} \bmod q$, $|t| \leq q$

1 $t := a(-q^{-1}) \bmod^{\pm} R$ // Taking the bottom half of the product
2 $t := (a + tq)/R$ // taking the top half of the sum
3 **return** t

B Supplementary Figures and Tables

```
1  ldr poly0, [bptr], #4
2  ldr poly2, [bptr], #4
3
4  smultt t, poly0, poly1
5  smulbt t2, t, qinv       // mont
6  smlabb t2, q, t2, t      // mont
7  smultb t2, t2, zeta
8  smlabb t2, poly0, poly1, t2
9  str t2, [rptr_tmp], #4
10 smuadx t, poly0, poly1
11 str t, [rptr_tmp], #4
```

```
12 neg zeta, zeta
13
14
15 smultt t, poly2, poly3
16 smulbt t2, t, qinv       // mont
17 smlabb t2, q, t2, t      // mont
18 smultb t2, t2, zeta
19 smlabb t2, poly2, poly3, t2
20 str t2, [rptr_tmp], #4
21 smuadx t2, poly2, poly3
22 str t2, [rptr_tmp], #4
```

(a)

```
1  ldr.w poly2, [bptr, #4]
2  ldr poly0, [bptr], #8
3
4  smultt t, poly0, poly1
5  smulbt t2, t, qinv       // mont
6  smlabb t2, q, t2, t      // mont
7  smultb t2, t2, zeta
8  smlabb t2, poly0, poly1, t2
9  smulbt t, t2, qinv       // mont
10 smlabb t, q, t, t2       // mont
11 smuadx t2, poly0, poly1
12 smulbt poly0, t2, qinv   // mont
13 smlabb poly0,q,poly0,t2  // mont
14 pkhtb t, poly0, t, asr#16
15 str t, [rptr], #4
```

```
16 neg zeta, zeta
17
18
19 smultt t, poly2, poly3
20 smulbt t2, t, qinv       // mont
21 smlabb t2, q, t2, t      // mont
22 smultb t2, t2, zeta
23 smlabb t2, poly2, poly3, t2
24 smulbt t, t2, qinv       // mont
25 smlabb t, q, t, t2       // mont
26 smuadx t2, poly2, poly3
27 smulbt poly0, t2, qinv   // mont
28 smlabb poly0,q,poly0,t2  // mont
29 pkhtb t, poly0, t, asr#16
30 str t, [rptr], #4
```

(b)

Figure 16: Two types of implementations for the pair-pointwise multiplication in *pqm4* repository. (a) The speed-optimized implementation of the macro **`doublebasemul_frombytes_asm_16_32`**. The *lazy reduction* strategy is applied to minimize the number of reductions, with each first-degree polynomial multiplication reducing two modular reductions that act on the result coefficients. Consequently, each result coefficient is individually stored into the memory. (b) The stack-optimized implementation of the macro **`doublebasemul_frombytes_asm`**. In a single first-degree polynomial multiplication, two result coefficients are packed into a 32-bit register using a **`pkhtb`** instruction and then stored into the memory using only one **`str`** instruction.

```
ldrd r4, r5, [r0], #8 // Multilicand and multiplier
NOP6 // Macro that generate 6 nop.w instructions to flush the pipeline
bl <trigger_high>
NOP6
smul<x><y> r3, r4, r5 // Target instruction
NOP6
bl <trigger_low>
```

(a)

```
ldrd r4, r5, [r0], #8 // Multilicand and multiplier
ldr r6, [r0], #4 // Accumulate number
NOP6 // Macro that generate 6 nop.w instructions to flush the pipeline
bl <trigger_high>
NOP6
smla<x><y> r3, r4, r5, r6 // Target instruction
NOP6
bl <trigger_low>
```

(b)

```
ldrd r4, r5, [r0], #8 // Multilicand and multiplier
NOP6 // Macro that generate 6 nop.w instructions to flush the pipeline
bl <trigger_high>
NOP6
smuad{x} r3, r4, r5 // Target instruction
NOP6
bl <trigger_low>
```

(c)

