SCA-MAYO: Demystifying Side-channel Security of MAYO Signature Scheme

N.K. Vishwaajith, *IIT Bombay*
Anindya Ganguly, *IIT Kanpur*
Trevor Yap, *NTU Singapore*
Debranjan Pal, *LNMIIT Jaipur*
Puja Mondal, *IIT Kanpur*
Suparna Kundu, *COSIC, KU Leuven*
Sayandeep Saha, *IIT Bombay*
Shivam Bhasin, *NTU Singapore*
Ingrid Verbauwhede, *COSIC, KU Leuven*
Angshuman Karmakar, *IIT Kanpur*

Abstract

Multivariate quadratic (MQ)-based digital signature schemes are promising candidates for the ongoing post-quantum (PQ) standardization procedure due to their compact signatures and fast verification. However, the side-channel security of their recent optimized and protected implementations remains underexplored.

This work presents novel side-channel attacks on the MQ-based signature scheme MAYO, targeting both unprotected and masked implementations. We develop the first non-profiled single-trace correlation power attack against the recent nibble-sliced implementation of MAYO. Next, we evaluate two masking approaches proposed for MQ-based signatures. Our findings indicate that masking sensitive variables alone leaves the implementation vulnerable to single-trace attacks, and that masking only public variables also fails to provide the intended protection. To assess the second case, we consider both profiled and blind adversaries. The blind setting adopts a recent deep-learning-based blind side-channel analysis framework and demonstrates secret-key recovery without access to the masked public values or a cloned device.

We validate our attacks on an STM32F3 Cortex-M4 platform and demonstrate that recently proposed countermeasures do not provide adequate protection against realistic single-trace adversaries. Finally, we discuss how the underlying attack methodology extends to other MQ-based signature schemes and present a strengthened masking and refreshing countermeasure that mitigates the proposed attacks.

1 Introduction

MAYO and MQ-DSAs. Multivariate quadratic digital signature algorithms (MQ-DSAs) are among the major families of post-quantum (PQ) signature schemes. Unlike lattice- and hash-based signatures, such as Dilithium [14, 28], Falcon [15], and SPHINCS+ [29], MQ-DSAs offer significantly shorter signatures, making them attractive for resource-constrained devices. Four MQ-DSAs, UOV [9,22], MAYO [7], SNOVA [37], and QR-UOV [17], are currently in the third round of the NIST's additional PQ digital signature standardization process [1]. These schemes follow the Hash-and-Sign paradigm, where signing relies on an Oil-Vinegar (OV) trapdoor [22] that enables efficient solution of a system of multivariate quadratic equations. Since these designs share a common algebraic foundation, many implementation techniques and cryptanalytic insights naturally transfer across the family. In this work, we focus on MAYO, a representative MQ-DSA that combines compact signatures with a relatively small public key, making it an attractive candidate for deployment on embedded platforms.

Profiled and Non-Profiled SCA. Side-channel analysis (SCA) exploits physical leakages, such as power consumption [24] and electromagnetic emanations [18], to recover secret information from cryptographic implementations [24, 25]. SCA attacks are broadly classified into non-profiled and profiled attacks. Non-profiled attacks rely on generic leakage models, such as the Hamming Weight (HW) or Hamming Distance (HD) model. In contrast, profiled attacks first learn device-specific leakage, typically by using a cloned device. Conventional SCA combines multiple traces of the same intermediate computation, a phenomenon commonly referred to as vertical leakage. In contrast, horizontal SCA exploits multiple leakages involving the secret value within a single execution to recover the secret. If only side-channel traces are available and the corresponding inputs or outputs of the target operations are unknown, the attack becomes a blind SCA problem [12, 27]. Recently, Rezaeezade *et al.* proposed a framework for deep-learning-based blind SCA [34]. In this work, we build upon their framework to attack masked implementations of MAYO.

Related works. Side-channel attacks on MQ-DSAs have garnered significant attention in recent years. Aulbach *et al.* provide a comprehensive survey on this topic [2], which is also summarized in Table 1. Early works focused on multi-trace CPA attacks against Rainbow [30, 32], which were later

extended to UOV [30]. Aulbach *et al.* [3] subsequently proposed the first *profiled single-trace* attack on UOV, demonstrating that recovering intermediate values is sufficient for secret-key recovery. Jendral *et al.* [21] later applied a similar idea to an earlier bitsliced implementation of MAYO [6] using deep learning. However, their attack does not extend to the current nibble-sliced implementation [5, 7, 8]. Consequently, the side-channel security of the state-of-the-art MAYO implementation remains largely unexplored, which is the primary motivation for our work.

Masked implementations of MQ-DSAs. Masking is a widely adopted countermeasure against side-channel attacks [20], where computations are performed on randomized shares to decorrelate the processed values from the underlying secret. Existing masking techniques for MQ-based digital signature schemes can be broadly classified into two categories: masking sensitive variables and masking public variables. Kundu *et al.* proposed a generic masking methodology for UOV and related MQ-DSAs that achieves arbitrary-order security against classical differential power attacks (DPA) by dividing sensitive variables into multiple shares [26]. The authors further note that this masking might not be sufficient to counter simple power attacks (SPA) and that additional operations, such as shuffling and mask refreshing, are required. In contrast, Aulbach *et al.* [2] proposed an alternative approach that masks the public matrices involved in secret-dependent computations, with the primary objective of preventing single-trace attacks. Although previous work discussed the construction of this countermeasure from a design perspective, its practical resistance to SPA or horizontal single-trace attacks has not been systematically evaluated. In our work, we show that masking only public values is insufficient to resist SPA on the protected implementation.

Table 1: Comparison of existing side-channel attacks on MQ-DSAs and the attacks proposed in this work.

Works	**Target/Setting (unprotected)**	**Attack Type**	**Traces**
[30]	Rainbow	CPA (non-profiled)	30
[32]	Rainbow	CPA (non-profiled)	40-475
[30]	UOV	CPA (non-profiled)	Multi
[3]	UOV	Template attack (profiled)	Single
[21]	MAYO	DL-assisted (profiled)	Single
Attacks	**MAYO impl.**	**Attack type/Adversary**	**Traces**
Sec. §3	unprotected	CPA (non-profiled, $\mathcal{A}_1$)	Single
Sec. §4.1	masking	CPA (non-profiled, $\mathcal{A}_1$)	Single
Sec. §4.2	public-value masking	DL-assisted (profiled, $\mathcal{A}_2$)	Single
Sec. §4.3	public-value masking	DL-assisted (non-profiled, $\mathcal{A}_3$)	Single

Our Contributions. We evaluate both unprotected and protected MAYO implementations. The multiply-accumulate operation is one of the major operations in MAYO, where interactions between public and private variables occur, and such an operation occurs in multiple places of the signing algorithm [5,7]. This operation has undergone a significant change in version 2 of MAYO [5] (Nibbling MAYO [8]) compared to the version 1 implementation, rendering the previous attacks infeasible for this case. So, we focus mainly on the multiply-accumulate subroutine `m_vec_mul_add`, as an attack point for both unprotected and protected implementations. In particular, we target this routine when the secret key matrix $\mathbf{O}$ is being multiplied with the public key matrices (`P1P1t_times_O`). Since the secret key and the public key are associated, the attack becomes a single trace attack with only one public input. In the single trace paradigm, we define three adversaries: a) A non-profiled adversary $\mathcal{A}_1$; b) A blind adversary $\mathcal{A}_2$ with access to a clone device; c) A blind adversary $\mathcal{A}_3$ without any clone device. The blind setting arises as an artifact of the masking scheme proposed in [2]. More precisely, the masking scheme masks the public values, making them inaccessible to the adversary, necessitating blind attacks.

- **Non-profiled single-trace CPA:** We present a non-profiled, single-trace CPA against unprotected MAYO under the adversarial model $\mathcal{A}_1$ to recover the full oil matrix. To the best of our knowledge, this is the first successful single-trace attack on nibbling MAYO implementation. The attack generalises to protected implementations when the secret key is split into additive shares and multiplied by the public matrices. Since the attacker has access to the public variables, the same CPA approach can be applied to recover each individual share.
- **Develop novel distinguisher:** However, the above techniques fail when the public values are additively masked. Since the attacker does not know these random shares, they cannot accurately predict the intermediate leakage. To overcome this limitation, we exploit a structural feature of the `m_vec_mul_add` routine arising solely due to nibble slicing. Our approach constructs templates during an offline phase, which are then used in an online phase to recover the secret key, without the knowledge of the masks in the online phase.
- **Deep learning method on public value masking implementation:** For the $\mathcal{A}_2$ adversary model, we develop an attack based on multi-task DNN and recover the key with a single trace segment from the target device. The offline (profiling) phase assumes the availability of a clone device and can do Hamming weight labeling of the profiling traces. We emphasize that the attack phase required no knowledge of the masked public information, owing to the property of nibble-slicing we exploit. In the $\mathcal{A}_3$ model, we adapted the DL-BSCA framework [34] for a horizontal single trace attack in conjunction with our proposed distin-

guisher. In this case, there is no clone device, and a part of the collected traces is labeled using clustering and used for building templates. We propose a new labeling strategy, called *MultiPoint-Slicing*, which avoids high computational costs while incorporating multiple sample points. We show that the attack reduces the guessing entropy to less than 2 for each secret key component. This significantly reduces the attack complexity over brute-force, and establishes that even under the weakest attacker assumptions, the proposed masking schemes are insufficient.

- **Countermeasure:** We integrate a combination of masking and refreshing techniques as the countermeasure in the implementation of the target operation. This approach increases implementation costs by approximately 59× to provide security against first-order SCA attacks. Additionally, we show that applying this masking and refreshing technique effectively mitigates our attack, as the per-coefficient Guessing Entropy of the target operation aligns closely with that of random guessing.

2 Attack Surfaces and Threat Models

Notations. We use bold uppercase letters (for example, $\mathbf{P}, \mathbf{O}$) for matrices and bold lowercase letters (for example, $\mathbf{x}, \mathbf{y}, \mathbf{o}$) for vectors. An $m \times n$ matrix is denoted by $\mathbf{A} = [a_{i,j}]_{m\times n}$, where $a_{i,j}$ is the entry in row i, column j. We use $q, n, m, v, o \in \mathbb{N}$ to denote the field size, number of variables, number of equations, number of vinegar variables, and the oil-space dimension, respectively. The vector $\mathbf{0}_m \in \mathbb{F}_q^m$ denotes the all-zero vector. We write $\mathsf{Upper}(\mathbf{A})$ for the upper triangular part of $\mathbf{A}$ with non-zero diagonal entries. We use $a \in_U S$ for uniform sampling from the set S. For a multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$, we define its *differential polar form* (denoted by $\mathcal{DP}$) as the symmetric bilinear map: $\mathcal{DP}(\mathbf{x}, \mathbf{y}) = \mathcal{P}(\mathbf{x}+\mathbf{y}) + \mathcal{P}(\mathbf{x}) + \mathcal{P}(\mathbf{y})$.

We recall standard background on leakage models, CPA, DL-assisted blind SCA, and masking in Appendix E.

2.1 MQ-DSAs and MAYO

The Oil–Vinegar (OV) MQ-map forms the core of MQ-DSAs following the Hash-and-Sign paradigm [22]. Consequently, schemes such as UOV [9, 22], MAYO [7], QR-UOV [17], and SNOVA [37] share a closely related underlying algebraic structure, all using the OV map as a trapdoor.

The key generation algorithm first samples a secret oil subspace O of dimension o by expanding a secret seed $\mathtt{seed}_{sk}$ into a matrix $\mathbf{O} = [O_{r,c}]_{v\times o}$ with linearly independent columns, where $v = n - o$. The entries $O_{r,c}$ are referred to as *oil coefficients*. It then constructs the public multivariate quadratic (OV) map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ such that $\mathcal{P}(\mathbf{O}) = \mathbf{0}$. The map consists of m homogeneous quadratic polynomials, each represented by a matrix $\mathbf{P}_i \in \mathbb{F}_q^{n\times n}$. Each $\mathbf{P}_i$ is partitioned into three sub-matrices, where $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v\times v}$ (upper triangular) and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v\times o}$ are derived from the public seed $\mathtt{seed}_{pk}$. The remaining upper triangular matrix $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ is calculated as $\mathsf{Upper}\left(-\mathbf{O}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^\mathsf{T}\mathbf{P}_i^{(2)}\right)$, which ensures $\mathcal{P}(\mathbf{O}) = \mathbf{0}$.

