DPA-style Attacks on VOLE-in-the-Head Signature Schemes

Jinnuo Li[1], Shu Han[1], Yunyan Zou[1], Fan Zhang[3], Puwen Wei[1,2], Weijia Wang[1,2(✉)] and Hao Cheng[1,2(✉)]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
jinnuo.li@mail.sdu.edu.cn
[2] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China
{pwei,wjwang,hao.cheng}@sdu.edu.cn
[3] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
fanzhang@zju.edu.cn

Abstract. VOLE-in-the-Head (VOLEitH) has emerged as a promising paradigm for the design of post-quantum signatures, as it enables efficient signing and verification while also achieving small signature sizes. In the ongoing NIST PQC standardization process for additional digital signatures, two VOLEitH signature schemes, namely FAEST and SDitH, have advanced to the third round. Regarding their practical security, especially resistance to power side-channel attacks, existing studies have focused on only a few specific operations, and thus a comprehensive analysis remains lacking. To address this gap, we present, to the best of our knowledge, the first DPA-style attack framework against VOLEitH signatures. Our framework targets the secret witness, whose recovery permits the generation of valid signatures for arbitrary messages. Based on this framework, we systematically analyze the leakage stemming from the computation of sensitive intermediate values involved in VOLE correlations, and characterize how such leakage can be linked to secret witness hypotheses for mounting DPA-style attacks. As a consequence, we propose six classes of DPA-style attacks on FAEST and SDitH, including two vertical and four horizontal attacks. Finally, the proposed attacks are validated using both simulated leakage under varying noise conditions and practical measurements on three ARM Cortex-M4 devices, demonstrating their effectiveness in both settings.

Keywords: Side-channel attacks, Differential power analysis, VOLE-in-the-Head, Post-quantum cryptography

1 Introduction

Although NIST has now selected five primary algorithms in its PQC standardization process, namely ML-KEM [FIP24b], ML-DSA [FIP24a], SLH-DSA [FIP24c], FALCON [FHK+20] (to be renamed FN-DSA), and HQC [GAA+25], it is still seeking to diversify its portfolio of signature schemes. To this end, NIST posted a call for additional PQ digital signature proposals in 2022 [Nat22]. In October 2024, it announced 14 candidates advancing to the second round [Nat24]. Among them, FAEST [BBB+25], SDitH [ABB+25b], and PERK [ABB+25a] are based on the Vector Oblivious Linear Evaluation-in-the-Head (VOLE-in-the-Head, or simply VOLEitH) technique; moreover, Mirath [AAB+25] and RYDE [ABB+25c] also include VOLEitH-based variants. More recently, in May 2026, both FAEST and SDitH advanced to the third round, where only nine candidates were selected [Nat26].

VOLEitH signature schemes. VOLEitH-based schemes derive signatures from a Fiat-Shamir-transformed zero-knowledge proof, where the signer first commits to VOLE correlations through batched all-but-one seed commitments and subsequently proves that the committed witness satisfies the target low-degree algebraic constraints. A significant advantage is that VOLE provides linearly homomorphic commitments and supports opening over a large challenge space, thereby enabling more compact and efficient proofs than classical MPC-in-the-Head-style constructions. This advantage makes VOLEitH a promising paradigm for designing PQ signatures, and the recent advancement of FAEST and SDitH further underscores the growing importance of the VOLEitH paradigm, motivating a systematic study of its security in both theory and practice.

Side-channel attacks. Side-Channel Attacks (SCAs), first introduced by Kocher [Koc96], have since become a crucial component in evaluating the real-world security of cryptographic implementations. Among various SCAs, power analysis attacks are particularly well-studied, due to their effectiveness and practicality, and they are commonly divided into Simple Power Analysis (SPA) [MDS99] and Differential Power Analysis (DPA) [KJJ99]. In this paper, we focus on the latter, as DPA can reliably leverage weak leakage and offer much stronger resilience against noise, misalignment, and implementation variability.

Related work. Recently, side-channel analysis of the VOLEitH paradigm began to receive more attention. Jendral and Dubrova [JD26b] present the first side-channel attacks on VOLEitH signature schemes, demonstrating two single-trace attacks against a masked implementation of FAEST. One attack exploits the leakage of witness bits during the byte combine operation, while the other leverages the leakage of VOLE tag bits during the VOLE transpose procedure. In their follow-up work [JD26a], they introduce the first fault-injection attacks on VOLEitH signatures, targeting the second-round reference implementations of FAEST and SDitH. In addition, Godard et al. [GAG+26] present side-channel attacks on the second-round versions of both FAEST and SDitH, recovering the witness from the bit-level leakage of the VOLE tags $\mathbf{V}$ in the field-multiplication function via a Belief Propagation (BP)-based Soft Analytical Side-Channel Attack (SASCA) in a non-supervised scenario.

Motivation. Based on the above, the existing attacks against VOLEitH signature schemes exhibit three limitations: 1) they exploit only the VOLE tags $\mathbf{V}$, overlooking the equally important VOLE-correlated intermediate $\mathbf{u}$; 2) they exploit the leakage only at the bit level, leaving unexplored how the intermediates leak under different leakage characteristics; and 3) they focus on only a few specific operations, without identifying the common VOLEitH routines in which such exploitable leakage repeatedly appears across different schemes. To address these limitations, we aim to provide a *comprehensive* understanding of DPA vulnerabilities in VOLEitH signature schemes.

Contributions. We focus on the two VOLEitH candidates in the third round of the NIST additional PQ signature competition, i.e., FAEST and SDitH, and our contributions are as follows.

- To enable a comprehensive analysis, we present a DPA-style attack framework against VOLEitH-based signature schemes, which provides a unified approach to identifying and exploiting side-channel leakage from intermediate computations in the signing procedure. The framework is able to systematically clarify the relationships between the secret key/witness and the different leaked intermediates, under different leakage characteristics.

- Based on our framework, we identify six classes of DPA-style attacks (including two vertical and four horizontal) for the second-round reference implementations of FAEST/SDitH, and demonstrate each class with a concrete instance. Our attacks cover different VOLE-correlated intermediates (i.e., $\mathbf{u}$ and $\mathbf{V}$) and different leakage characteristics (i.e., bit and Hamming weight), and each attack instance enables full recovery of the secret witness. A summary of the various presented attacks indicates that the exploitable leakage repeatedly appears in shared VOLEitH routines, showing that it originates from computational structures common to VOLEitH signature schemes.
- We validate the proposed attacks through both simulations and practical measurements. Simulations under varying classification accuracies and noise levels evaluate the effectiveness and robustness of all six attack instances. We further collect power traces from three ARM Cortex-M4 devices to train Convolutional Neural Network (CNN)-based leakage classifiers, assess their performance in both same-device and cross-device settings, and use their predictions in practical attack evaluations. The results demonstrate that the identified leakage can be reliably exploited in practice and validate the practical feasibility of the proposed attacks.

Organization. The remainder of this paper is organized as follows. Section 2 introduces the DPA-style attacks and provides the necessary background on VOLEitH signature schemes. In Section 3, we present our DPA-style attack framework, as well as various attacks, for FAEST and SDitH. Section 4 describes the experimental validation of the proposed attacks. Section 5 reports the changes introduced in Version 3 of FAEST and SDitH and discusses their implications for the proposed framework. Finally, we conclude the paper in Section 6.

2 Background

In this Section, we recall DPA-style attacks, introduce the notation, and review the VOLEitH framework, highlighting the main components that our attacks exploit.

2.1 DPA-style attacks

DPA-style attacks [MOP07, Chapter 6] form a well-established class of power side-channel attacks, whose generic framework is illustrated in Figure 1. They typically follow a "divide-and-conquer" strategy: the full key k is split (e.g., byte-wise or bit-wise) into smaller *partial* keys k_i, which are attacked and recovered independently. Each partial key ranges over a search space that is small enough to be enumerated exhaustively, and thus the attack reduces to distinguishing the correct value of k_i from all of its candidates.