Figure 17: Target ARM assembly code. The 32-bit registers $r4$, $r5$, and $r6$ are designated for storing the multiplicand, multiplier, and the accumulate number, respectively. (a) Signed 16-bit multiply instructions. If `<x>` is b, the bottom half (bits [15:0]) of $r4$ is used as the multiplicand; otherwise, if `<x>` is t, the top half (bits [31:16]) of $r4$ is used. If `<y>` is b, the bottom half of $r5$ is used as the multiplier; otherwise, the top half of $r5$ is used. (b) Signed 16-bit multiply-accumulate instructions. The usage of parameters `<x>` and `<y>` is consistent with that in (a). (c) Signed dual 16-bit multiply instructions. If `x` is present, the multiplications are bottom $\times$ top and top $\times$ bottom. If the `x` is omitted, the multiplications are bottom $\times$ bottom and top $\times$ top.

```
ldr.w poly0, [bptr], #4

smulwt t, zeta, poly1
smlabt t, t, q, qa
smultt t, poly0, t
smlabb t, poly0, poly1, t
mul t, t, qinv       // plantard
smlatt t, t, q, qa   // plantard
smuadx t2, poly0, poly1
mul t2, t2, qinv     // plantard
smlatt t2, t2, q, qa // plantard
pkhtb t, t2, t, asr#16
str t, [rptr], #4
```

```
neg zeta, zeta
ldr.w poly0, [bptr], #4

smulwt t, zeta, poly3
smlabt t, t, q, qa
smultt t, poly0, t
smlabb t, poly0, poly3, t
mul t, t, qinv       // plantard
smlatt t, t, q, qa   // plantard
smuadx t2, poly0, poly3
mul t2, t2, qinv     // plantard
smlatt t2, t2, q, qa // plantard
pkhtb t, t2, t, asr#16
str t, [rptr], #4
```

Figure 18: The macro **doublebasemul_frombytes_asm** of the stack-optimized implementation with Plantard reduction in *pqm4* repository. In comparison with the stack-optimized implementation utilizing Montgomery reduction, this implementation employs one fewer modular reduction operation in a single first-degree polynomial multiplication.

Table 9: F-test of explanatory variables for each multiply instruction with 49,920 trace fragments for profiling ($n_c = 780$ for the unprotected implementation, $n_c = 390$ for the first-order masked implementation). The numbers in brackets of the first column indicate the width of the variable. The numbers in brackets of data columns indicate the width of the variable after selection. Tests which fail to reject at the 5% level are shaded grey. df1's are shown in brackets of data columns.

	Instructions	`smultt`	`smulbt`	`smlabb`	`smultb`	`smlabb`
	Serial # (t)	0	1	2	3	4
	R^2	0.6270	0.7526	0.8592	0.7338	0.9260
	Adjusted R^2	0.6234	0.7484	0.8570	0.7300	0.9247
	df2	49438	49090	49156	49212	49065
F-statistics	$\mathbf{O}_{0,(t)}$ (32)	1165.94 (24)	213.1 (24)	(0)	65.09 (12)	50.71 (24)
	$\mathbf{O}_{1,(t)}$ (32)	44.31 (24)	(0)	22.55 (27)	12.94 (12)	14.57 (24)
	$\mathbf{O}_{2,(t)}$ (32)	(0)	(0)	15.03 (23)	(0)	9.33 (24)
	$\mathbf{R}_{(t)}$ (32)	32.14 (24)	(0)	(0)	(0)	(0)
	$\mathbf{DMb}_{(t)}$ (580)	(0)	6.66 (365)	23.94 (253)	(0)	10.93 (396)
	$\mathbf{DMt}_{(t)}$ (580)	30.23 (405)	(0)	(0)	9.73 (332)	(0)
	$\mathbf{TO}_{0,(t)}$ (32)	(0)	53.47 (24)	2.97 (11)	12.44 (10)	61.02 (12)
	$\mathbf{TO}_{1,(t)}$ (32)	(0)	124.51 (24)	0.51 (10)	47.54 (27)	96.64 (12)
	$\mathbf{TO}_{2,(t)}$ (32)	(0)	(0)	(0)	32.57 (24)	(0)
	$\mathbf{TR}_{(t)}$ (32)	154.30 (4)	3.15 (12)	16.15 (16)	129.79 (23)	31.53 (11)
	$\mathbf{TDMb}_{(t)}$ (580)	(0)	(0)	79.92 (423)	8.95 (267)	(0)
	$\mathbf{TDMt}_{(t)}$ (580)	(0)	8.90 (380)	(0)	(0)	4.59 (351)