During the signature generation, the signer needs to substitute the vinegar vector $\mathbf{v} \in_U \mathbb{F}_q^n$ into the public map $\mathcal{P}$. This partial evaluation separates the quadratic system into two parts: (i) that depends only on $\mathbf{v}$ and (ii) that remains linear in the oil variables. The resulting linear system is captured by a matrix $\mathbf{L}_i$, which encodes the coefficients of the oil variables after substitution. The linear matrix, $\mathbf{L}_i$ is computed as

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}) \cdot \mathbf{O} + \mathbf{P}_i^{(2)}. \tag{1}$$

This matrix $\mathbf{L}_i$ can either be precomputed during key generation or computed during the signature generation phase.

In MAYO [7], the oil subspace has a small dimension. The signature generation employs a *whipping map* that expands the oil-space dimension to ensure $m \approx k \cdot o$, where k is the whipping parameter. Suppose $\mathcal{P}$ be an (n, m)-OV map that vanishes on an o-dimensional secret subspace O. The MAYO construction extends this using a *whipping map*, denoted by $\mathcal{P}^* : \mathbb{F}_q^{nk} \to \mathbb{F}_q^m$, defined as

$$\mathcal{P}^*(\mathbf{s}_1, \ldots, \mathbf{s}_k) = \sum_{i=1}^{k} \mathbf{E}_{i,i}\,\mathcal{P}(\mathbf{s}_i) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{i,j}\,\mathcal{DP}(\mathbf{s}_i, \mathbf{s}_j).$$

Here, $\mathbf{E}_{i,j} \in \mathbb{F}_q^{m\times m}$ are public parameters. During signature generation, the signer first samples $(\mathbf{v}_1, \ldots, \mathbf{v}_k) \in_U \mathbb{F}_q^{nk}$ uniformly at random, and then computes $(\mathbf{o}_1, \ldots, \mathbf{o}_k) \in \mathsf{O}^k$ such that $\mathcal{P}^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t}$. Then it outputs $(\mathbf{s}_1, \ldots, \mathbf{s}_k) = (\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k)$ as the signature. During verification, the verifier reconstructs the whipping map $\mathcal{P}^*$ from $\mathcal{P}$ and evaluates it on the signature vectors. For MAYO-SL1 (128-bit) parameter set [7], $n = 86, m = 78, o = 8, k = 10$, and $q = 16$ (see Table 6, Appendix D).

Sensitivity Analysis of MAYO. To analyze the side-channel security of MAYO implementations, we identify the secret-dependent computations in the signing algorithm. During signing, the sensitive values are the vinegar vectors $\mathbf{v}_i$ and the oil vectors $\mathbf{o}_i$. Here, the vectors $\mathbf{v}_i$ are randomly sampled for each signing instance and can be treated as *ephemeral* secrets, whereas $\mathbf{O}$ constitutes the *static* or *long-term* secret key. If an attacker recovers at least one such vector, they can construct an equivalent oil subspace and forge the signature scheme in MQ-DSAs [3, 23, 33]. Hence, recovering a vinegar or oil vector is a primary attack objective.

In MAYO [5, 7], the secret key $\mathtt{seed}_{sk}$ is expanded to the oil matrix $\mathbf{O}$ during signing. A first sensitive computation is the derivation of $\mathbf{L}_i$ (see Equation 1), which involves multiplying the secret matrix $\mathbf{O}$ with public matrices $\mathbf{P}_i^{(1)}$. This step is implemented by the `P1P1t_times_O` subroutine [5], and leakage

from $\mathbf{L}_i$ directly reveals information about $\mathbf{O}$. Next, the multiplication of $\mathbf{P}_i^{(1)}$ with the vinegar vectors $\mathbf{v}_i$, implemented via `P1_times_Vt`, is another point of public-private interaction and qualifies as a leakage point. Finally, the computation $\mathbf{s}_i \leftarrow \mathbf{v}_i + \mathbf{O} \cdot \mathbf{x}_i$, implemented by `mat_mul`, involves the secret matrix $\mathbf{O}$ and public $\mathbf{x}_i$ and qualifies as an attack point. All routines refer to the nibbling MAYO implementation [5]. In the implementation we target, the subroutines `P1P1t_times_O`, `P1_times_Vt`, and `compute_M_and_VPV` internally use the multiply-accumulate subroutine `m_vec_mul_add`. In contrast, `mat_mul` uses `mul_f`. The `mul_f` multiplication subroutine is also used in earlier bit-sliced implementations [6].

Our work focuses on the target function `m_vec_mul_add`, which forms the core of matrix-matrix multiplications underlying multiple secret-dependent computations. Then we analyse how its leakage can be exploited to recover the secret.

2.2 Threat Models

To formalize our analysis, we define threat models based on three distinct adversaries, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, and three implementation settings: *unprotected*, *secret-key masking*, and *public-valued masking*. Across all scenarios, we assume that MAYO is executed on a device where the adversary has physical access, such as an embedded device. Table 1 summarizes the threat models considered in this paper, highlighting adversary capabilities and the corresponding implementations.

Under $\mathcal{A}_1$, the attacker operates in the typical *non-profiling framework*. In this setting, the attacker has access to the public variables used by the target operation. Within this framework, we develop a CPA attack, as shown in Section 3, against the unprotected implementation. This approach naturally extends to additive-key masking (see Section 4.1).

However, the above techniques fail when the public values are randomized into multiple shares, as proposed by Aulbach et al. [2]. This is because the public variables available to the adversary do not coincide with the inputs of the target operation, which instead consist of masked shares of the original public value that are typically inaccessible to the adversary. This means the adversary must operate in a *blind setting*. As a result, new attack techniques are needed. We investigate two adversary models: $\mathcal{A}_2$ (profile blind, see Section 4.2) and $\mathcal{A}_3$ (non-profiled blind, see Section 4.3).

In the $\mathcal{A}_2$ setting, we assume that the attacker has access to a clone device. Thus, the adversary knows the masked values of the public variables during the profiling phase but does not have access to them on the target (or attack) device. This aligns with the standard profiled attack threat model. In contrast, in a more difficult $\mathcal{A}_3$ setting, the attacker does not have access to a clone device or any information about the mask values of the public variables, and only has access to side-channel traces. This setting represents the case when mask are generated through internal random number generators but never revealed to the user. As the final result is unmasked before output, a standard user does not need knowledge of this mask. Consequently, the attacker operates in a fully blind setting, rendering $\mathcal{A}_3$ the most restricted adversary and its attack the most difficult to execute.

3 Non-profiled Single Trace SCA on MAYO

In Section 2, we identified several potential SCA targets in MAYO . We focus on the matrix–matrix multiplication routine `P1P1t_times_O`, which computes $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}) \cdot \mathbf{O} + \mathbf{P}_i^{(2)}$ and involves interactions between public matrices and the secret oil matrix. This computation relies on repeated multiply-accumulate operations to multiply oil coefficients with the public matrix coefficients, thereby serving as a primary source of leakage. The initial MAYO implementation [6] follows a bit-sliced design, where the `bitsliced_m_vec_mul_add` subroutine is used to perform the multiply-accumulate operation. Jendral and Dubrova [21] demonstrated a single-trace SCA on this implementation using a profiled, DL-based approach. In contrast, the recent nibbling MAYO implementation [5, 8] replaces `bitsliced_m_vec_mul_add` with `m_vec_mul_add`. This modification changes the leakage characteristics and opens a new avenue for SCA. Specifically, the attack proposed in [21] does not naturally scale for the nibble-sliced implementation, which we shall elaborate in Sec. 3.1.

3.1 Nibble-sliced Multiply-Accumulate.

In the nibble-sliced implementation, the multiply-accumulate subroutine `m_vec_mul_add` requires three inputs: 4-bit secret coefficient **a** (**a** stores an oil coefficient from **O** matrix), a 64-bit vector **in** containing coefficients from the matrices $\mathbf{P}_i^{(1)}$, and **acc**, which is initialized with coefficients from $\mathbf{P}_i^{(2)}$, and gradually accumulates each entry of the product matrix. In a single invocation, `m_vec_mul_add` processes 16 elements over the field $\mathbb{F}_{16}$ in parallel by packing them into the public vector **in**. Each field element is multiplied by the same coefficient **a**, and the resulting products are XOR-accumulated into the corresponding positions of **acc**. To enable efficient computation, the routine first expands **a** into a precomputed word $\mathbf{tab} = (8\mathbf{a} \,\|\, 4\mathbf{a} \,\|\, 2\mathbf{a} \,\|\, \mathbf{a})$ through `mul_table(a)`. The packed representation allows the routine to operate on all 16 field elements simultaneously within each machine word. Specifically, it extracts the 0th, 1st, 2nd, and 3rd bits of every nibble inside **in** parallel, multiplies each extracted bit-plane with the corresponding portion of **tab**, and XORs the resulting contributions. This recombination step reconstructs the products $\mathbf{in}[j] \cdot \mathbf{a}$ for all packed elements in parallel, after which the accumulator is updated accordingly.

Within `P1P1t_times_O`, the function `m_vec_mul_add` processes coefficients from the public matrices $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)_{i=1}^{m}$ in a batched manner via the input vector **in** (see Listing 1). Each invocation multiplies a fixed oil

coefficient **a** with the corresponding entries across 16 public matrices. This process is repeated $\texttt{m_vec_limbs} = \lceil m/16 \rceil$ times. For an oil coefficient $O_{r,c}$, the routine performs exactly $\texttt{m_vec_limbs} \cdot (v-1)$ multiplications. Hence, a single execution of `P1P1t_times_O` algorithm in MAYO results in a total of $v \cdot o \cdot \big(\texttt{m_vec_limbs} \cdot (v-1)\big)$ multiplications. For example, under the MAYO-SL1 parameter set (Table 6 in Appendix D) with $m = 78$, $v = 78$, and $\texttt{m_vec_limbs} = 5$, each oil coefficient triggers $5 \times 77 = 385$ multiplications, yielding a total of 240,240 calls during `P1P1t_times_O`.

```
1  static inline uint32_t mul_table(uint8_t b){
2        uint32_t x=((uint32_t)b)*0x08040201;
3        uint32_t high_nibble_mask=0xf0f0f0f0;
4        uint32_t high_half=x&high_nibble_mask;
5    return (x^(high_half>>4)^(high_half>>3));}
6
7  static inline void m_vec_mul_add(int m_vec_limbs, const
         uint64_t *in, unsigned char a, uint64_t *acc){
8    (void)m_vec_limbs;
9    uint32_t tab=mul_table(a);
10   uint64_t lsb_mask=0x1111111111111111ULL;
11   for(int i=0;i<M_VEC_LIMBS_MAX;i++)
12     acc[i]^=( in[i]       &lsb_mask)*( tab       &0xff)
13            ^((in[i]>>1)&lsb_mask)*((tab>> 8)&0xf)
14            ^((in[i]>>2)&lsb_mask)*((tab>>16)&0xf)
15            ^((in[i]>>3)&lsb_mask)*((tab>>24)&0xf);  }
16
17 static inline void P1P1t_times_O(const mayo_params_t* p
       , const uint64_t* P1, const unsigned char* O,
       uint64_t* acc){
18   const int param_o=PARAM_o(p);
19   const int param_v=PARAM_v(p);
20   const int m_vec_limbs=PARAM_m_vec_limbs(p);
21   int bs_mat_entries_used=0;
22   for(int r=0;r<param_v;r++)
23     for(int c=r;c<param_v;c++){
24       if(c==r){bs_mat_entries_used++; continue;}
25       for(int k=0;k<param_o;k++){
26         m_vec_mul_add(m_vec_limbs,P1+m_vec_limbs*
         bs_mat_entries_used, O[c*param_o+k],acc+
         m_vec_limbs*(r*param_o+k));
27         m_vec_mul_add(m_vec_limbs,P1+m_vec_limbs*
         bs_mat_entries_used, O[r*param_o+k],acc+
         m_vec_limbs*(c*param_o+k));  }
28         bs_mat_entries_used++; } }
```

Listing 1: Core routines of the nibble-sliced implementation.