The basis of this distinction is the relation between an intermediate value processed by the device and the corresponding physical leakage. To be more specific, let $\boldsymbol{x} = [x_0, \ldots, x_{N-1}]$ be the known inputs, where N is the number of measurements and x_j is the input used in the j-th measurement. For a candidate $\bar{k}_i$ of the partial key k_i and an input x_j, the attack first computes the target intermediate value $z_j^{\bar{k}_i} = F(\bar{k}_i, x_j)$. Here, F denotes the selected target computation, and $z_j^{\bar{k}_i}$ denotes the value used for leakage prediction or comparison in the attack. Depending on the targeted leakage characteristic, this value can be a byte value, the Hamming weight of a byte, or a single bit.

Let L be the leakage function. For the correct partial key k_i, the measured trace can be abstracted as $t_j = L(z_j^{k_i}) + \varepsilon_j$, where ε_j denotes probabilistic noise and other unrelated variations. Let M be the *power model* that approximates the leakage function L, so that $t_j \approx M(z_j^{k_i})$ up to noise. A typical DPA-style attack can be described as follows:

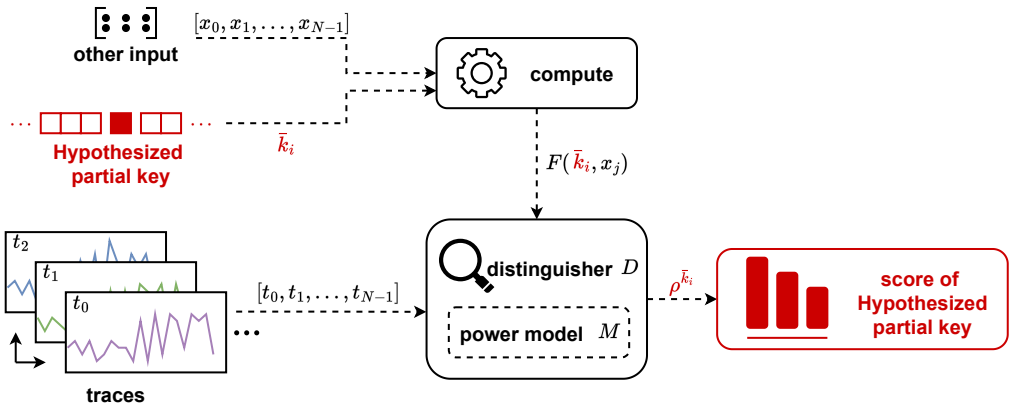

Figure 1: Framework of DPA-style attacks.

1. Make a hypothesized partial key $\bar{k}_i$ and compute the corresponding target intermediate values $\boldsymbol{z}^{\bar{k}_i} = F(\bar{k}_i, \boldsymbol{x})$ across the known inputs $\boldsymbol{x}$.

2. Apply the power model M to obtain the predicted leakage $M(\boldsymbol{z}^{\bar{k}_i})$.

3. The distinguisher D compares the predicted leakage with the measured trace and computes a score $\rho^{\bar{k}_i}$ for the candidate:

$$\rho^{\bar{k}_i} = D\left(\boldsymbol{t}, M(\boldsymbol{z}^{\bar{k}_i})\right).$$

4. Repeat the above procedure for all candidates $\bar{k}_i$. The candidate with the highest score is selected as the recovered partial key:

$$\tilde{k}_i = \arg\max_{\bar{k}_i} D\left(\boldsymbol{t}, M(\boldsymbol{z}^{\bar{k}_i})\right).$$

Applying the same scoring procedure to each partial key k_i reconstructs the full key k.

In the non-profiled setting, the power model M is assumed a priori, with common choices including the Hamming weight model. In contrast, a profiled attack learns a power model from a controlled device with known intermediate values. Profiling is optional, but it usually reduces the number of required traces. Such profiled models are commonly formulated as generative or discriminative models. The generative approach [SLP05] models the leakage conditional on an intermediate value, synthesizes the estimated leakage, and compares it with the measured leakage. The discriminative approach instead predicts a target intermediate value from the measured trace [CRR02]. Its attack flow slightly differs from the procedure described above. During the attack, the trained model C is applied to the relevant trace segments to produce a one-dimensional proxy trace $C(\boldsymbol{t})$. The distinguisher then compares this proxy trace with the corresponding candidate values $\boldsymbol{z}^{\bar{k}_i}$ and computes a score for each candidate. The resulting scores are ranked to recover the partial key. With models such as CNNs, this formulation also helps reduce the need for manual alignment and exploit leakage across multiple power samples.

DPA-style attacks fall into two orthogonal categories [CFG+10], namely *vertical* and *horizontal* attacks. The former exploits the leakage of an intermediate value across many executions with different inputs, while the latter instead exploits multiple intermediate values within a single execution. In this sense, the latter can be regarded as an extension of the former, and typically requires far fewer traces. SPA [MOP07, Chapter 5] is then

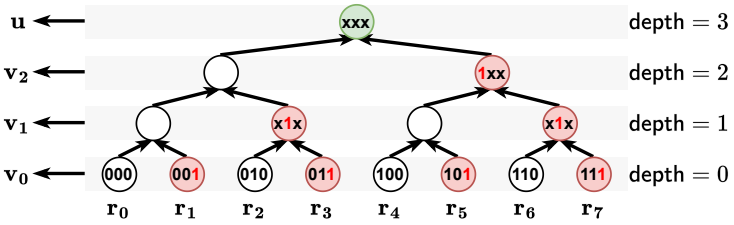

Figure 2: A simplified illustration for the generation of $\mathbf{u}$ and $\mathbf{v}$ with $\mathsf{depth} = 3$.

a degenerate vertical attack, in which the key is read from a single trace without any statistical aggregation over multiple measurements.

2.2 Notation

In the remainder of this paper, elements of a finite field are denoted by lowercase letters, such as x. Vectors are denoted by lowercase bold letters, such as $\boldsymbol{x}$, and matrices by uppercase bold letters, such as $\boldsymbol{X}$. Subscripts index members of a collection; e.g., $\boldsymbol{x}_i$ denotes the i-th vector in $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots\}$. For a vector $\boldsymbol{x}$, $\boldsymbol{x}[i]$ denotes its i-th element, and $\boldsymbol{x}[i\!:\!j]$ the contiguous subvector whose element indices lie in $[i, j]$.

In particular, when binary representations are relevant, we use a sans-serif lowercase letter x to denote an element of $\mathbb{F}_2$, which can be viewed as a single bit. An element of $\mathbb{F}_{2^m}$ is denoted by a lowercase bold sans-serif letter, such as $\mathbf{x}$, and can be viewed as an m-bit vector through its representation over $\mathbb{F}_2$. Vectors and matrices over such representations are likewise denoted in bold sans-serif: $\mathbf{x}$ is a bit vector, and $\mathbf{X} \in \mathbb{F}_2^{n\times m}$ is a bit matrix, whose i-th row is $\mathbf{X}[i] \in \mathbb{F}_2^m$ and whose entry in row i and column j is $\mathbf{X}[i][j]$. We also use $\mathbf{X}^T[i]$ to denote the i-th column of $\mathbf{X}$. As several of our attacks operate at the byte level, $\mathbf{x}^{(i)}$ denotes the i-th byte of a bit vector $\mathbf{x}$, that is, the 8-bit block $\mathbf{x}[8i\!:\!8i{+}7]$. We use $\cdot$ to denote finite-field multiplication, extended componentwise to vectors. Since these fields have characteristic 2, addition and subtraction coincide, and we use $+$, $-$, and $\oplus$ interchangeably.

2.3 VOLE-in-the-Head framework

A VOLE correlation over the finite field $\mathbb{F}_{2^\lambda}$ of length ℓ consists of secret random bits $\mathsf{u}_i \in \mathbb{F}_2$, secret VOLE tags $\mathbf{v}_i \in \mathbb{F}_{2^\lambda}$, a global key $\boldsymbol{\Delta} \in \mathbb{F}_{2^\lambda}$, and local VOLE keys $\mathbf{q}_i \in \mathbb{F}_{2^\lambda}$, for $i \in [0, \ell)$, such that

$$\mathbf{q}_i = \mathsf{u}_i \cdot \boldsymbol{\Delta} + \mathbf{v}_i, \tag{1}$$

where $\mathsf{u}_i \cdot \boldsymbol{\Delta}$ equals $\boldsymbol{\Delta}$ if $\mathsf{u}_i = 1$ and $\mathbf{0}$ otherwise. The prover knows u_i and $\mathbf{v}_i$, while the verifier is given $\mathbf{q}_i$ and $\boldsymbol{\Delta}$.