	Instructions	`smulbt`	`smlabb`	`smuadx`	`smulbt`	`smlabb`
	Serial # (t)	5	6	7	8	9
	R^2	0.8654	0.7086	0.7326	0.7718	0.6847
	Adjusted R^2	0.8629	0.7041	0.7256	0.7658	0.6798
	df2	48995	49145	48635	48640	49147
F-statistics	$\mathbf{O}_{0,(t)}$ (32)	161.29 (25)	(0)	31.55 (24)	116.53 (25)	(0)
	$\mathbf{O}_{1,(t)}$ (32)	(0)	5.74 (28)	3.64 (24)	(0)	7.02 (28)
	$\mathbf{O}_{2,(t)}$ (32)	(0)	-0.0 (23)	(0)	(0)	0.54 (23)
	$\mathbf{R}_{(t)}$ (32)	(0)	(0)	35.59 (12)	(0)	(0)
	$\mathbf{DMb}_{(t)}$ (580)	6.29 (365)	21.51 (254)	3.20 (381)	1.93 (365)	23.08 (252)
	$\mathbf{DMt}_{(t)}$ (580)	(0)	(0)	12.85 (376)	(0)	(0)
	$\mathbf{TO}_{0,(t)}$ (32)	44.19 (24)	54.24 (12)	21.42 (14)	60.53 (24)	-0.0 (12)
	$\mathbf{TO}_{1,(t)}$ (32)	50.61 (24)	1.95 (10)	306.60 (29)	122.05 (24)	0.26 (10)
	$\mathbf{TO}_{2,(t)}$ (32)	15.29 (20)	(0)	164.04 (24)	(0)	(0)
	$\mathbf{TR}_{(t)}$ (32)	7.82 (25)	3.70 (24)	181.00 (23)	123.91 (26)	7.14 (24)
	$\mathbf{TDMb}_{(t)}$ (580)	13.80 (441)	66.09 (423)	5.69 (377)	1.72 (424)	77.01 (423)
	$\mathbf{TDMt}_{(t)}$ (580)	(0)	(0)	(0)	40.92 (391)	(0)

Table 10: The performance of the first-order attacks enhanced with Cond**1**/Cond**2** across different n_{hw} values. The best results for a specific n_{hw} of each attack are shaded grey.

Cond**2**-augmented KCPA

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	100	6,400	−|14	−|14	−|12	−|11	−|10	−|8	−|8	−|7
	200	12,800	−|14	−|14	−|12	−|11	−|10	−|8	−|8	−|7
	500	32,000	−|14	−|13	−|12	−|11	−|9	−|8	−|8	−|6
	780	49,920	−|14	−|13	−|12	−|11	−|9	−|8	−|8	−|6

Cond**1**-augmented CCPA

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	100	6,400	7|28	7|29	7|29	7|30	7|31	7|31	7|31	7|32
	200	12,800	7|25	7|26	7|26	7|26	7|27	7|27	8|27	8|28
	500	32,000	7|24	7|24	7|24	7|25	7|25	7|26	7|26	7|26
	780	49,920	7|23	7|23	7|24	7|24	7|24	7|25	7|25	7|25

Cond**2**-augmented CCPA

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	100	6,400	−|17	−|18	−|16	−|16	−|15	−|14	−|15	−|16
	200	12,800	−|17	−|17	−|16	−|15	−|14	−|14	−|14	−|15
	500	32,000	−|17	−|17	−|16	−|14	−|14	−|13	−|13	−|14
	780	49,920	−|17	−|17	−|16	−|14	−|14	−|13	−|13	−|14

Table 11: The performance of the second-order CCPA attacks enhanced with Cond**1** across different n_{hw} values. The best results for a specific n_{hw} of each attack are shaded grey.

$th_{rank} = 512$

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	50	6,400	73|245	72|252	71|260	71|292	70|281	70|302	70|306	70|311
	100	12,800	71|214	70|214	70|219	70|225	69|230	68|244	69|249	67|255
	250	32,000	70|214	69|214	68|219	68|222	67|230	67|242	67|248	67|253
	390	49,920	70|201	68|203	68|212	68|217	67|223	67|238	67|244	67|247

$th_{rank} = 768$

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	50	6,400	103|242	97|249	94|250	93|256	92|263	88|282	88|287	88|292
	100	12,800	99|207	94|213	93|217	92|223	88|230	87|244	87|249	86|254
	250	32,000	97|208	93|214	92|214	92|221	88|228	86|242	86|251	84|257
	390	49,920	97|202	93|210	89|214	91|221	86|223	86|240	86|248	86|253

$th_{rank} = 1024$

Performance	n_c	n_p	n_{hw}							
			1	2	3	4	5	6	7	8
n_a for $\hat{sk}_1 \| \hat{sk}_0$	50	6,400	131|264	126|282	123|282	121|292	116|304	112|319	108|326	108|324
	100	12,800	128|264	125|241	121|242	119|249	114|261	111|274	108|284	107|284
	250	32,000	126|227	122|238	118|242	115|249	112|259	108|272	107|280	106|284
	390	49,920	126|222	123|235	118|242	115|249	112|253	108|269	107|272	106|278