Differences between nibble-sliced and bitsliced multiplications. The two multiply-accumulate subroutines, `m_vec_mul_add` and `bitsliced_m_vec_mul_add`, follow fundamentally different mechanisms. This difference is crucial from a side-channel perspective, as the underlying data representation and execution strategy directly influence the resulting power leakage patterns.

In `m_vec_mul_add` the secret oil coefficient $\mathbf{a} \in \mathbb{F}_{2^4}$ is a 4-bit value, and processed in 4-bit chunks. Therefore, the leakage is the combination of all 4-bits. In contrast, `bitsliced_m_vec_mul_add` subroutine implements the same multiplication using a *bitsliced* representation. Here, the public vector **in** is split across four 32-bit arrays, each storing one coefficient bit of many $\mathbb{F}_{2^4}$ elements in parallel. The bits of the secret oil coefficient $\mathbf{a} \in \mathbb{F}_{2^4}$ are expanded into masks that are either all zeros or all ones. When a bit of **a** is zero, the corresponding operation processes zero values, whereas when it is one, real data is propagated and accumulated. Thus, even though the control flow remains constant, the switching activity depends on each individual bit of **a**, as each bit is processed separately. In [21], this strong bit dependency was utilised to build templates for 0 and 1, and recover the secret oil coefficients bit by bit, which is not feasible for the nibbling MAYO implementation. However, this new implementation exposes a different type of leakage amenable to non-profiled CPA, which we discuss in the following section.

3.2 Recovering Oil Matrix with CPA.

Attack Point Selection. We first investigate potential leakage points, beginning with the secret-dependent computation **tab** = `mul_table(`**a**`)`. Although **tab** is reused across all loop iterations, it represents the multiplication table of a single $\mathbb{F}_{16}$ element and exhibits a fixed, limited HW range. Consequently, **tab** alone carries insufficient exploitable information and does not provide a strong distinguisher for the CPA.

We next focus on the following multiplication in the loop (Lines 12–15, Listing 1).

$$\texttt{op}[k] = \underbrace{(\mathbf{in}[j] \gg k \;\&\; \texttt{lsb_mask})}_{\text{public operand}} * \underbrace{((\mathbf{tab} \gg 8k) \;\&\; \texttt{0xf})}_{\text{private operand}}$$

Here $k \in \{0,1,2,3\}$ and the `lsb_mask` is `0x1111111111111111`. The public operand, therefore, contains a value of the form $\texttt{0x}u_1u_2\cdots u_{16}$ where each u_k is either 0000 or 0001. The private operand is a 4-bit nibble containing $2^k\mathbf{a} \mod x^4+x+1$. The multiplication, therefore, results in a value of the form $\texttt{op}[k] = \texttt{0x}t_1t_2\cdots t_{16}$ where each t_k is either 0000 or $2^k\mathbf{a}$. The HW of $\texttt{op}[k]$ becomes:

$$\mathrm{HW}(\texttt{op}[k]) = \mathrm{HW}(\text{public operand}) \cdot \mathrm{HW}(\text{private operand}).$$

The size of the key candidate **a** is 4-bit; therefore, we have a total of 5 possible HW classes for **a**. Now in $\mathrm{HW}(\texttt{op}[0])$, we have a term corresponding to the public value and a term corresponding to the secret **a**. Once the secret **a** is fixed, the value $\mathrm{HW}(\text{private operand})$ is constant for the different public values. The average of $\mathrm{HW}(\texttt{op}[0])$ is:

$$\begin{aligned}
&\mathbb{E}(\mathrm{HW}(\texttt{op}[0])) \\
&= \mathrm{HW}\left((\mathbf{tab} \gg 0)\&\texttt{0xf}\right) \cdot \frac{1}{T}\sum_{t=1}^{T} \mathrm{HW}((\mathbf{in}[j] \gg 0)\&\texttt{lsb_mask})) \\
&= \mathrm{HW}(((\mathbf{tab} \gg 0) \;\&\; \texttt{0xf}) \cdot const\,.
\end{aligned}$$

Here T is the number of different public values considered and *const* is the (constant) average of the public values. In other words, the HW of the private operand is scaled by a constant. The calculation is the same for all k, and leakage of op mainly depends upon the private operand. Now,

HW(op[0]) splits all possible 16 values into 5 classes based on the HWs of **a**: the first class consists of only one value $\{0\}$, the second class consists of the following four values $\{1,2,4,8\}$, the third class consists of the following six values $\{3,5,6,9,10,12\}$, the fourth class consists of the following four values $\{7,11,13,14\}$, and the final class consists of only one value $\{15\}$. As the second, third, and fourth classes consist of multiple values, we cannot differentiate each possible value only with HW(op[0]). We need to use other HW(op[k]), where $k = 1,2,3$ to further partition the second, third, and fourth classes. We observed that the HW leakages of the four individual multiplications (HW(op[0]), HW(op[1]), HW(op[2]), and HW(op[3])) are needed to distinguish all possible 16 values of **a** uniquely and demonstrate this with Figure 1.

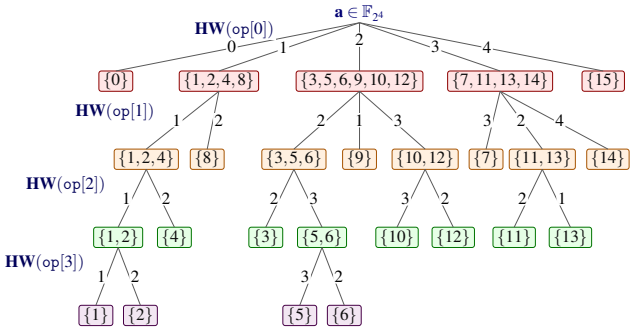

Figure 1: Explanation of HW classes for `m_vec_mul_add`

We, therefore, focus on the accumulator **acc**. It aggregates the results of these secret-dependent multiplications across iterations, and contains information from all four `op`[k]. Moreover, the accumulator is eventually written back to memory through a power-expensive *store* operation, further increasing its observability.

Power Measurement. We measure power consumption using a CW308-STM32F3 platform with an ARM Cortex-M4 STM32F3SAM4S microcontroller running at 7.37 MHz. We note that the entire MAYO code does not fit into this microcontroller. Therefore, we only measure the multiplication operations. Traces are collected during the signing algorithm, focusing on the accumulator update and its subsequent memory store. While the software uses a 64-bit accumulator, the hardware operates on 32-bit registers. Our analysis focuses on the lower 32 bits, though the same methodology applies to the upper 32-bit segments. Empirically, these traces suffice for full key recovery, so the upper 32 bits are omitted. The memory store after each update induces a strong and well-aligned leakage, which we exploit in our analysis.

During the signing process, `mayo_expand_sk` is executed. Within `mayo_expand_sk`, `expand_P1_P2` is first run followed by `P1P1t_times_O`. Figure 2 illustrates the power consumption profile the end of `expand_P1_P2` (white region) followed by the beginning of `P1P1t_times_O` (yellow region). A clear transition in the power pattern is observable between these regions, allowing the start of the multiplication phase to be identified with high precision. The power consumption profile in yellow region of Figure 2 also reveals distinct temporal markers that serve as reliable anchors for identifying each multiplication operation. Specifically, two prominent power spikes consistently prior to each multiplication. By utilizing these spikes as anchor points, we can perform precise trace segmentation and alignment, effectively compensating for clock jitter or sampling offsets. The entire trace is then partitioned into individual trace segments corresponding to each multiplication operation, where each segment (red region) depicts a single multiplication.

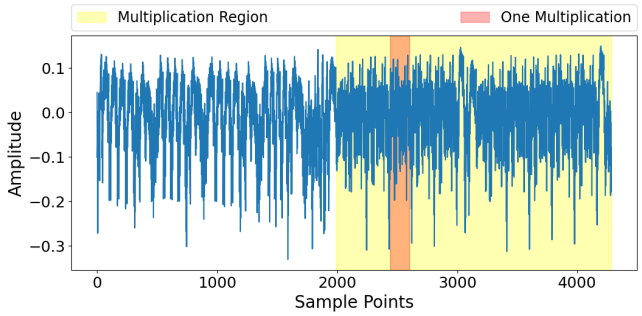

Figure 2: Temporal alignment of power traces via precomputation spikes for multiplication segmentation. The red region represents the trace segment for one multiplication and yellow region corresponds to the start of `P1P1t_times_O`.

Correlation Power Analysis. For each guess of an oil coefficient $O_{r,c} \in \mathbb{F}_{16}$, we compute `mul_table()` and evaluate the corresponding accumulator updates using the known public inputs **in**$[j]$, yielding hypothetical values for each key hypothesis. We consider the HW of each accumulator update as hypothetical leakage. Each oil coefficient gets multiplied 385 times with different public coefficients, resulting in 385 distinct traces and hypothetical leakages. Finally, we correlate the hypothetical leakage corresponding to each guess with the actual leakage (with Pearson's correlation) and select the highest correlated value as the correct oil coefficient.

As shown in Figure 3, the correct key hypothesis (highlighted in red) exhibits fast convergence. It quickly reaches a stable correlation within the first 25 trace segments for the coefficient $O_{0,0}$. Conversely, incorrect hypotheses (indicated in gray) remain clustered around the zero-correlation baseline. This difference in correlation between the correct and incorrect keys helps recover the secret.

Sequential Recovery of Oil Coefficients. The accumulator matrix **acc** is initialized with the coefficients of $\mathbf{P}_i^{(2)}$. The `P1P1t_times_O` subroutine (described in Listing 1) updates the entries of the accumulator matrix **acc**=$[acc_{r_1,c_1}]_{v\times o}$ in the

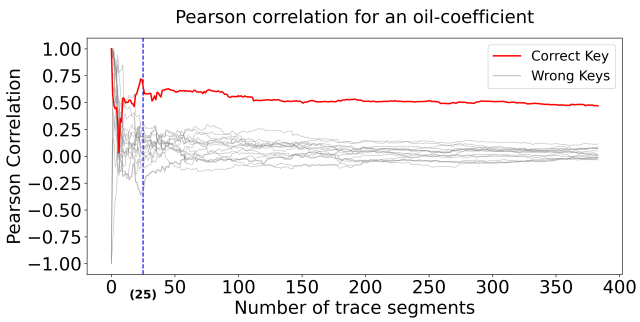

Figure 3: PCC for $O_{0,0}$ as the number of trace segments increases.

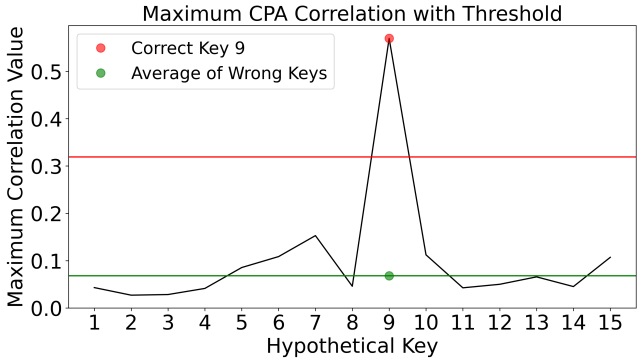

Figure 4: The thresholding idea.

following way: $\forall\, 0 \le r < v,\ r+1 \le c < v$ and $0 \le \beta < o$

$$\begin{aligned} acc_{r,\beta} &= acc_{r,\beta} + P1_{r,c} \times O_{c,\beta} \\ acc_{c,\beta} &= acc_{c,\beta} + P1_{r,c} \times O_{r,\beta} \end{aligned} \tag{2}$$

Here, $\mathbf{P}_i^{(1)} = [P1_{r,c}]_{v\times v}$ is a upper triangular matrix, so $P1_{r,c} = 0$ for $r > c$. After initialization, the first term that accumulates $acc_{\alpha,\beta}$ for any $1 \le \alpha < v$ and $0 \le \beta < o$ uses $O_{0,\beta}$ as a multiplicand. When $\alpha = 0$, the first term that $acc_{0,\beta}$ accumulates for any $(0,\beta)$ uses $O_{1,\beta}$ as an multiplicand since $(0,0)$-th term of $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})$ is 0. For any $2 \le \alpha < v$ and $0 \le \beta < o$, the second term that $acc_{\alpha,\beta}$ accumulates uses $O_{1,\beta}$ as an multiplicand. When $\alpha = 0$ or 1 then the second term that $acc_{\alpha,\beta}$ for any $(0,\beta)$ accumulates uses $O_{2,\beta}$ as an multiplicand because $(1,1)$-th term of $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})$ is 0. This sequential accumulation process continues $v-2$ times for all (α,β).