VOLEitH is a technique for realizing such correlations inside publicly verifiable zero-knowledge proof systems such as FAEST. The prover first commits to u_i and $\mathbf{v}_i$ through a VOLE commitment. The verifier then sends a random challenge $\boldsymbol{\Delta}$, and the prover replies with an opening that reveals exactly the local keys $\mathbf{q}_i$ satisfying Equation 1, and nothing more. In the non-interactive signature setting, $\boldsymbol{\Delta}$ is derived through the Fiat-Shamir transform rather than sent by an external verifier.

For efficiency, VOLEitH does not generate correlations over $\mathbb{F}_{2^\lambda}$ directly. Instead, it generates several small VOLE instances over smaller fields and combines them into one large correlation. We first describe a single small VOLE instance over $\mathbb{F}_{2^k}$, where k is the bit-length of the small field, and then explain how τ such instances are combined.

An all-but-one vector commitment of length $N = 2^k$ can be used to generate a VOLE commitment in $\mathbb{F}_{2^k}$. The construction begins with a root seed, which is expanded via a

pseudorandom generator to form a complete binary tree, commonly referred to as a GGM tree. The leaves of this tree serve as pseudorandom seeds $\mathbf{sd}_i$ for commitment. A signer holding the N seeds $\mathbf{sd}_i$ can further expand them into N strings $\mathbf{r}_i$.

VOLE conversion. These $N = 2^k$ strings $\mathbf{r}_i \in \mathbb{F}_2^\ell$, indexed by $i \in [0, N)$, are then combined into the VOLE-correlated values $\mathbf{u}$ and $\mathbf{v}$ of the small VOLE instance over $\mathbb{F}_{2^k}$, which we call the *VOLE conversion*:

$$\mathbf{u} = \sum_{i=0}^{N-1} \mathbf{r}_i, \qquad \mathbf{v} = \sum_{i=0}^{N-1} \mathbf{i} \cdot \mathbf{r}_i.$$

Here, $\mathbf{u} \in \mathbb{F}_2^\ell$ is the XOR of all strings $\mathbf{r}_i$. When computing $\mathbf{v}$, each index i is encoded as an element $\mathbf{i} \in \mathbb{F}_{2^k}$. The value $\mathbf{v}$ is represented in bit-sliced form by k binary vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{k-1} \in \mathbb{F}_2^\ell$. For each $j \in [0, k)$, $\mathbf{v}_j$ is the XOR of those strings $\mathbf{r}_i$ for which the j-th bit $\mathbf{i}[j]$ of $\mathbf{i}$ is equal to 1. Figure 2 illustrates a simplified example with $k = 3$.

To obtain the VOLE correlation from the above equation, consider a verifier who later learns all seeds $\mathbf{sd}_i$ except the unopened seed $\mathbf{sd}_\Delta$, for some index $\Delta \in [0, N)$ (viewed as an $\mathbb{F}_{2^k}$ element), which is a random global key playing the role of the small-field global key in this instance. The verifier computes:

$$\mathbf{q} = \sum_{i=0}^{N-1} (\mathbf{\Delta} - \mathbf{i}) \cdot \mathbf{r}_i = \mathbf{\Delta} \cdot \sum_{i=0}^{N-1} \mathbf{r}_i - \sum_{i=0}^{N-1} \mathbf{i} \cdot \mathbf{r}_i = \mathbf{\Delta} \cdot \mathbf{u} - \mathbf{v}.$$

In this way, the verifier reconstructs the VOLE correlation over the finite field $\mathbb{F}_{2^k}$.

Witness embedding. To build a proof of knowledge for a chosen witness $\mathbf{w} \in \mathbb{F}_2^\ell$ rather than for the random bits $\mathbf{u}$, the prover computes the masked witness $\mathbf{d} \stackrel{\text{def}}{=} \mathbf{w} - \mathbf{u}$ and sends it to the verifier, who then updates its VOLE outputs to obtain

$$\mathbf{q'} \stackrel{\text{def}}{=} \mathbf{q} + \mathbf{\Delta} \cdot \mathbf{d} = \mathbf{\Delta} \cdot (\mathbf{u} + \mathbf{d}) - \mathbf{v} = \mathbf{\Delta} \cdot \mathbf{w} - \mathbf{v}.$$

Thus, $(\mathbf{w}, \mathbf{v})$ held by the prover and $(\mathbf{\Delta}, \mathbf{q'})$ held by the verifier form a valid VOLE correlation for the witness $\mathbf{w}$.

Combining small VOLEs. To build the full correlation, the signer runs the vector commitment for each of the τ small instances, obtaining τ pairs $(\mathbf{u}_i, \mathbf{v}_i)$. For FAEST, the small fields are $\mathbb{F}_{2^{k_0}}$ and $\mathbb{F}_{2^{k_1}}$, with τ_0 instances over the former and τ_1 over the latter. We now move from the vector view to a matrix view: by decomposing each entry of the tag vector $\mathbf{v}_i$ into its k bits, we represent it as a binary submatrix $\mathbf{V}_i \in \mathbb{F}_2^{\ell \times k}$ stored column-wise. Throughout, a single subscript on $\mathbf{V}$ or $\mathbf{Q}$ denotes such a per-instance submatrix rather than a row.

Combining the instances requires them to share the same random bits. The signer therefore adopts the secret bits of the first instance, $\mathbf{u}_0$, as the global secret $\mathbf{u}$, computes the correction values $\mathbf{c}_i = \mathbf{u}_i - \mathbf{u}_0$, and sends them publicly to the verifier, so that every instance can be expressed with respect to this unified $\mathbf{u}$. For each small VOLE, after opening and applying the correction values, the verifier learns $(\mathbf{\Delta}_i, \mathbf{Q}_i)$ satisfying

$$\mathbf{Q}_i = \big(\mathbf{\Delta}_i[0] \cdot \mathbf{u}_i \;||\; \cdots \;||\; \mathbf{\Delta}_i[k-1] \cdot \mathbf{u}_i\big) - \mathbf{V}_i,$$

where $\mathbf{\Delta}_i[j] \in \mathbb{F}_2$ denotes the j-th bit in the fixed binary representation of $\mathbf{\Delta}_i \in \mathbb{F}_{2^k}$, and $||$ denotes the horizontal concatenation of its two operands, which may be vectors or matrices. The columns of $\mathbf{Q}_i$ thus stack the k selected copies of $\mathbf{u}$ side by side. The

per-instance submatrices are then concatenated horizontally, using the same operator, into the final $\ell \times \lambda$ matrices

$$\mathbf{V} = (\mathbf{V}_0 \,||\, \cdots \,||\, \mathbf{V}_{\tau-1}), \qquad \mathbf{Q} = (\mathbf{Q}_0 \,||\, \cdots \,||\, \mathbf{Q}_{\tau-1}),$$

whose column count is λ. Without grinding, the τ small VOLEs supply all $\lambda = \tau k$ columns. In FAEST v2 with grinding, they supply only $\lambda - w_{\text{grind}}$ of them; the remaining w_{grind} columns are zero, corresponding to the upper bits of $\boldsymbol{\Delta}$ that grinding fixes to zero. The corresponding positions are represented as zero-padding columns in the binary VOLE matrices.

Viewing each row $\mathbf{V}[i]$ or $\mathbf{Q}[i]$ as an element of $\mathbb{F}_{2^\lambda}$, the two matrices form a VOLE correlation whose global key $\boldsymbol{\Delta}$ is assembled from the bits of the per-instance $\boldsymbol{\Delta}_i$. After the witness embedding, this correlation satisfies the row-wise relation

$$\mathbf{Q}[i] = \mathbf{V}[i] + \mathbf{w}[i] \cdot \boldsymbol{\Delta}$$

for each i.