In this attack, we capture power traces during the execution of the accumulation operations described in Equation 2. Therefore, to recover all entries of the oil matrix $\mathbf{O}$, we need to follow a row-wise sequential recovery process.

Table 2 shows the recovery process of an arbitrary β, where $0 \le \beta < o$, column of $\mathbf{O}$. This recovery process follows the structured accumulation pattern of $acc_{\alpha,\beta}$, where $0 \le \alpha < v$ and assists in the sequential recovery of each row element of the β-th column of $\mathbf{O}$, $O_{\delta,\beta}$, where $0 \le \delta < v$.

Table 2: Sequential recovery pattern for oil coefficients.

Steps	**Target** (γ)	**Intermediate accumulation in** $\mathbf{acc}_{\alpha,\beta}$
0	$O_{0,\beta}$	$\mathbf{acc}_{\alpha,\beta} + \gamma P1_{\alpha,0}$
1	$O_{1,\beta}$	$\mathbf{acc}_{\alpha,\beta} + O_{0,\beta} P1_{\alpha,0} + \gamma P1_{\alpha,1}$
$\vdots$	$\vdots$	$\vdots$
δ	$O_{\delta,\beta}$	$\mathbf{acc}_{\alpha,\beta} + \sum_{k=0}^{\delta-1} O_{k,\beta} P1_{\alpha,k} + \gamma P1_{\alpha,\delta}$
$\vdots$	$\vdots$	$\vdots$
$v-1$	$O_{v-1,\beta}$	$\mathbf{acc}_{\alpha,\beta} + \sum_{k=0}^{v-2} O_{k,\beta} P1_{\alpha,k} + \gamma P1_{\alpha,v-1}$

At the δ-th step, a hypothetical value γ is recovered for the target oil coefficient, $O_{\delta,\beta}$, and the accumulator combines it with the terms accumulated previously. The accumulated value is used in the subsequent $(\delta+1)$-th step. This dependency enforces a row-wise recovery, since each coefficient influences subsequent accumulations.

During the matrix–matrix multiplication `P1P1t_times_O` subroutine, each coefficient $O_{\delta,\beta}$ contributes to $(v-1)$ entries of the symmetric matrix $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})$. Concretely, $O_{\delta,\beta}$ interacts with the set $\{P1_{0,\beta}, P1_{1,\beta}, \ldots, P1_{\beta-1,\beta}, P1_{\beta,\beta+1}, P1_{\beta,\beta+2}, \ldots, P1_{\beta,v-1}\}$, and its contribution is accumulated into the corresponding entries $acc_{\alpha,\beta}$, where $0 \le \alpha < v$ but excluding $\alpha = \delta$. These $(v-1)$ computations are used to recover each $O_{\delta,\beta}$ element of the oil matrix $\mathbf{O}$. In a nutshell, we always have enough traces (5×77) to recover each coefficient reliably, even though the recovery process is sequential.

Handling Zero Key Values for $\mathcal{A}_1$. While every non-zero oil coefficient can be recovered through the CPA attack, the correlation based distinguisher fails when the value of the coefficient is zero. This anomaly is due to the multiplication result: the output is always zero when the oil coefficient is zero, irrespective of the public component. To handle the zero-valued coefficient, we propose a threshold-based strategy.

Our key recovery procedure is as follows: while guessing the candidate coefficients we exclude the zero value. If the coefficient is originally non-zero, then we observe a correlation peak for the correct value. However, if the coefficient is zero, no significant peak can be observed. This is illustrated in Figure 5. Here, the peaks for the non-zero keys are significantly high, compared to the case while the key is originally zero. This observation provides a way to identify zero keys without computing Pearson's correlation.

To determine when a coefficient assumes a zero value, our strategy is as follows: We first recover one non-zero coefficient from the first row of the oil matrix. This can be done with trial and error, as most of the coefficients are non-zero. Once we figure out a non-zero coefficient, we consider its correlation peak (C_{peak}) (ref. Figure 4) and the average correlation values for the incorrect key guesses (C_{wavg}). Next we set a threshold as $Th = \frac{C_{peak}+C_{wavg}}{2}$. During a key recovery

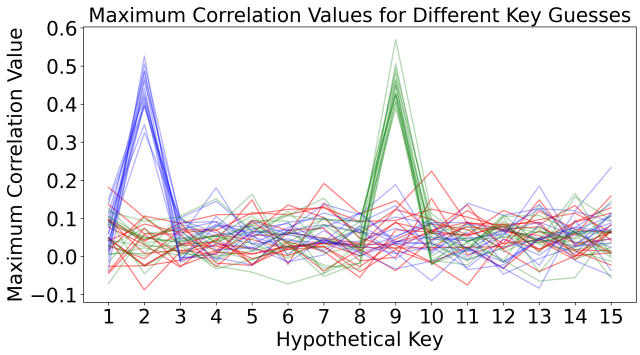

Figure 5: Maximum correlation between hypothetical power distributions (per key guess) and the real trace, for three correct keys: `0x02` (blue), `0x09` (green), and `0x00` (red); sharp non-zero peaks contrast with the flat zero-key profile to resolve recovery ambiguity.

operation, if the correlation value is $< Th$ we consider the key as zero. If it is $\geq Th$, then we consider the coefficient to be non-zero and take the coefficient corresponding to the correlation peak as the correct coefficient value.

Error Correction. While the thresholding approach outlined above works fine for 99% of the zero coefficients in our experiments, there are cases of mispredictions. Mispredictions arise when noise suppresses the peak of the correct non-zero coefficient below the threshold. Although it may still be the largest among all guesses, it is incorrectly identified as zero. This results in an incorrect accumulator update, which degrades the correlation of subsequent coefficients and can lead to a consecutive erroneous zero recoveries. Since the entries of $\mathbf{O}$ are sampled uniformly at random, long sequences of zeros in a column are highly unlikely and therefore indicate a likely misprediction.

To address such cases, we retain two candidates for each position: zero and the top-ranked non-zero CPA guess. We then apply a *look-ahead error-correction* mechanism with search depth two (see Figure 6), and select the candidate that remains consistent with subsequent recovery steps.

Starting from the current state $(\mathbf{acc}, O_{r,c})$, CPA provides a candidate γ_0 for $O_{r,c}$. As mentioned, we consider two hypotheses: $O_{r,c} = 0$ and $O_{r,c} = \gamma_0$. These hypotheses define two branches, leading to accumulator states $\mathbf{acc}_0$ and $\mathbf{acc}_{\gamma_0}$, respectively, as shown at the first level of the figure. From each state, we attempt to recover the next coefficient $O_{r+1,c}$, producing the tentative outputs g_0 (when hypothesis is $O_{r,c} = 0$) and g_{γ_0} (when hypothesis is $O_{r,c} = \gamma_0$).

Assuming that consecutive zero entries in $\mathbf{O}$ are unlikely, we use the outcome in $O_{r+1,c}$ to correct the decision for $O_{r,c}$. If the hypothesis $O_{r,c} = 0$ leads to a non-zero recovery while $O_{r,c} = \gamma_0$ leads to zero, we interpret the latter as an error and set $O_{r,c} = 0$. Conversely, if the γ_0-branch yields a non-zero value and the zero-branch does not, we interpret $O_{r,c} = \gamma_0$.

If both hypotheses produce identical outcomes (both zero or both non-zero), the error cannot be resolved at this stage. We then extend the verification to the next level, as shown in Figure 6. Each branch is expanded by considering both possibilities for $O_{r+1,c}$, resulting in four candidate paths with accumulator states $\mathbf{acc}_{0,0}, \mathbf{acc}_{0,\gamma_1}, \mathbf{acc}_{\gamma_0,0}, \mathbf{acc}_{\gamma_0,\gamma_2}$. From each of these states, CPA is applied again to estimate $O_{r+2,c}$, yielding $g_{0,0}$, g_{0,γ_1}, $g_{\gamma_0,0}$, and g_{γ_0,γ_2}.

Error correction is achieved by identifying which branch remains consistent at depth two. If a non-zero value appears only in branches originating from γ_0, we confirm $O_{r,c} = \gamma_0$; if it appears only in branches originating from 0, we correct to $O_{r,c} = 0$. In this way, the algorithm selects the hypothesis whose propagation remains stable across multiple steps, effectively filtering out noise-induced errors.

It is possible that multiple non-zero candidates, or all-zero outcomes, appear for $O_{r+2,c}$. In such cases, this procedure does not provide any additional information, and we therefore fall back to the initial candidate guess $O_{r,c} = \gamma_0$. This scenario was not observed in our experiments.

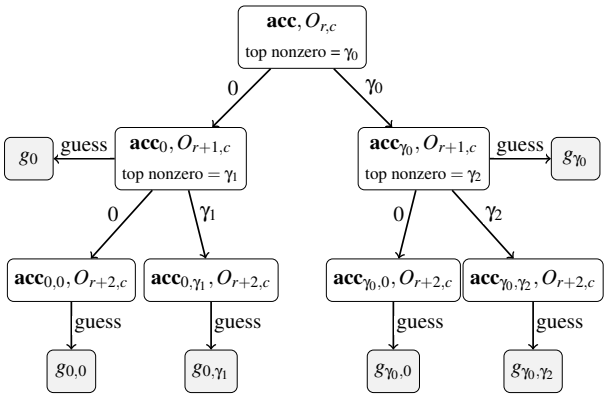

Figure 6: Look-ahead error-correction mechanism

Extension Within MAYO. We present a single-trace, non-profiled CPA targeting the `m_vec_mul_add` function during the execution of `P1P1t_times_O`. This core matrix–vector multiplication `m_vec_mul_add` is also used in `P1_times_Vt`, and hence the proposed CPA can be directly applied to that computation as well. Recall that this operation involves multiplying public matrices with vinegar vectors; therefore, the proposed single-trace CPA enables the recovery of vinegar vectors, which in turn leads to key recovery. In addition, our observations extend to all other variants of MAYO (as the computation is the same with an increased number of variables) and other MQ-DSA candidates, including UOV [9], QR-UOV [17], and SNOVA [37] (see Appendix F). The specific target operations, however, would differ across schemes.

4 SCA on Masked MAYO

As noted earlier, masking is a widely used countermeasure against SCAs [11, 20] (see Appendix E.2 for background on masking as a countermeasure). Existing masking techniques for MQ-DSAs [2, 26] follow two approaches: (i) masking sensitive variables [26], (ii) masking public variables involved in secret-dependent computations [2].

We begin with a masking technique that splits the secret coefficient into two additive shares in Section 4.1. In Section 4.2 and Section 4.3, we explore SCA on public value masking based on adversary $\mathcal{A}_2$ and $\mathcal{A}_3$ respectively.

4.1 SCA on Additive Secret-Value Masking

We first analyze a basic masking approach that splits the secret oil matrix $\mathbf{O}$ into two additive shares. Concretely, the oil matrix is shared as $\hat{\mathbf{O}}_1 = \mathbf{O} + \hat{\mathbf{O}}_0$, where $\hat{\mathbf{O}}_0$ is sampled uniformly at random. This allows the independent computation of the linear system coefficients $\mathbf{L}_i$ for each share:

$$\begin{aligned} \hat{\mathbf{L}}_{i,0} &= \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\hat{\mathbf{O}}_0 + \mathbf{P}_i^{(2)}, \\ \hat{\mathbf{L}}_{i,1} &= \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\hat{\mathbf{O}}_1 + \mathbf{0}_{v\times o}. \end{aligned} \tag{3}$$

The final coefficients are then reconstructed via the XOR operation $\mathbf{L}_i = \hat{\mathbf{L}}_{i,0} + \hat{\mathbf{L}}_{i,1}$.