VOLE transpose. The final tag matrix $\mathbf{V}$ is produced column-wise by the small VOLE instances. The VOLE transpose is an implementation step that reorders its storage into a row-wise layout. This reordering does not change the underlying matrix; it only makes each row $\mathbf{V}[i]$ contiguous in memory, so that it can be read directly as one element of $\mathbb{F}_{2^\lambda}$.

VOLE consistency check. The consistency check prevents the signer from cheating on the correction values $\mathbf{c}_i$, ensuring that $\mathbf{u}_0$ is committed as the global $\mathbf{u}$ and that the VOLE correlation reconstructed by the verifier remains valid. To this end, a compressing matrix $\mathbf{H}$ defines a linear hash function. Conceptually, the signer computes the compressed values

$$\mathbf{u'} = \mathbf{Hu}, \quad \mathbf{V'} = \mathbf{HV},$$

and sends them to the verifier. Using the public correction values $\mathbf{c}_i$, the verifier reconstructs $\mathbf{Q}$, computes $\mathbf{Q'} = \mathbf{HQ}$ and checks that the VOLE relation still holds between $\mathbf{u'}$, $\mathbf{V'}$ and $\mathbf{Q'}$.

In concrete instantiations, however, these compressed values are still too large to be transmitted explicitly. Instead of sending $\mathbf{u'}$ and $\mathbf{V'}$ in full, the signer includes only a hash digest derived from them, or equivalently binds this digest into the Fiat-Shamir transcript. On the verifier's side, the compressed values are recomputed from the opened VOLE data, subjected to the same compression and hashing procedure, and the check succeeds if the resulting digest, or the transcript value derived from it, matches the one encoded in the signature.

3 Attacks

3.1 Attacker model

We consider an attacker who has physical access to the signing device to acquire side-channel information during the execution of Sign(), and can obtain the signatures it produces, from which the public values used in our attacks are derived by running the verification procedure. We also assume that the keys (pk, sk) are static, and that the attacker possesses fully controllable profiling devices that are similar to the target device. In the profiling phase, the attacker executes Sign() on a profiling device with known intermediate values, in order to create models of the targeted operations. The profiling and attack procedures are performed on different physical devices. Both phases assume

Table 1: Main parameter sets of FAEST v2.

Schemes	λ	ℓ	w_{grind}	τ	τ_0	τ_1	k_0	k_1	$\hat{\ell}$
`FAEST-128f`	128	1280	8	16	8	8	7	8	1680
`FAEST-192f`	192	2496	8	24	8	16	7	8	3088
`FAEST-256f`	256	3104	8	32	8	24	7	8	3888

Table 2: Main parameter sets of SDitH v2.

Schemes	λ	$\lvert\mathsf{wit}\rvert_2$	w	τ	κ	w_{pow}	B
`SDitH2-L1-gf2-fast`	128	616	56	16	8	2	16
`SDitH2-L3-gf2-fast`	192	880	73	24	8	2	16
`SDitH2-L5-gf2-fast`	256	1144	104	32	8	2	16

that the attacker is able to isolate the sequence of target operations within the execution of Sign(), which can be achieved using, for example, pattern matching techniques or more advanced time-series analysis, but this part is not studied here.

The goal of the attacker is to recover the secret witness $\mathbf{w}$, since knowing $\mathbf{w}$ suffices to reproduce the associated zero-knowledge proof and thus to forge signatures on arbitrary messages. For FAEST, it is not necessary to recover the complete witness, because its first part is the private key. For SDitH, however, recovery of the complete witness is needed, as it is a pseudorandom value generated from a random seed.

3.2 Attack methodology

Given the generic framework of DPA-style attacks in Figure 1, we now instantiate it for VOLEitH signature schemes. Since recovering $\mathbf{w}$ is enough for the attacker, in our attacks we use the partial-witness hypothesis $\bar{\mathbf{w}}$ instead of the generic key hypothesis $\bar{k}$.

To apply a DPA-style distinguisher to such hypotheses, the witness hypothesis $\bar{\mathbf{w}}$ must be linked to intermediate values that are actually processed during signing, and this link is established by the following VOLEitH linear relations:

$$\mathbf{w}[i] = \mathbf{d}[i] \oplus \mathbf{u}[i] \tag{2}$$

$$\mathbf{Q}[i] = \mathbf{V}[i] \oplus \mathbf{w}[i] \cdot \boldsymbol{\Delta} \tag{3}$$

Here, $\mathbf{d}$, $\mathbf{Q}$ and $\boldsymbol{\Delta}$ can be reconstructed during verification, where $\mathbf{Q}$ denotes the verifier-side VOLE key matrix after the public witness-embedding correction. Based on these relations, for each witness hypothesis $\bar{\mathbf{w}}$, we can derive a corresponding hypothetical value of either $\mathbf{u}$ or $\mathbf{V}$, the two *intermediate variables* we target. The specific part of such an intermediate variable, which the device processes and whose value is predicted from $\bar{\mathbf{w}}$, is called the *Sensitive Intermediate Value (SIV)* exploited by the attack. We consider two leakage characteristics of the SIV, namely Hamming weight leakage and bit leakage. The latter, also known as the determiner model [RBSC22], arises when the secret is processed *bit by bit*. Therefore, each step leaks a single bit, and a single observation suffices to reveal whether that bit is 0 or 1.

Applying our framework to FAEST and SDitH, we systematically analyze the procedures where $\mathbf{u}$ or $\mathbf{V}$ are computed, accessed, or processed. We identify *six* attack classes and demonstrate each with a concrete attack instance. In these instances, we use Pearson correlation for Hamming weight leakage and matching scores for bit leakage; under hard decisions, the latter is equivalent to majority voting. In the following description, we take `FAEST-128f` and `SDitH2-L1-gf2-fast` as representative parameter sets, and their main parameters are summarized in Table 1 and Table 2, respectively.

```
1  typedef struct {
2    uint64_t values[2];
3  } bf128_t;
4
5  #define BF_VALUE(v, i) ((v).values[i])
6
7  uint64_t bf128_bit_to_uint64_mask(bf128_t value, unsigned int bit) {
8    const unsigned int byte_idx = bit / 64;
9    const unsigned int bit_idx  = bit % 64;
10   return -((BF_VALUE(value, byte_idx) » bit_idx) & 1);
11 }
12
13 bf128_t bf128_mul(bf128_t lhs, bf128_t rhs) {
14   bf128_t result = bf128_and_64(lhs, bf128_bit_to_uint64_mask(rhs, 0));
15   for (unsigned int idx = 1; idx != 128; ++idx) {
16     const uint64_t mask = bf128_bit_to_uint64_mask(lhs, 128 - 1);
17     lhs = bf128_shift_left_1(lhs);
18     BF_VALUE(lhs, 0) ^= (mask & bf128_modulus);
19     result = bf128_add(result, bf128_and_64(lhs, bf128_bit_to_uint64_mask(rhs, idx)));
20   }
21   return result;
22 }
```

Figure 3: The C code of `bf128_mul` and `bf128_bit_to_uint64_mask`.

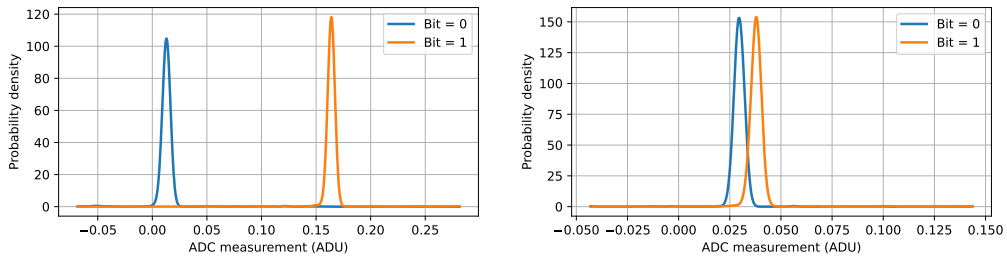

Figure 4: Distributions of power consumption (average over 1000 traces) for bit 0 and bit 1 in `bf128_mul` (left) and `bitvec_get_value` (right).

3.3 Vertical attacks

In vertical attacks, the witness is recovered through statistical analysis over multiple traces. Within our framework, we identify two attack classes in this category, both targeting the VOLE random bits **u** but under different leakage characteristics.

3.3.1 Class 1: vertical attack using bit leakage of u

Leakage analysis. Since **u** is a critical intermediate variable, analyzing the operations involving it is of central importance. We note that the VOLEHash computation of FAEST.Sign, which realizes the VOLE consistency check, involves **u**, and we are particularly interested in the underlying finite-field multiplications used in this routine. Specifically, Figure 3 shows the code snippet of $\mathbb{F}_{2^\lambda}$ multiplication, i.e., `bf128_mul`, in the FAEST reference implementation. Per lines 14 and 19, the operand `rhs` is processed bit by bit under the control of `idx`, and for each bit, the subfunction `bf128_bit_to_uint64_mask` (lines 7–11) is invoked to compute a 64-bit value `mask`.

Focusing on line 10, we observe that the generated `mask` is, in binary, either all 1s or all 0s, depending on the processed bit of `rhs`. Since `mask` can only take these two values, the large difference in their Hamming weights induces clearly distinguishable power consumption, as shown in Figure 4 (left). This leakage can therefore be used to predict the value of the associated bit directly. Since `bf128_mul` is called repeatedly with successive λ-bit chunks of **u** as the operand `rhs`, the processed bits of **u** act as bit-level SIVs and leak through this operation. Through Equation 2, these SIVs can be linked to witness-bit

Algorithm 1 Attack instance 1.

Input: N signatures $\{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$; N corresponding traces $\{t_0, t_1, \ldots, t_{N-1}\}$;
Input: profiled classifier C; distinguisher D.
Output: sk.

for $j \in [0, N)$ **do**
  $\mathbf{d}_j \leftarrow \text{FAEST.Verify}(\sigma_j, \ldots)$ ▷ Parse σ_j
  $\widehat{\mathbf{u}}_j \leftarrow C(\text{Preprocess}(t_j))$ ▷ Predict the bits of $\mathbf{u}_j$
end for
for $i \in [0, \lambda)$ **do** ▷ Bit-level divide-and-conquer
  $\widetilde{\mathbf{w}}[i] \leftarrow \underset{\bar{\mathbf{w}}[i] \in \{0,1\}}{\arg\max}\ D\left(\widehat{\mathbf{u}}_j[i], \mathbf{d}_j[i] \oplus \bar{\mathbf{w}}[i]\right)_{j=0}^{N-1}$
end for
$sk \leftarrow \widetilde{\mathbf{w}}[0 : \lambda - 1]$
return sk

hypotheses and scored over multiple executions. We refer to this setting as class 1, a vertical DPA-style attack based on the bit leakage of $\mathbf{u}$.

A specific attack instance. Building on this analysis, we present a concrete instance of this class, as detailed in Algorithm 1. Since the leakage directly exposes the processed bits of $\mathbf{u}$, we adopt a bit-level divide-and-conquer strategy and define the hypothesis space for the i-th witness bit as $\bar{\mathbf{w}}[i] \in \{0, 1\}$.

The attack proceeds by observing N independent executions of FAEST.Sign. For the j-th execution, we record the public signature σ_j and collect the corresponding power trace t_j. By running the verification procedure on σ_j, we derive the public value $\mathbf{d}_j$. According to Equation 2, each candidate hypothesis $\bar{\mathbf{w}}[i]$ yields the hypothetical bit

$$\bar{\mathbf{u}}_j[i] = \mathbf{d}_j[i] \oplus \bar{\mathbf{w}}[i].$$

The discriminative power model of Section 2 is instantiated here as a profiled classifier C, which predicts the processed bit of $\mathbf{u}$ from the corresponding trace segment and outputs the predicted bit $\widehat{\mathbf{u}}_j[i]$. The distinguisher compares the predicted bits with the hypothetical bits over all N executions, and the hypothesis with the highest score is selected as the recovered witness bit. Iterating this procedure for all $i \in [0, \lambda)$ reconstructs the witness vector $\widetilde{\mathbf{w}}[0 : \lambda - 1]$, which directly corresponds to the secret key sk of FAEST.

3.3.2 Class 2: vertical attack using HW leakage of u

Leakage analysis. We next consider the same intermediate variable $\mathbf{u}$ under byte-wise Hamming weight leakage rather than bit leakage, a setting we refer to as class 2. To this end, we analyze the VOLE conversion routine introduced in Section 2.3, another procedure that processes $\mathbf{u}$. As illustrated in Figure 2, this routine expands the random seed and converts it into the VOLE-correlated values $\mathbf{u}$ and $\mathbf{V}$ through a series of XOR operations. In the reference implementation of FAEST, this is realized via byte-wise XOR operations, and the relevant code snippet is shown in Figure 5. In line 11, `xor_u8_array` updates `V(j)`, which corresponds to a column of the matrix $\mathbf{V}$, by XORing it with `R(j, 2 * idx + 1)`. In line 12, another byte-wise XOR combines `R(j, 2 * idx)` and `R(j, 2 * idx + 1)` to produce the next-level value `R(j + 1, idx)`. After the iterative conversion completes, the final value `R(depth, 0)` is copied to the output buffer `u` in line 16. Therefore, the byte-wise XOR operations in this routine directly manipulate the intermediate values that form $\mathbf{V}$ and $\mathbf{u}$.

Under the Hamming weight model, the memory write performed by each byte-wise XOR operation may leak the Hamming weight of the corresponding output byte. Consequently,

```c
unsigned int convert_to_vole(const uint8_t* iv, const uint8_t* sd, bool sd0_bot,
                             unsigned int i, unsigned int outlen, uint8_t* u,
                             uint8_t* v, const faest_paramset_t* params) {
  /* ... Processing ... */
#define R(row, column) (r + (((row) % 2) * num_instances + (column)) * outlen)
#define V(idx) (v + (idx) * outlen)
  /* ... Processing ... */
  for (unsigned int j = 0; j < depth; j++) {
    unsigned int depthloop = num_instances >> (j + 1);
    for (unsigned int idx = 0; idx < depthloop; idx++) {
      xor_u8_array(V(j), R(j, 2 * idx + 1), V(j), outlen);
      xor_u8_array(R(j, 2 * idx), R(j, 2 * idx + 1), R(j + 1, idx), outlen);
    }
  }
  if (!sd0_bot && u != NULL) {
    memcpy(u, R(depth, 0), outlen);
  }
/* ... Processing ... */
}

void xor_u8_array(const uint8_t* a, const uint8_t* b, uint8_t* out, size_t len) {
  for (size_t i = 0; i < len; i++) {
    out[i] = a[i]^b[i];
  }
}
```

Figure 5: The C code of `convert_to_vole` and `xor_u8_array`.

the byte-wise Hamming weight information of both $\mathbf{V}$ and $\mathbf{u}$ becomes available, providing the leakage source for the following vertical DPA-style attack.

A specific attack instance. Building on this leakage source, we present a concrete instance, as detailed in Algorithm 2. Since the leakage is byte-wise, we adopt a byte-level divide-and-conquer strategy and define the hypothesis space for the i-th witness byte as $\bar{\mathbf{w}}^{(i)} \in [0, 2^8)$.

As in class 1, the attack aggregates leakage over N traces, but exploits Hamming weight leakage instead of bit leakage. From each execution we derive the public value $\mathbf{d}_j$ by running the verification procedure, and by Equation 2 each candidate $\bar{\mathbf{w}}^{(i)}$ induces the hypothetical byte

$$\bar{\mathbf{u}}_j^{(i)} = \mathbf{d}_j^{(i)} \oplus \bar{\mathbf{w}}^{(i)}.$$

The classifier predicts the byte-wise Hamming weight $\widehat{h}_{i,j}$ from the corresponding trace segment, where i and j index the byte position and the trace, respectively. Thus, for each candidate $\bar{\mathbf{w}}^{(i)}$, the distinguisher compares $(\widehat{h}_{i,j})_{j=0}^{N-1}$ with $(\mathrm{HW}(\bar{\mathbf{u}}_j^{(i)}))_{j=0}^{N-1}$ over all N traces. We take the candidate with the highest score as $\widetilde{\mathbf{w}}^{(i)}$. Repeating this over all $\lambda/8$ byte positions recovers the first λ bits of the witness, which correspond to the FAEST secret key.