Since the public key components remain known to the attacker, the CPA described in Section 3 remains effective even in this masked setting. Consequently, the secret remains vulnerable to the adversary $\mathcal{A}_1$. In particular, the attack recovers the random shares $\hat{\mathbf{O}}_0$ and $\hat{\mathbf{O}}_1$, which can then be combined to reconstruct the original oil matrix.

The same observation naturally extends to *higher-order masking*, since each share of the secret value is processed independently with the same public value. Hence, increasing only the masking order does not mitigate the proposed attack.

4.2 SCA on Public-Value Masking under $\mathcal{A}_2$

Aulbach *et al.* [2] proposed an alternative masking of the public values during the execution of the subroutine `P1P1t_times_O`. Here, the public variables that interact with secret data are split into additive shares to protect secret-dependent computations [2]. The masking strategy leverages the linear structure of the matrix construction $\mathbf{L}_i = \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$, which is linear in both $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. Instead of masking the oil matrix $\mathbf{O}$, the public matrices $\mathbf{P}_i^{(1)}$ are split into two additive shares $\mathbf{P}_i^{(1)} = \hat{\mathbf{P}}_{i,0}^{(1)} + \hat{\mathbf{P}}_{i,1}^{(1)}$, where $\hat{\mathbf{P}}_{i,0}^{(1)}$ is sampled uniformly at random and $\hat{\mathbf{P}}_{i,1}^{(1)} = \mathbf{P}_i^{(1)} + \hat{\mathbf{P}}_{i,0}^{(1)}$. The matrix $\mathbf{P}_i^{(2)}$ is shared in a degenerate manner by setting $\hat{\mathbf{P}}_{i,0}^{(2)} = \mathbf{P}_i^{(2)}$, and $\hat{\mathbf{P}}_{i,1}^{(2)} = \mathbf{0}_{v\times o}$. Using these shares, the computation of $\mathbf{L}_i$ is carried out independently on each share by invoking the routine computing $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})\mathbf{O} + \mathbf{P}_i^{(2)}$ twice:

$$\begin{aligned} \hat{\hat{\mathbf{L}}}_{i,0} &= \left(\hat{\mathbf{P}}_{i,0}^{(1)} + \hat{\mathbf{P}}_{i,0}^{(1)\top}\right)\mathbf{O} + \hat{\mathbf{P}}_{i,0}^{(2)}, \\ \hat{\hat{\mathbf{L}}}_{i,1} &= \left(\hat{\mathbf{P}}_{i,1}^{(1)} + \hat{\mathbf{P}}_{i,1}^{(1)\top}\right)\mathbf{O} + \hat{\mathbf{P}}_{i,1}^{(2)}. \end{aligned} \tag{4}$$

By linearity, recombining the two shares yields $\hat{\hat{\mathbf{L}}}_{i,0} + \hat{\hat{\mathbf{L}}}_{i,1} = \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O} + \mathbf{P}_i^{(2)} = \mathbf{L}_i$, thereby preserving functional correctness. In this approach, the attacker does not know these shares. As a result, the adversary cannot accurately predict the intermediate leakage. As for the implementation, `P1P1t_times_O` is executed twice, we refer readers to Listing 3 in Appendix H. Given public coefficients are inaccessible, the adversary operates under blind scenario. As a result, the CPA methodologies described in Section 3 and Section 4.1 are not applicable in this context.

To analyze the side-channel security of such implementations, we consider two types of adversaries: (i) *a strong adversary with access to a clone device,* $\mathcal{A}_2$ and (ii) *a weak adversary who has no access to the clone device,* $\mathcal{A}_3$. In both settings, we use only a single trace and partition it into segments in the same manner as in Section 3. We therefore adapted the DL-BSCA framework from [34] to process these trace segments and perform the horizontal attack. Note that the original DL-BSCA [34] was not a horizontal single-trace attack, which is the case for us.

We first recall the threat model of $\mathcal{A}_2$. In this scenario, the attacker has a cloned device, which provides them with complete knowledge of the mask values, the public variables, and the oil coefficients used by the cloned device in the offline stage. However, during the attack phase, the adversary has access only to the power traces from the target device, not to the mask values. We target the operations defined in Listing 1 similar to the CPA in Section 3.2. However, in this case, we cannot target the accumulator as it is randomized. Rather, we target the HW for each operation in Listing 1:

$$\begin{aligned} h_0 &= \mathrm{HW}(\texttt{(in[i]\&lsb_ask)*(tab \& 0xff)}) \\ h_1 &= \mathrm{HW}(\texttt{((in[i]»1)\&lsb_ask)*((tab » 8)\&0xf)}) \\ h_2 &= \mathrm{HW}(\texttt{((in[i]»2)\&lsb_ask)*((tab » 16)\&0xf)}) \\ h_3 &= \mathrm{HW}(\texttt{((in[i]»3)\&lsb_ask)*((tab » 24)\&0xf)}). \end{aligned}$$

As explained in Section 3.2, each of h_js solely depends upon the HW of the private part, and the average HW of the public part only works as a scaling factor. Therefore, even though the HW of the public part is unknown, the oil coefficient can be distinguished based on the scaled h_js. Since each h_j is the HW of multiplication between a 8-bit and a 4-bit values, h_js take values in range 0-32.

Profiling Phase: Multi-Task DNN. During the profiling phase, for each coefficient hypothesis, we collect the corresponding set of trace segments and train a dedicated multi-task

DNN on this data. This procedure is repeated for all oil coefficient hypotheses, enabled by access to a clone device. Each multi-task DNN takes in the trace segments and outputs the probabilities of $\boldsymbol{h} = (h_0, h_1, h_2, h_3)$. The overall architecture of the multi-task is depicted in Figure 7. The architecture consists of a shared backbone (comprising l fully connected layers) followed by four independent branches. Each branch is constructed with two fully connected layers to map the shared features to their corresponding h_j. We employ a softmax activation function at the end of each branch to generate the probability distribution for each h_j.

We define a multi-task loss function that aggregates the individual losses from each branch. Specifically, for each branch $j \in \{0,1,2,3\}$, we compute the categorical cross-entropy loss between the predicted probability distribution of h_j and the corresponding ground-truth label. The total objective function, $\mathcal{L}_{total}$, is expressed as the weighted sum of these individual losses:

$$\mathcal{L}_{total} = \sum_{i=1}^{4} \mathcal{L}_{CCE}(\hat{y}_j, y_j^*) \tag{5}$$

where y_j^* is the corresponding ground-truth probability and $\hat{y}_j$ is the predicted probability for h_j.

Attack Phase: A Novel Distinguisher. In this attack phase, we present a *novel distinguisher* that does not require any information about the random share of public matrices. The attack phase is further split into two stages: offline and online stages. We observe that for a fixed coefficient k the empirical mean of $\tilde{\boldsymbol{h}}_k = \frac{1}{N_o}\sum_{i_o=1}^{N_o}(h_0^{(k,i_o)}, h_1^{(k,i_o)}, h_2^{(k,i_o)}, h_3^{(k,i_o)})$ varies significantly across different coefficient values. Here, $h_j^{(k,i_o)}$ denotes the variable h_j obtained using the coefficient-dependent value `tab` (derived from k), together with the i_o^{th} sampled public variable of `in[i]`. To exploit this, the offline stage involves generating N_o uniformly random public variables `in[i]` for each candidate coefficient value k to compute the empirical mean $\tilde{\boldsymbol{h}}_k$ and the corresponding empirical covariance matrix $\tilde{\boldsymbol{\Sigma}}_k$ of $(h_0^{(k)}, h_1^{(k)}, h_2^{(k)}, h_3^{(k)})$.

Subsequently, during the online stage, the attack trace segments of the target device are given to the trained multi-task DNN. Such trace segments can be gathered from a single trace.Then, we apply the arg max for each of the branch to obtain the predict HW value of h_j which we denote as $\hat{h}_j$ (i.e. $\hat{\boldsymbol{h}} = (\hat{h}_0, \hat{h}_1, \hat{h}_2, \hat{h}_3)$. We compute the score for each of the coefficients k based on the log-likelihood:

$$\begin{aligned}\text{score}[k] &= \sum_{i_a=1}^{N_a} \log \Pr(\boldsymbol{h}_k \mid \tilde{\boldsymbol{h}}_k, \tilde{\boldsymbol{\Sigma}}_k) \\ &= \sum_{i_a=1}^{N_a} -\frac{1}{2}(4\log(2\pi) + \log|\tilde{\boldsymbol{\Sigma}}_k| + (\hat{\boldsymbol{h}}_{i_a} - \tilde{\boldsymbol{h}}_k)^T \tilde{\boldsymbol{\Sigma}}_k^{-1}(\hat{\boldsymbol{h}}_{i_a} - \tilde{\boldsymbol{h}}_k))\end{aligned}$$

where $|\tilde{\boldsymbol{\Sigma}}_k|$ is the determinant of $\tilde{\boldsymbol{\Sigma}}_k$ and N_a is the total number of attack trace segments. The attacker organizes the candidate

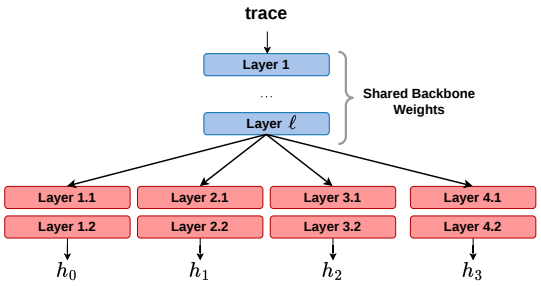

Figure 7: Multi-Task DNN Architecture.

coefficient values into a guess vector $\boldsymbol{G} = [G_0, \ldots, G_{|\mathcal{K}-1|}]$ by sorting their log-likelihood scores in descending order. In this structure, G_0 represents the most probable coefficient candidate, while $G_{|\mathcal{K}-1|}$ represents the least probable. The specific position of a coefficient value k within this sorted vector, denoted as $g(k)$, defines its rank. The effectiveness of an attack is often measured by Guessing Entropy (GE). It represents the expected rank of the true coefficient value k^* across a series of independent trials [35]. An attack is fully successful when $GE = 0$ across N_a attack trace segments, meaning the correct coefficient is ranked first. We denote by $NTGE$ the minimum number of attack trace segments required to achieve $GE = 0$.

During the attack phase, the attack trace segments are evaluated on all trained multi-task models DNN_k, for each coefficient (i.e., oil coefficient) hypothesis k to compute the corresponding GE, denoted as GE_k. In this context, GE_k serves as the distinguishing metric. For the correct coefficient hypothesis $\hat{k}$, the $GE_{\hat{k}}$ converges to 0, whereas for the remaining 14 models ($k \neq \hat{k}$), the GE_k remains significantly greater than 0. This enables the adversary to distinguish between the correct and incorrect coefficient hypotheses.

Result. We try to recover all 15 non-zero coefficient values. In each of these coefficients, we train the network with 10000 profiling trace segments from the same fixed coefficient. We evaluated over 100 different multi-task DNN architectures, following the hyperparameter search space detailed in Table 8 of Appendix I. As illustrated in Table 3, we successfully recovered all coefficients, achieving a $GE = 0$ with only a single attack trace segment in each case.

Table 3: Attack performance under adversary $\mathcal{A}_2$

Coefficient Value	1	2	3	4	5	6	7	8
GE	0	0	0	0	0	0	0	0
$NTGE$	1	1	1	1	1	1	1	1
Coeffieient Value	9	10	11	12	13	14	15	
GE	0	0	0	0	0	0	0	
$NTGE$	1	1	1	1	1	1	1	

Handling Zero-Value Coefficient for $\mathcal{A}_2$. We first observe that when the secret coefficient value is zero, the resulting product of the multiplication remains zero regardless of the other input. Therefore, the distinguisher stated above could not help to distinguish the coefficient when it is zero. To overcome this, we implemented a preliminary binary classifier using a Multi-Layer Perceptron (MLP) specifically designed to distinguish between trace segments generated by a zero coefficient and those from non-zero coefficients. This model achieves a 99% classification accuracy, allowing it to function as a highly reliable pre-processing filter. By applying this classifier first, we can identify zero-cofficient instances before proceeding with the primary attack stages described above.

4.3 SCA on Public-Value Masking under $\mathcal{A}_3$

For the sake of completeness, we also evaluate the case where the adversary does not possess a clone device. In this scenario, the adversary only has access to traces (also trace segments) from a protected implementation of the target. This poses a significant challenge because the adversary lacks the public matrices, making it impossible to generate ground-truth labels for training. In other words, the adversary does not have access to the HW values used for distinguishing the coefficient.

Following the methodology in [34], we divide the target trace segments into training and attack sets. However, since the training set is unlabeled (HW values are unknown), we must apply a labeling technique to enable the DNN to learn. This effectively transforms the task into training a DNN under a noisy label setting, as the generated labels are estimates rather than known values. A quick overview of the process is as follows.

(a) Identify relevant Points of Interest (PoIs) within the trace segments for each h_j.
(b) Labeling a subset of the trace segments for DNN training.
(c) Predict the labels of remaining trace segments using DNN for coefficient recovery

For Step **(a)**, as PoI selection in full blind selection is an open challenge, we will follow previous works and use correlation to find these PoIs. We leave this area as future work. As for the trainning of DNN in step **(b)** and the coefficient recovery step in step **(c)** are the same as in the profiled setting. Therefore, we will focus on the labeling of the trace segments.

Regarding labeling the trace segments in step (b), while previous works have used Slicing [38], Variance Analysis [13], and MultiPoint-Clustering [34], these methods have clear drawbacks. Slicing and Variance Analysis typically focus on a single PoI, which may not capture enough information for recovery. Conversely, MultiPoint-Clustering is computationally impractical here; since each h_j represents a Hamming Weight with only 33 possible classes, the resulting class space is far too large for clustering to be feasible.

To overcome these issues, we propose MultiPoint-Slicing. This technique generalizes the Slicing concept to incorporate multiple sample points, providing a more robust labeling process for the target trace segments while avoiding the heavy computational costs associated with clustering.

MultiPoint-Slicing. Firstly, we describe the slicing labeling technique for h_j as the distribution is slightly different due to how the operation works. The Linge et al. [38]'s Slicing method assumes a monotonic relationship between the power amplitude (l_v) at a PoI and the Hamming weight (h_v) of the underlying intermediate value. To label N trace segments without ground-truth data, the process has three main steps:

1. **Sorting:** Trace segments are sorted in ascending order based on their measured power amplitudes at the selected PoIs for each h_j.
2. **Slicing**: Let $x_j = (in[i] \gg j)\&lsb_ask$ and $r_j = ((tab \gg 8j)\&0xf)$, then $h_j = HW(x_j * r_j) = HW(x_j) * HW(r_j)$. Hence, x_j is an 8-bit variable while r_j is an 4-bit variable. Assuming that each bit is uniformly distributed, then for N trace segments, we slice them into 33 classes. For a given class c, the expected number of trace segments N_h is derived from the combinatorial probability:
$$N_h = N * \sum_{\kappa \cdot s = h} \frac{\binom{8}{\kappa}\binom{4}{s}}{2^{12}} \tag{6}$$
where the terms $\binom{8}{\kappa}$ and $\binom{4}{s}$ represent the number of possible bit combinations that result in a specific Hamming weight for the 8 bit variable x_j and the 4-bit variable r_j, respectively.
3. **Label Assignment:** The sorted trace segments are partitioned into 33 classes according to the calculated N_c, where the first N_0 trace segments are assigned as $h_v = 0$, the subsequent N_1 trace segments are assigned $h_v = 1$ the process continues until the final trace segments are labeled as $h_v = 32$.

Building on this foundation, we propose MultiPoint-Slicing labeling. Its overall structure and methodology are presented in Figure 8. Suppose from the PoI selection, we obtain a set of points of interest PoI_j, that corresponding to the variable h_j. Slicing is then applied independently at each PoI across all traces to generate labels based on the observed amplitudes. Consequently, for each trace $\boldsymbol{t}^{(i_a)}$, a collection of labels is obtained over the set PoI_j. We then perform weighted majority voting to determine the final label $Y_j^{(i_a)}$, which serves as the estimate of h_j for the trace $\boldsymbol{t}^{(i_a)}$:

$$Y_j^{(i_a)} = \arg max_{h \in \{0,\ldots,32\}} \left(\frac{|PoI_j^h|}{N_h}\right) \tag{7}$$

where $|PoI_j^h|$ is the total number of sample points in PoI_j that is labeled as h.

It is worth mentioning that the number of traces in the clustering and training phase is not limited to a single trace

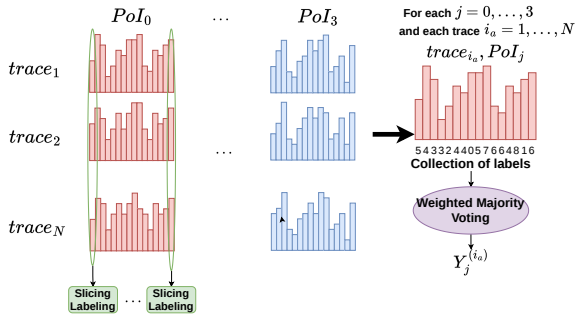

Figure 8: MultiPoint-Slicing.

here. This is because, with masking, the same public and private key can generate an unlimited number of different traces. This extra advantage for the adversary is purely an artifact of the public value masking.

Result. We test 100 different models using the same hyperparameter search space in Appendix I and compute their guessing entropy. The results are described in Table 4 (visualization in Figure 8 of Appendix J). We observe that GE decreases to less than 2 for the coefficients tested. This confirms that it is indeed a more challenging setting than before. Since the proposed labeling strategy is heuristic, the labels used to train the multi-task DNN inherently contain noise, which leads to $GE > 0$. Nevertheless, the observed reduction in guessing entropy is substantial, with the correct coefficient consistently ranked within the top three candidates on average.

Limitations. Other than the PoI selection, which is not fully blind (as in previous work on blind SCA), the described attack does not explicitly account for zero-valued oil coefficients. As $GE < 2$, the correct coefficient is expected to appear among the top three non-zero candidates. Therefore, the adversary must, on average, enumerate four possible values per coefficient to recover the correct one. It is a well-known fact that recovering a single column of the oil matrix is sufficient to reconstruct the full oil matrix [33]. Hence, the adversary must evaluate approximately 4^{n-o} possibilities to recover one column. Nevertheless, this remains more efficient than a brute-force search, which requires $(2^4)^{n-o}$ evaluation. Hence, this shows that under the adversary model $\mathcal{A}_3$, the attack becomes significantly more difficult than under $\mathcal{A}_2$. On the other hand, it also shows that even in the most challenging adversarial setting, public-key masking does not provide sound security.

Generality of the Threat Models. We extend our attacks to *higher-order public-value masking*. Here, the public matrices are split into additional shares, each of which is multiplied by the same oil matrix and exhibits the same leakage characteristics. Consequently, increasing the masking order does not provide additional resistance against either the profiled attack under $\mathcal{A}_2$ or the blind attack under $\mathcal{A}_3$; instead, it provides more trace segments that process the same secret coefficient.

Table 4: Attack performance under adversary $\mathcal{A}_3$.

Coefficient Value	1	2	3	4	5	6	7	8
GE	0.74	1	0.24	1.22	0.92	0.39	0	0.21
NTGE	–	–	–	–	–	–	11	–
Coefficient Value	9	10	11	12	13	14	15	
GE	1.86	0.02	0.84	1.14	0.8	0	0.54	
NTGE	–	–	–	–	–	1	–	

Although the attacks presented in this work are instantiated on MAYO, the underlying threat models are not specific to this construction. Instead, they capture side-channel scenarios at the implementation level that naturally arise in MQ-DSAs.

Under adversary $\mathcal{A}_1$, the attacks in Sections 3 and 4.1 exploit a horizontal leakage pattern, where a fixed secret value is repeatedly combined with many public values in a single signature generation. Consequently, a single execution produces a large number of secret-dependent intermediate values corresponding to the same key hypothesis. This implementation pattern is not unique to MAYO; similar repeated secret–public interactions also occur in UOV, QR-UOV, and SNOVA, although the corresponding target routines differ across implementations. Therefore, the proposed CPA methodology can be adapted by identifying the analogous secret-dependent multiplication routine in each implementation.

Under adversaries $\mathcal{A}_2$ and $\mathcal{A}_3$, we consider implementations protected using public-valued masking. To the best of our knowledge, such masking techniques have only been proposed for MAYO and UOV [2]. Nevertheless, the underlying masking methodology is independent of the algebraic structure of the scheme and can be applied to other MQ-based signature schemes whenever the target secret-dependent computation is expressed as repeated operations between public and secret values. Consequently, the profiled and blind attack models introduced in this work naturally extend to masked implementations of other MQ-based signature schemes if similar public-valued masking is adopted.

5 Potential Countermeasure

The attacks described in this work depend on the sequential execution of the matrix-matrix multiplication $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O}$. Therefore, shuffling the sequence of the multiplications and accumulations in every execution of the matrix-matrix multiplication $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O}$ would make the attack more difficult. A similar countermeasure has been proposed in [30]. However, if an adversary can perform an advanced side-channel attack that leaks the sequence of operations along with the power traces of the corresponding multiplication (and

accumulation) operations, the shuffled implementation also becomes subject to side-channel attacks [19].

A combination of masking and refreshing is another countermeasure proposed to provide protection against horizontal/simple single-trace/multi-trace power attacks. Recently, this countermeasure has been proposed to secure the number-theoretic transform (NTT)-based polynomial multiplication used in Kyber's decapsulation operation [31]. Here, masking is applied only to secret-related/sensitive variables, and the masking shares for those variables are refreshed after each use. However, the application of the masking and refreshing technique incurs significant overhead. As shown in [31], the first-order masking and refreshing technique has approximately $11\times$ overhead compared to the first-order masked NTT implementation. As mentioned in Section 3.1, in the unprotected matrix-matrix multiplication $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O}$, the multiplication is performed $\left(v \cdot o \cdot \texttt{m_vec_limbs} \cdot (v-1)\right)$ times, e.g. for MAYO-SL1, this number is 240240. The number of multiplications doubles during first-order masking, and applying refreshing to each element of $\mathbf{O}$ after every multiplication operation will cost approximately $\left(v \cdot o \cdot \texttt{m_vec_limbs} \cdot (v-1)\right)$ additional refreshing operations.

An estimation of the implementation cost in terms of execution time (microseconds) of integrating first-order masking and refreshing countermeasure on the $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O}$ operation is presented in Table 5. It is implemented in C, compiled with GCC version 11.4.0 with the -O0 optimization flag, and executed on a laptop with an *i*7-10610U CPU (1.80 GHz × 8) and Ubuntu 22.04.5 operating system. The overhead factor for the integration of only the first-order masking countermeasure is $31\times$, whereas the overhead factor increases to $59\times$ when the combination of first-order masking and refreshing countermeasure is applied.

Table 5: Comparing the cost of applying first-order masking and a combination of first-order masking and refreshing countermeasure to the target operation of MAYO-SL1.

	Unmasked	Masked	Masking + Refreshing
Time (μs)	1522	47314	89881
Overhead	$1\times$	$31\times$	$59\times$

After applying the masking and refreshing countermeasure on the target operation $\left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top}\right)\mathbf{O}$, we perform a side-channel analysis and compute per-coefficient Guessing Entropy, as shown in Figure 9. The resulting Guessing Entropy remains close to that of random guessing, which demonstrates that the proposed masking and refreshing countermeasure effectively mitigates our attack.

First-order masking is ineffective against higher-order differential power attacks; therefore, higher-order masking is necessary. To safeguard all vulnerable areas in the signature and key-generation algorithms of MAYO from horizontal and arbitrary-order differential power side-channel attacks, we need to implement higher-order masking and refreshing techniques. To harvest the real security benefits of higher-order masking, it is desirable to operate in a noisy environment. This noise can be native to the device or enhanced using hiding methods, such as shuffling, for low-noise targets. This will significantly increase the cost of the secure implementation of the MAYO signature and key-generation algorithm. Therefore, an optimized protected implementation of MAYO is required that strikes a balance between security and performance overhead; we leave it as future work.