Additional attack points. A similar attack can be mounted on the VOLE conversion routine of SDitH, following the same steps as above, except that the order of the XOR operations is adjusted to the SDitH implementation. Since this attack also exploits the Hamming weight leakage of $\mathbf{u}$ in a vertical setting, it can be viewed as another instantiation of class 2.

Extension to a horizontal setting. Instance 2 also admits a horizontal variant. Within a single signing, the VOLE conversion routine is executed once per small VOLE instance, i.e., τ times in total, and each execution leaks the byte-wise Hamming weights of its own output $\mathbf{u}_k$, $k \in [0, \tau)$. Since the correction values $\mathbf{c}_k$ relating $\mathbf{u}_k$ to $\mathbf{u}$ are public, a single witness byte hypothesis $\bar{\mathbf{w}}^{(i)}$ determines the corresponding byte in all τ executions simultaneously.

Algorithm 2 Attack instance 2.

Input: N signatures $\{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$; N corresponding traces $\{t_0, t_1, \ldots, t_{N-1}\}$;
Input: profiled classifier C; distinguisher D.
Output: sk.

1: **for** $j \in [0, N)$ **do**
2: $\quad \mathbf{d}_j \leftarrow \text{FAEST.Verify}(\sigma_j, \ldots)$ ▷ Parse σ_j
3: $\quad \widehat{\boldsymbol{h}}_j \leftarrow C(\text{Preprocess}(t_j))$ ▷ Predict byte-wise HWs of $\mathbf{u}_j$
4: **end for**
5: **for** $i \in [0, \lambda/8)$ **do** ▷ Byte-level divide-and-conquer
6: $\quad \widetilde{\mathbf{w}}^{(i)} \leftarrow \underset{\bar{\mathbf{w}}^{(i)} \in [0,2^8)}{\arg\max}\ D\left(\widehat{h}_{i,j}, \text{HW}\left(\mathbf{d}_j^{(i)} \oplus \bar{\mathbf{w}}^{(i)}\right)\right)_{j=0}^{N-1}$
7: **end for**
8: $sk \leftarrow \widetilde{\mathbf{w}}[0:\lambda-1]$
9: **return** sk

The remainder of the attack follows the same DPA-style procedure as in instance 2, the only difference being that the leakage samples are aggregated across the τ instances of one trace rather than across multiple signatures.

3.4 Horizontal attacks

Among horizontal DPA-style attacks, we identify four classes. Three of them exploit the VOLE tags: one via bit leakage, one via Hamming weight leakage of the row-wise layout of $\mathbf{V}$, and one via Hamming weight leakage of the column-wise layout of $\mathbf{V}$. The fourth exploits bit leakage of the random bits $\mathbf{u}$.

3.4.1 Class 3: horizontal attack using bit leakage of V

Leakage analysis. Class 3 considers bit leakage of the VOLE tag matrix $\mathbf{V}$, but unlike classes 1 and 2, it exploits this leakage in a horizontal setting. By Equation 3, each candidate $\bar{\mathbf{w}}[i]$ determines the corresponding row $\bar{\mathbf{V}}[i]$, whose bit positions provide multiple SIVs for the same hypothesis within one execution.

A concrete leakage source is the bit extraction mechanism in `bf128_mul` analyzed in class 1. Here, however, successive chunks of $\mathbf{V}$ rather than $\mathbf{u}$ are passed to `bf128_mul`. Since the operand `rhs` is processed bit by bit, the leakage reveals the individual processed bits of $\mathbf{V}$, which serve as the bit-level SIVs that class 3 exploits.

A specific attack instance. Exploiting these SIVs, we present a concrete instance, as detailed in Algorithm 3. We reconstruct the public values $\mathbf{Q}$ and $\boldsymbol{\Delta}$ from the signature σ by running the verification procedure. By Equation 3, for the i-th witness bit, each candidate $\bar{\mathbf{w}}[i] \in \{0, 1\}$ determines a hypothetical value of the i-th row of $\mathbf{V}$; at bit position x, this value is

$$\bar{\mathbf{V}}[i][x] = \mathbf{Q}[i][x] \oplus \bar{\mathbf{w}}[i] \cdot \boldsymbol{\Delta}[x].$$

This equation gives the attack its horizontal structure: a single hypothesis $\bar{\mathbf{w}}[i]$ determines the hypothetical row $\bar{\mathbf{V}}[i]$, whose bit positions serve as multiple SIVs within one execution. These positions are informative only when $\boldsymbol{\Delta}[x] = 1$; where $\boldsymbol{\Delta}[x] = 0$, the bit $\bar{\mathbf{V}}[i][x] = \mathbf{Q}[i][x]$ is independent of the hypothesis and hence carries no information. We therefore restrict the scoring to the informative positions $\mathcal{I}_{\boldsymbol{\Delta}} \overset{\text{def}}{=} \{x \in [0, \lambda) : \boldsymbol{\Delta}[x] = 1\}$, over which the distinguisher compares the predicted bits $\widehat{\mathbf{V}}[i][x]$ with the hypothetical ones and selects the highest-scoring hypothesis.

Algorithm 3 Attack instance 3.

Input: a signature σ; a corresponding trace t; profiled classifier C; distinguisher D.
Output: sk.

1: $(\mathbf{Q}, \mathbf{\Delta}) \leftarrow \text{FAEST.Verify}(\sigma, \dots)$ ▷ Parse σ
2: $\widehat{\mathbf{V}} \leftarrow C(\text{Preprocess}(t))$ ▷ Predict the bits of $\mathbf{V}$
3: $\mathcal{I}_{\mathbf{\Delta}} \leftarrow \{x \in [0, \lambda) : \mathbf{\Delta}[x] = 1\}$ ▷ Select informative positions
4: **for** $i \in [0, \lambda)$ **do** ▷ Bit-level divide-and-conquer
5: $\quad \widetilde{\mathbf{w}}[i] \leftarrow \underset{\bar{\mathbf{w}}[i] \in \{0,1\}}{\arg\max}\ D\left(\widehat{\mathbf{V}}[i][x], \mathbf{Q}[i][x] \oplus \bar{\mathbf{w}}[i] \cdot \mathbf{\Delta}[x]\right)_{x \in \mathcal{I}_{\mathbf{\Delta}}}$
6: **end for**
7: $sk \leftarrow \widetilde{\mathbf{w}}[0 : \lambda - 1]$
8: **return** sk

Additional attack points. We identify an attack point of this class in the VOLE transpose of FAEST v2. This routine reorders $\mathbf{V}$ into its row-wise layout through bit moves, and the rows it processes include the rows $\mathbf{V}[i]$ associated with the first λ witness bits, which form the FAEST secret key. Class 3 therefore applies directly to these rows: the bit moves leak the individual bits of $\mathbf{V}[i]$, from which the witness bits are recovered as in Algorithm 3. In contrast, Jendral and Dubrova [JD26b] attack the VOLE transpose of FAEST v1, recovering the VOLE tags associated with the S-box outputs of the first four AES key-schedule rounds, rather than the first λ witness bits, and then reconstructing the fourth round key and reversing the key schedule to recover sk. A similar attack point also exists in the VOLE transpose of SDitH. Although the corresponding function in the reference code relies on `__uint128_t`, which cannot be compiled for the 32-bit ARM architecture, the same leakage risk may arise in future ARM-based implementations that support these operations. Beyond the VOLE transpose, Godard et al. [GAG+26] attack the field multiplication in ZK.EncCstrnts of FAEST and PIOP.Prover.Run of SDitH. These operations leak $\mathbf{V}$ at the bit level in the same way as class 3 and can therefore be viewed as further instantiations of this class.

3.4.2 Class 4: horizontal attack using row-wise HW leakage of V

Leakage analysis. In class 4, we target the row-wise layout of $\mathbf{V}$ under byte-wise Hamming weight leakage[1]. The attack point lies in the VOLE consistency check of SDitH, specifically in the function `matrix_f2_times_vector_flambda_ref`, shown in Figure 6. This function multiplies the binary matrix $\mathbf{M}$ (`a` in the code) by a block vector `b`, where the blocks of `b` correspond to $\mathbf{V}$, and writes the accumulated result to `res`. Each entry of $\mathbf{M}$ is processed by a conditional XOR: when $\mathbf{M}[r][i] = 1$, the block corresponding to $\mathbf{V}[i]$ is XORed byte by byte into the r-th result block of `res`; when $\mathbf{M}[r][i] = 0$, this step is skipped. Thus, for a given column i, the byte-wise writes involving $\mathbf{V}[i]$ update the result blocks of all rows r with $\mathbf{M}[r][i] = 1$. As before, these writes leak the Hamming weights of the updated result-block bytes.

This leakage differs from the bit leakage exploited in class 3, where we obtain predicted bit values of $\mathbf{V}[i]$ and compare them directly with bit-level hypotheses. In the present case, we observe only byte-wise Hamming weights of result-block bytes. In addition, since the result block may already hold contributions from previously processed columns, the leakage observed while processing $\mathbf{V}[i]$ is not the Hamming weight of a byte of $\mathbf{V}[i]$ alone, but that of the XOR between $\mathbf{V}[i]$ and the previously accumulated rows selected by $\mathbf{M}$. Even so,

[1] For simplicity, in the SDitH attacks in classes 4 and 6, we index the witness part of $\mathbf{u}$ and $\mathbf{V}$ directly by the witness index. In the SDitH ordering, this part appears after the first $\lambda + B$ auxiliary entries. Thus, $\mathbf{u}[i]$ and $\mathbf{V}[i]$ refer to the entries associated with the i-th witness bit $\mathbf{w}[i]$, rather than to the entries at global index i in the SDitH ordering. The same convention is used for the corresponding $\mathbf{Q}[i]$.

```c
void matrix_f2_times_vector_flambda_ref(uint64_t lambda, uint64_t nrows,
                                        uint64_t ncols, flambda_t* res,
                                        const bitmat_t* a, const flambda_t* b)
{
    const uint8_t* amat = (uint8_t*)a;
    const uint8_t* bvec = (uint8_t*)b;
    /* ... Processing ... */
    // For each row i
    for (uint64_t i = 0; i < nrows; i++) {
        // Pointer to the result block for row i
        uint8_t* r_i = res + i * lambda_bytes;
        for (uint64_t j = 0; j < ncols; j++) {
            // Extract bit A[i,j]
            uint64_t bit_index = i*ncols + j;
            uint8_t a_ij = (amat[bit_index/8] >> (bit_index%8)) & 1;
            if (a_ij) {
                // If a_ij is 1, XOR the block B[j] into r_i
                const uint8_t* b_j = bvec + j * lambda_bytes;
                for (int x = 0; x < lambda_bytes; x++) {
                    r_i[x] ^= b_j[x];
                }
            }
        }
    }
}
```

Figure 6: The C code of `matrix_f2_times_vector_flambda_ref`.

the leakage can still be linked to the witness progressively. By Equation 3, each row $\mathbf{V}[i]$ is determined by the public values $\mathbf{Q}[i]$ and $\boldsymbol{\Delta}$, together with the single witness bit $\mathbf{w}[i]$. Once the previous witness bits have been recovered, we can compute the corresponding rows of $\mathbf{V}$ and incorporate them into the hypothetical result-block byte for the current witness-bit hypothesis. The only term that remains unknown is then the current row $\mathbf{V}[i]$, which depends on the current hypothesis $\bar{\mathbf{w}}[i]$.

A specific attack instance. Building on this progressive recovery, we present a concrete instance, as detailed in Algorithm 4. As before, we reconstruct the public values $\mathbf{Q}$, $\boldsymbol{\Delta}$, and $\mathbf{M}$ from σ. For the i-th witness bit, we test the two hypotheses $\bar{\mathbf{w}}[i] \in \{0, 1\}$. Each hypothesis determines a candidate row

$$\bar{\mathbf{V}}[i] = \mathbf{Q}[i] \oplus \bar{\mathbf{w}}[i] \cdot \boldsymbol{\Delta}.$$

For the current witness-bit index i, the row $\mathbf{V}[i]$ is XORed into the r-th result block only when $\mathbf{M}[r][i] = 1$. We therefore collect the row indices r whose result block is updated by $\mathbf{V}[i]$, namely

$$\mathcal{I}_{\mathbf{M}} \stackrel{\text{def}}{=} \{r : \mathbf{M}[r][i] = 1\}.$$

We recover the witness bits in increasing order of the column index i, matching the order in which the columns of $\mathbf{M}$ are accumulated. For each $r \in \mathcal{I}_{\mathbf{M}}$ and each byte position x, the hypothetical result-block byte under the candidate $\bar{\mathbf{w}}[i]$ is

$$\bar{\mathbf{r}}_{r,i}^{(x)} = \left(\bigoplus_{i'=0}^{i-1} \mathbf{M}[r][i'] \cdot \left(\widetilde{\mathbf{V}}[i'] \right)^{(x)} \right) \oplus \left(\bar{\mathbf{V}}[i] \right)^{(x)}.$$

For $i = 0$, the sum $\bigoplus_{i'=0}^{i-1}$ is empty and treated as zero. For $i > 0$, it incorporates the rows $\widetilde{\mathbf{V}}[i']$ with $i' < i$ computed from the previously recovered witness bits.

From the trace t, we extract the segments corresponding to the byte-wise result-block writes at column i and feed them into the profiled classifier C. For each $r \in \mathcal{I}_{\mathbf{M}}$ and byte position x, the classifier outputs the predicted Hamming weight of the result-block byte after processing column i of $\mathbf{M}$, denoted by $\widehat{h}_{r,i,x}$. The distinguisher compares $\widehat{h}_{r,i,x}$

Algorithm 4 Attack instance 4.

Input: a signature σ; a corresponding trace t; profiled classifier C; distinguisher D.
Output: $\widetilde{\mathbf{w}}$.

$(\mathbf{Q}, \mathbf{\Delta}, \mathbf{M}) \leftarrow \mathrm{SDitH.Verify}(\sigma, \dots)$ ▷ Parse σ
$\widehat{\boldsymbol{h}} \leftarrow C(\mathrm{Preprocess}(t))$ ▷ Predict byte-wise HWs of result-block writes
for $i \in [0, |\mathsf{wit}|_2)$ **do** ▷ Bit-level progressive recovery
  $\mathcal{I}_{\mathbf{M}} \leftarrow \{r : \mathbf{M}[r][i] = 1\}$ ▷ Select rows r updated by $\mathbf{V}[i]$
  for $\bar{\mathbf{w}}[i] \in \{0, 1\}$ **do**
    $\bar{\mathbf{V}}[i] \leftarrow \mathbf{Q}[i] \oplus \bar{\mathbf{w}}[i] \cdot \mathbf{\Delta}$
    $\bar{\mathbf{r}}_{r,i}^{(x)} \leftarrow \left(\bigoplus_{i'=0}^{i-1} \mathbf{M}[r][i'] \cdot \left(\widetilde{\mathbf{V}}[i'] \right)^{(x)} \right) \oplus \left(\bar{\mathbf{V}}[i] \right)^{(x)}$ for $r \in \mathcal{I}_{\mathbf{M}}$ and $x \in [0, \lambda/8)$
    $\rho^{\bar{\mathbf{w}}[i]} \leftarrow D\left(\widehat{h}_{r,i,x}, \mathrm{HW}\left(\bar{\mathbf{r}}_{r,i}^{(x)} \right) \right)_{r \in \mathcal{I}_{\mathbf{M}},\, x \in [0, \lambda/8)}$
  end for
  $\widetilde{\mathbf{w}}[i] \leftarrow \underset{\bar{\mathbf{w}}[i] \in \{0,1\}}{\arg\max}\ \rho^{\bar{\mathbf{w}}[i]}$
  $\widetilde{\mathbf{V}}[i] \leftarrow \mathbf{Q}[i] \oplus \widetilde{\mathbf{w}}[i] \cdot \mathbf{\Delta}$ ▷ Store the recovered row $\widetilde{\mathbf{V}}[i]$ for subsequent recovery
end for
return $\widetilde{\mathbf{w}}$

with $\mathrm{HW}(\bar{\mathbf{r}}_{r,i}^{(x)})$ over all selected rows $r \in \mathcal{I}_{\mathbf{M}}$ and byte positions x, and selects the highest-scoring hypothesis. After recovering $\widetilde{\mathbf{w}}[i]$, we compute $\widetilde{\mathbf{V}}[i]$ and use it in the hypothetical result-block bytes for subsequent witness bits. Iterating this procedure over all i recovers the full witness.