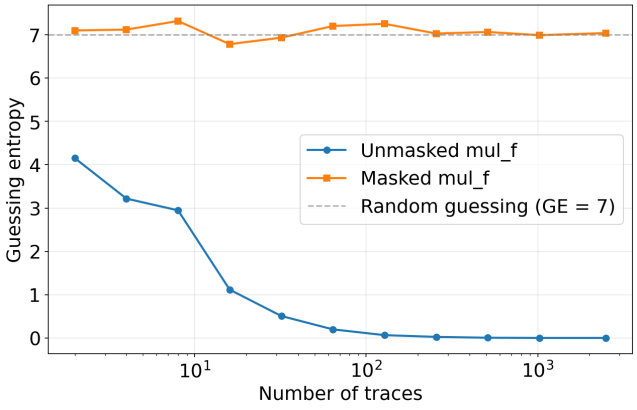

Figure 9: Side-channel analysis on a coefficient of the target operation after applying the masking and refreshing.

6 Conclusion

We present a comprehensive single-trace SCA of MAYO, covering both unprotected and masked implementations under a hierarchy of progressively weaker adversary models. For unprotected nibble-based MAYO, we show that classical single-trace CPA suffices to recover the full secret key, giving the first successful single-trace attack in this setting; the same methodology also applies to additive secret-sharing when public values are observable. When masking randomizes public values, classical techniques become ineffective, motivating new techniques that enable attacks without online access to masked public variables. We further design deep learning attacks, including a multi-task DNN for $\mathcal{A}_2$ enabling full key recovery. Further, multi point slicing extension of DL-BSCA is proposed for the harder $\mathcal{A}_3$ model. Across all settings, our methods reduce guessing entropy below 2 per key component and can be mounted using commodity hardware. These results demonstrate that even basic single-trace attack setups can seriously threaten MQ-DSA implementations, underscoring the need for stronger protections.

A Open Science Policy

Our research results will be made publicly available to support transparency and reproducibility. All datasets used in this work will be made publicly available, along with the corresponding implementation and side-channel traces to facilitate independent verification and further research.

B Ethical Considerations

This work studies side-channel vulnerabilities in MAYO, a candidate post-quantum signature scheme, with the goal of improving implementation security prior to deployment. The proposed attacks are evaluated in controlled settings and aim to inform stronger countermeasures.

All experiments are conducted on publicly available reference implementations; no live systems, user data, or human subjects are involved. The collected traces contain no sensitive information.

We follow responsible disclosure by clearly stating assumptions and countermeasures and aim to support secure standardization and implementation of post-quantum cryptography.

C Use of AI tools.

AI-based tools were used only to assist in language refinement and presentation. All technical content, experiments, and conclusions are solely the responsibility of the authors, and no AI system was used to generate scientific results or claims.

D Parameters for MAYO

Table 6 lists the MAYO parameters defining its security level. See the MAYO specification [7] for details.

Table 6: MAYO Parameter Sets [7]

Security level	n	m	o	k	q
128 (SL-1)	86	78	8	10	16
192 (SL-3)	118	108	10	11	16
256 (SL-5)	154	142	12	12	16

E Background on SCA and Masking

For completeness, and due to space constraints in the main paper, we recall standard background on power-based side-channel leakage here. We first recall the leakage models used throughout this work, namely Hamming weight (HW) and Hamming distance (HD). We then recall Correlation Power Analysis (CPA), the primary technique used to recover secret-dependent intermediates from measured traces, along with its deep learning-based, profiled extension used for the blind setting (DL-assisted Blind SCA). Finally, we recall masking, the countermeasure whose security against these attacks we analyze in Section 4.1 onward.

E.1 CPA and Deep Learning based SCA

Correlation Power Analysis (CPA). CPA is a non-profiled side-channel technique that exploits the dependence of power consumption on the data being processed, typically captured using a Hamming weight (HW) or Hamming distance (HD) leakage model [10]. Given a target intermediate that depends on both known data and a secret key hypothesis, CPA measures how well the predicted leakage of each hypothesis correlates with the observed power traces, using the Pearson correlation coefficient (PCC) as the distinguisher.

The PCC between $\boldsymbol{x}$ and $\boldsymbol{y}$ is given by

$$\rho(\boldsymbol{x},\boldsymbol{y}) = \frac{\sum_{j=0}^{N-1}\left(x^{(j)}-\bar{\boldsymbol{x}}\right)\left(y^{(j)}-\bar{\boldsymbol{y}}\right)}{\sqrt{\sum_{j=0}^{N-1}\left(x^{(j)}-\bar{\boldsymbol{x}}\right)^2}\sqrt{\sum_{j=0}^{N-1}\left(y^{(j)}-\bar{\boldsymbol{y}}\right)^2}} \tag{8}$$

where $\boldsymbol{x} = (x^{(0)},\ldots,x^{(N-1)})$ and $\boldsymbol{y} = (y^{(0)},\ldots,y^{(N-1)})$, and $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{y}}$ denote the respective means. The PCC takes values in $[-1,1]$ and quantifies the strength of the linear relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$; the correct key hypothesis is the one whose predicted leakage attains the highest correlation with the measured traces.

Concretely, a CPA attack proceeds through the following five steps.

1. *Intermediate Value Selection*. Select a target intermediate as a function of known data and secret key material.
2. *Power Measurement*. Measure power traces and identify time samples corresponding to this computation.
3. *Hypothesis Computation*. Compute hypothetical values for all candidates of the secret-dependent values.
4. *Leakage Modeling*. Map hypotheses to predicted leakage using the Hamming weight or Hamming distance.
5. *Correlation-Based Key Distinguishing*. Compute the PCC between the predicted and measured leakage for each hypothesis, and select the key hypothesis with the highest correlation.

Deep Learning-assisted Blind SCA. DL based attacks are variants of profiled attacks where the adversary can do extensive leakage profiling [4]. The main idea is to relate a hypothetical leakage model (HW, HD, or identity leakage) with the profiling power traces by learning a model. Later, at the attack phase, the traces to be used for attack are queried against the model to recover the probability distributions of the target variable. A score is then assigned to each key hypothesis, typically derived from the log-likelihood of its estimated probability distribution. Following the principle of

Maximum Likelihood Estimation (MLE), the key hypothesis with the highest score is selected.

Formally, given a set of observations $\boldsymbol{y} = (y^{(0)}, \ldots, y^{(N-1)})$ and a parametric model with parameter θ, the likelihood function is defined as

$$f(\theta \mid \boldsymbol{y}) = \prod_{j=0}^{N-1} Pr\big(y^{(j)} \mid \theta\big),$$

where $Pr(\cdot \mid \theta)$ denotes the probability density (or mass) function. The maximum likelihood estimate is then given by

$$\hat{\theta} = \arg\max_{\theta} \; f(\theta \mid \boldsymbol{y}),$$

or, equivalently, by maximizing the log-likelihood:

$$\hat{\theta} = \arg\max_{\theta} \sum_{j=0}^{N-1} \log Pr\big(y^{(j)} \mid \theta\big).$$

In the blind setting, where the adversary lacks access to public variables, this profiling and scoring procedure is adapted into the DL-BSCA framework, which we build on in Section 4.3.

E.2 Masking as a Countermeasure

Masking [11, 20] is a provably secure countermeasure against SCA that decorrelates leakage from sensitive intermediates by randomizing computations with independent masks. In Boolean masking, an intermediate x is represented as

$$x = \hat{x}_1 + \hat{x}_2 + \cdots + \hat{x}_d,$$

where each share $\hat{x}_i$ is uniformly random except for the constraint that their XOR equals x. A d-th-order secure masking scheme guarantees that any leakage combination of order less than d is independent of the secret. Despite its performance overhead, masking remains the only countermeasure with composable theoretical guarantees against SCAs, and thus forms the cornerstone of side-channel-secure implementations of post-quantum cryptographic schemes.

F SCA on UOV, QR-UOV and SNOVA

Now, we discuss how to extend our single-trace non-profiled SCA to other MQ-DSAs, like UOV [9], QR-UOV [17], and SNOVA [37]. Although implementations of SNOVA and QR-UOV utilize a compositional structure, UOV [9] uses a structure based on subspaces similar to MAYO . However, all three schemes compute $\mathbf{L}_i$ as shown in 1, either during key-generation or during the signature generation phase. Thus, although the exact attack strategy rely on the implementation of the matrix-matrix multiplication and accumulation operation $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}) \cdot \mathbf{O} + \mathbf{P}_i^{(2)}$, our proposed attack can be adapted to UOV, SNOVA, and QR-UOV. We discuss our single-trace non-profiled SCA on UOV implementations in a simulated, high-noise environment.

F.1 Single Trace Non-profiled SCA on UOV

The UOV specification [9] uses two parameter sets: $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. The $\mathbb{F}_{16}$ is only used in `uov-Is` parameters, and $\mathbb{F}_{256}$ is used for all security levels. The multiplication `gf16v_mul_u32` is used for $\mathbb{F}_{16}$ multiplication, and the subroutine `gf256v_mul_u32` is used for $\mathbb{F}_{256}$ field multiplication. We analyse the leakage point for both multiplications, and the cause behind leakages.

Leakage points for `gf256v_mul_u32`. The output of the subroutine `gf256v_mul_u32`[1] is the primary leakage point, as it aggregates multiple secret-dependent operations within the multiplication loop. The key source of leakage is the update `r32 ^= a32 * ((b32 >> i) & 1)` where the accumulator `r32` undergoes a state transition only when the i-th bit of the secret multiplier b equals one. This conditional update introduces a clear data-dependent switching behavior.

At the hardware level, the Hamming distance of `r32` between successive iterations depends on the secret-dependent evolution of `a32`. Consequently, power or electromagnetic leakage observed when `r32` is updated exhibits a high signal-to-noise ratio, enabling an attacker to distinguish multiplier bits based on the presence or absence of observable switching activity. Hence, the output of the multiplication constitutes a natural and effective leakage point under standard side-channel leakage models. y the same reasoning, the output of the subroutine `gf16v_mul_u32` is the main leakage point $\mathbb{F}_{16}$ multiplication.

For `uov-Ip` parameters [9], $n = 112$ $m = 44$, $v = n - m = 68$, and $q = 256$. The subroutine `gf256v_mul_u32` is invoked a total of 603 times per oil coefficients, resulting in 1,804,176 invocations during a single UOV-`uov-Ip` signing operation. Consequently, a sufficient number of trace segments are available, making a non-profiled single-trace attack on UOV feasible. Once a single oil coefficient is recovered, the sequential nature of the underlying matrix multiplication allows the attacker to progressively recover the entire oil matrix.

F.2 Extension to SNOVA and QR-UOV

We now examine this leakage source in SNOVA [37] and QR-UOV [17]. In SNOVA, the secret vinegar vector is multiplied repeatedly by the public key matrices during signing. This computation is carried out by the `subroutine gf _mat _mul _add` [36], which is invoked once per public matrix and therefore reuses the same vinegar coefficients across many calls within a single signing execution, making it the natural attack target. In QR-UOV, the same role is played by the `VECTOR_V_MUL_SYMMETRIC_MATRIX_VxV` subroutine [16], where the sensitive vinegar component acts on the public component multiple times in the course of evaluating the quadratic map. As with MAYO, the underlying cause in

[1] UOV GitHub: `gf16.h` file

both schemes is the repeated reuse of a single secret operand across many public inputs within one execution, which is precisely the condition our CPA methodology requires; the specific field or ring arithmetic used by each subroutine determines how this leakage manifests and must be characterized separately for each. An experimental validation of both attacks is left for future work. Table 7 summarizes the attack surface across the three schemes considered in this appendix.

Table 7: Extension of the proposed single-trace non-profiled CPA to UOV, SNOVA, and QR-UOV.

Scheme	Sensitive Variable	Target Subroutine	Field
UOV [9]	Oil	`gf16v_mul_u32` `gf256v_mul_u32`	$\mathbb{F}_{16}$ or $\mathbb{F}_{256}$
SNOVA [37]	Vinegar	`gf_mat_mul_add`	$\mathbb{F}_{16}$
QR-UOV [17]	Vinegar	`VECTOR_MUL_SYMMETRIC_MATRIX_VxV`	$\mathbb{F}_{127}$

To resist these attacks (Table 7), the proposed countermeasure (Section 5) extends to UOV, SNOVA, and QR-UOV.

G Bit-sliced Multiply-Accumulate Routine

The routine `bitsliced_m_vec_mul_add` implements multiply accumulate of a vector **in** over $\mathbb{F}_{2^4}$ by a scalar $\mathbf{a} \in \mathbb{F}_{2^4}$ using a bitsliced representation [6]. The input vector is distributed across four 32-bit slices, where each slice stores one coefficient bit of multiple field elements in parallel. The scalar $\mathbf{a}$, written as $\mathbf{a} = a_0 + a_1x + a_2x^2 + a_3x^3$, is decomposed into its binary coefficients, and each a_k is expanded into a full-word mask (either all zeros or all ones). The multiplication is then realized by computing the contributions of $x^k \cdot \mathbf{in}$ through fixed XOR-based linear transformations corresponding to multiplication by x in the field, and conditionally accumulating these contributions into the output using the masks.

The routine can be found in the MAYO submission.[2] We present the pseudo-code of the subroutine in the List: 2.

```
void bitsliced_m_vec_mul_add(int m_legs, const uint32_t
      *in, unsigned char a, uint32_t *acc) {
    const uint32_t *in0 = in;
    const uint32_t *in1 = in + m_legs;
    const uint32_t *in2 = in + 2 * m_legs;
    const uint32_t *in3 = in + 3 * m_legs;

    uint32_t *acc0 = acc;
    uint32_t *acc1 = acc + m_legs;
    uint32_t *acc2 = acc + 2 * m_legs;
    uint32_t *acc3 = acc + 3 * m_legs;

    uint32_t aa = a;
    uint32_t a0 = -(aa & 1);
    uint32_t a1 = -((aa >> 1) & 1);
    uint32_t a2 = -((aa >> 2) & 1);
    uint32_t a3 = -((aa >> 3) & 1);

    uint32_t x, y, z;
    for (int i = 0; i < m_legs; i++) {
        acc0[i] ^= a0 & in0[i];
        acc1[i] ^= a0 & in1[i];
        acc2[i] ^= a0 & in2[i];
        acc3[i] ^= a0 & in3[i];

        x = in0[i] ^ in3[i];
        acc0[i] ^= a1 & in3[i];
        acc1[i] ^= a1 & x;
        acc2[i] ^= a1 & in1[i];
        acc3[i] ^= a1 & in2[i];

        y = in3[i] ^ in2[i];
        acc0[i] ^= a2 & in2[i];
        acc1[i] ^= a2 & y;
        acc2[i] ^= a2 & x;
        acc3[i] ^= a2 & in1[i];

        z = in2[i] ^ in1[i];
        acc0[i] ^= a3 & in1[i];
        acc1[i] ^= a3 & z;
        acc2[i] ^= a3 & y;
        acc3[i] ^= a3 & x;
    }
}
```

Listing 2: `bitsliced` multiplication and accumulation

H Public-Valued Masking Routine

Listing 3 illustrates the implementation of public-values masking technique, where `P1P1t_times_O` is applied twice, once with `P1_share1` and `P2_shares2` that corresponds to $\mathbf{P}^1_{i,0}$ and $\mathbf{P}^1_{i,1}$ respectively. For more details, refer to the work of Aulbach *et al.* [2].

```
// Shares of P1: P1_share1 and P1_share2
uint64_t P1_share1[P1_LIMBS_MAX], P1_share2[
     P1_LIMBS_MAX];
for(int i = 0; i < P1_LIMBS_MAX; i++){
    P1_share1[i] = ((uint64_t)rand() << 32) | rand();
    P1_share2[i] = P1_share1[i] ^ P1[i]; }

// First share of P2 is P2
uint64_t P2_share1[P2_LIMBS_MAX];
for(int i = 0; i < P2_LIMBS_MAX; i++)
    P2_share1[i] = P2[i];
// Second share of P2 is 0
uint64_t P2_share2[P2_LIMBS_MAX];
memset(P2_share2, 0, P2_LIMBS_MAX * sizeof(uint64_t));

// Masking case execution
P1P1t_times_O(P1_share1, sk.O, P2_share1);
P1P1t_times_O(P1_share2, sk.O, P2_share2);
```

Listing 3: Public-key masking.

[2] https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures

I Hyperparameter Search Space for DNN used.

Table 8 provides the hyperparameters search space of the multi-task DNN used in Section 4.

Table 8: Hyperparameter search space.

Number of Dense Layers, ℓ	1 to 8 in a step of 1
Neurons per layer	$10, 20, 50, 100, 200, 300, 400, 500$
Batch size	100 to 1000 in a step of 100
Activation function	*ReLU*, *SeLU*, *ELU*, *tanh*
Optimizer	Adam or RMSprop
Learning Rate	$1e-3, 5e-4, 1e-4, 5e-5, 1e-5$
Weight Initializer	Random Uniform or Glorot Uniform or He Uniform

J Result for Public-Value Masking under $\mathcal{A}_3$

Figure 10 illustrates the result for the attack using multi-task DNN in Section 4.3.

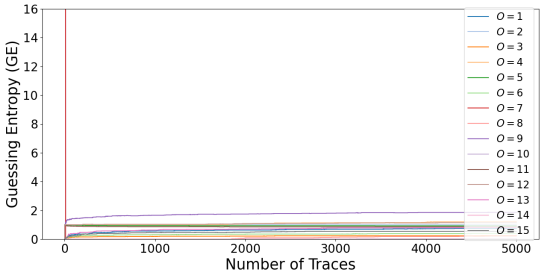

Figure 10: Guessing Entropy for each coefficient under adversary $\mathcal{A}_3$

References

[1] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, Quynh Dang, Thinh Hung Dang, John Kelsy, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. Status Report on the Second Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process. NIST Internal Report (IR) NIST IR 8610, National Institute of Standards and Technology, Gaithersburg, MD, 2026. doi:10.6028/NIST.IR.8610.

[2] Thomas Aulbach, Fabio Campos, and Juliane Krämer. Sok: On the physical security of uov-based signature schemes. In Ruben Niederhagen and Markku-Juhani O. Saarinen, editors, *Post-Quantum Cryptography*, pages 199–231, Cham, 2025. Springer Nature Switzerland.

[3] Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace side-channel assisted kipnis-shamir attack on UOV. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):221–245, 2023. doi:10.46586/TCHES.V2023.I3.221-245.

[4] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptogr. Eng.*, 10(2):163–188, 2020.

[5] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. *MAYO C implementation*, June 2023. Available at https://github.com/PQCMayo/MAYO-C. Accessed June, 2023.

[6] Ward Beullens, Fabio Campos, Sofia Celi, Basil Hess, and Matthias J. Kannwischer. Mayo specification document version 1.0, 2023. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/mayo-spec-web.pdf.

[7] Ward Beullens, Fabio Campos, Sofia Celi, Basil Hess, and Matthias J. Kannwischer. Mayo specification document version 2.0, 2025. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/mayo-spec-round2-web.pdf.

[8] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized Implementations for AVX2 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):252–275, Mar. 2024. URL: https://tches.iacr.org/index.php/TCHES/article/view/11427, doi:10.46586/tches.v2024.i2.252-275.

[9] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. Uov: Unbalanced oil and vinegar algorithm specifications and supporting documentation version 2.0, 2025. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/uov-spec-round2-web.pdf.

[10] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[12] Christophe Clavier and Léo Reynaud. Improved blind side-channel analysis by exploitation of joint distributions of leakages. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 24–44, Cham, 2017. Springer International Publishing.

[13] Christophe Clavier and Léo Reynaud. Improved Blind Side-Channel Analysis by Exploitation of Joint Distributions of Leakages. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 24–44, Cham, 2017. Springer. doi:10.1007/978-3-319-66787-4_2.

[14] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. https://eprint.iacr.org/2017/633. URL: https://eprint.iacr.org/2017/633.

[15] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU, 2020. https://falcon-sign.info/falcon.pdf.

[16] Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Tsuyoshi Takagi, Haruhisa Kosuge, Kimihiro Yamakoshi, Rika Akiyama, Satoshi Nakamura, Shingo Orihara, and Koha Kinjo. *The Round 2 Reference Implementation of QR-UOV*, December 2024. Available at https://github.com/qruov/round2.

[17] Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Tsuyoshi Takagi, Haruhisa Kosuge, Kimihiro Yamakoshi, Rika Akiyama, Satoshi Nakamura, Shingo Orihara, and Koha Kinjo. QR-UOV specification document, 2025. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/qruov-spec-round2-web.pdf.

[18] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 251–261, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[19] Julius Hermelink, Silvan Streit, Emanuele Strieder, and Katharina Thieme. Adapting belief propagation to counter shuffling of ntts. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):60–88, 2023. URL: https://doi.org/10.46586/tches.v2023.i1.60-88, doi:10.46586/TCHES.V2023.I1.60-88.

[20] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[21] Sönke Jendral and Elena Dubrova. Single-trace side-channel attacks on mayo exploiting leaky modular multiplication. In *Proceedings of the 2025 1st Workshop on Quantum-Resistant Cryptography and Security*, QRSEC '25, page 21–30, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3733820.3764682.

[22] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 206–222, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[23] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil and vinegar signature scheme. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 257–266, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[24] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[25] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[26] Suparna Kundu, Quinten Norga, Angshuman Karmakar, Uttam Kumar Ojha, Anindya Ganguly, and Ingrid Verbauwhede. mUOV: Masking the unbalanced oil and vinegar digital signature scheme at first- and higher-order. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 1994–2008, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3719027.3765188.

[27] Yanis Linge, Cécile Dumas, and Sophie Lambert-Lacroix. Using the joint distributions of a cryptographic function in side channel analysis. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 199–213, Cham, 2014. Springer International Publishing.

[28] NIST. FIPS 204 Module-Lattice-Based Digital Signature Standard. Online. Accessed 10th November, 2024, 2024. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf.

[29] NIST. FIPS 205 Stateless Hash-Based Digital Signature Standard. Online. Accessed 10th November, 2024, 2024. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf.

[30] Aesun Park, Kyung-Ah Shim, Namhun Koo, and Dong-Guk Han. Side-channel attacks on post-quantum signature schemes based on multivariate quadratic equations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):500–523, Aug. 2018. doi:10.13154/tches.v2018.i3.500-523.

[31] Duyên Pay and François-Xavier Standaert. Keep it simple: Refreshing the NTT of kyber's decapsulation to prevent plaintext-checking side-channel attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2026:472–499, 2026. URL: https://doi.org/10.46586/tches.v2026.i1.472-499, doi:10.46586/TCHES.V2026.I1.472-499.

[32] David Pokorný, Petr Socha, and Martin Novotný. Side-channel attack on rainbow post-quantum signature. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 565–568, Piscataway, NJ, USA, 2021. IEEE. doi:10.23919/DATE51398.2021.9474157.

[33] Pierre Pébereau. One vector to rule them all: Key recovery from one vector in UOV schemes. Cryptology ePrint Archive, Paper 2023/1131, 2023. URL: https://eprint.iacr.org/2023/1131.

[34] Azade Rezaeezade, Trevor Yap, Dirmanto Jap, Shivam Bhasin, and Stjepan Picek. Breaking the blindfold: deep learning-based blind side-channel analysis. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[35] François-Xavier Standaert, Tal Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-01001-9_26.

[36] Lih-Chung Wang, Chun-Yen Chou, Jintai Ding, Yen-Liang Kuan, Jan Adriaan Leegwater, Ming-Siou Li, Bo-Shu Tseng, Po-En Tseng, and Chia-Chun Wang. *SNOVA Reference, Optimized, and AVX2 implementations*, August 2025. Available at https://github.com/PQCLAB-SNOVA/SNOVA.

[37] Lih-Chung Wang, Chun-Yen Chou, Jintai Ding, Yen-Liang Kuan, Jan Adriaan Leegwater, Ming-Siou Li, Bo-Shu Tseng, Po-En Tseng, and Chia-Chun Wang. Snova specification document version 2.0, 2025. URL: https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/snova-spec-round2-web.pdf.

[38] Yanis Linge and Cécile Dumas and Sophie Lambert-Lacroix. Using the Joint Distributions of a Cryptographic Function in Side Channel Analysis. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 199–213, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10175-0_14.