Additional attack points. An additional attack point of class 4 appears in the "build lines" procedure of SDitH. This procedure implements finite-field multiplication by decomposing it into bit shifts, XOR accumulation, and a final reduction modulo the field polynomial. During the XOR accumulation phase, intermediate values derived from rows of $\mathbf{V}$ are repeatedly updated by shifts and XORs, and their Hamming weight leakage yields another instantiation of class 4. Like the VOLE transpose routine of SDitH, the reference C implementation of build lines relies on `__uint128_t` and hence cannot be compiled for the 32-bit ARM architecture either. We therefore leave the experimental validation of this attack point on an ARM-optimized implementation to future work.

3.4.3 Class 5: horizontal attack using column-wise HW leakage of V

Leakage analysis. The same VOLE conversion routine of FAEST exploited in class 2 also leaks $\mathbf{V}$, which we now target instead of $\mathbf{u}$. Whereas class 4 observes $\mathbf{V}$ through its row-wise layout, class 5 exploits the column-wise layout in which $\mathbf{V}$ is also stored. These two layouts group the bits of $\mathbf{V}$ differently. In the row-wise layout, a byte contains eight consecutive bits from a single row of $\mathbf{V}$. In the column-wise layout, by contrast, a byte contains one bit from each of eight consecutive rows. Hence, the i-th byte of column x is $\big(\mathbf{V}[8i][x], \mathbf{V}[8i+1][x], \dots, \mathbf{V}[8i+7][x]\big)$, and its Hamming weight is $\sum_{y=0}^{7} \mathbf{V}[8i+y][x]$. Thus, one leakage sample in the column-wise layout combines the bits of eight consecutive rows of $\mathbf{V}$ at the same column position x. Since these eight rows correspond to eight consecutive witness bits, the natural divide-and-conquer unit is one witness byte rather than one witness bit.

A specific attack instance. Algorithm 5 details a concrete instance that exploits the Hamming weight leakage of the column-wise bytes of $\mathbf{V}$. After reconstructing $\mathbf{Q}$ and $\mathbf{\Delta}$,

Algorithm 5 Attack instance 5.

Input: a signature σ; a corresponding trace t; profiled classifier C; distinguisher D.
Output: sk.

1: $(\mathbf{Q}, \mathbf{\Delta}) \leftarrow \text{FAEST.Verify}(\sigma, \dots)$ ▷ Parse σ
2: $\widehat{\boldsymbol{h}} \leftarrow C(\text{Preprocess}(t))$ ▷ Predict HWs of column-wise bytes of $\mathbf{V}$
3: **for** $i \in [0, \lambda/8)$ **do** ▷ Byte-level divide-and-conquer
4: $\quad \widetilde{\mathbf{w}}^{(i)} \leftarrow \underset{\bar{\mathbf{w}}^{(i)} \in [0,2^8)}{\arg\max}\ D\left(\widehat{h}_{i,x}, \sum_{y=0}^{7} \left(\mathbf{Q}[8i+y][x] \oplus \bar{\mathbf{w}}[8i+y] \cdot \mathbf{\Delta}[x]\right)\right)_{x \in [0,\lambda)}$
5: **end for**
6: $sk \leftarrow \widetilde{\mathbf{w}}[0 : \lambda - 1]$
7: **return** sk

```c
uint8_t bitvec_get_value(const bitvec_t* v, uint64_t idx) {
  const uint8_t* vec = (const uint8_t*)v;
  return ((vec[idx / 8] >> (idx % 8)) & 1);
}

void matrix_vector_product_f2_ref(uint64_t nrows, uint64_t ncols, bitvec_t* res,
                                  const bitmat_t* a, const bitvec_t* b) {
  memset(res, 0, (nrows + 7) / 8);
  for (uint64_t i = 0; i < nrows; ++i) {
    uint8_t ri = 0;
    for (uint64_t j = 0; j < ncols; ++j) {
      ri ^= bitmat_get_value(a, ncols, i, j) & bitvec_get_value(b, j);
    }
    bitvec_value_xor_to(res, i, ri);
  }
}
```

Figure 7: The C code of `bitvec_get_value` and `matrix_vector_product_f2_ref`.

we enumerate candidates $\bar{\mathbf{w}}^{(i)} \in [0, 2^8)$ for the i-th witness byte. By Equation 3, the hypothetical Hamming weight of the column-x byte under the candidate $\bar{\mathbf{w}}^{(i)}$ is

$$\sum_{y=0}^{7} \left(\mathbf{Q}[8i+y][x] \oplus \bar{\mathbf{w}}[8i+y] \cdot \mathbf{\Delta}[x]\right).$$

From the trace, the classifier outputs the predicted Hamming weight $\widehat{h}_{i,x}$ of the corresponding column-wise byte. The distinguisher then compares $\widehat{h}_{i,x}$ with the hypothetical Hamming weights over all column positions $x \in [0, \lambda)$ and selects the highest-scoring byte candidate. Iterating this byte-level procedure for all $i \in [0, \lambda/8)$ recovers the first λ bits of the witness, that is, the FAEST secret key.

Additional attack points. Just as class 2 extends to the VOLE conversion routine of SDitH, so does class 5: the same SDitH routine that leaks $\mathbf{u}$ also leaks $\mathbf{V}$ through its column-wise layout. With the order of the XOR operations adapted to the SDitH implementation, this gives another instantiation of class 5.

3.4.4 Class 6: horizontal attack using bit leakage of u

Leakage analysis. In this class, we again consider bit leakage of $\mathbf{u}$, but in a horizontal setting. Compared with class 1, it exploits the repeated processing of the same bit of $\mathbf{u}$ within a single execution and combines the corresponding leakage samples to score the two hypotheses for each witness bit.

The attack point lies in the consistency check of $\mathbf{u}$ in BLC.Commit of SDitH, implemented by the matrix-vector multiplication function `matrix_vector_product_f2_ref`

Algorithm 6 Attack instance 6.

Input: a signature σ; a corresponding trace t; profiled classifier C; distinguisher D.
Output: $\widetilde{\mathbf{w}}$.

1: $\mathbf{d} \leftarrow \mathrm{SDitH.Verify}(\sigma, \dots)$ ▷ Parse σ
2: $\{\widehat{\mathbf{u}}_s\}_{s=0}^{\mathtt{nrows}-1} \leftarrow C(\mathrm{Preprocess}(t))$ ▷ Predict the bit of $\mathbf{u}$ at each access
3: **for** $i \in [0, |\mathsf{wit}|_2)$ **do** ▷ Bit-level divide-and-conquer
4: $\quad \widetilde{\mathbf{w}}[i] \leftarrow \underset{\bar{\mathbf{w}}[i]\in\{0,1\}}{\arg\max}\ D\left((\widehat{\mathbf{u}}_s[i])_{s=0}^{\mathtt{nrows}-1},\ \mathbf{d}[i] \oplus \bar{\mathbf{w}}[i]\right)$
5: **end for**
6: **return** $\widetilde{\mathbf{w}}$

(Figure 7), which multiplies the matrix $\mathbf{M}$ by the vector $\mathbf{u}$ over $\mathbb{F}_2$: for each row of $\mathbf{M}$, it computes an inner product by XOR-accumulating the pairwise bit products.

The leakage arises from how this multiplication accesses $\mathbf{u}$. For each of the `nrows` rows of $\mathbf{M}$, the routine recomputes the inner product with $\mathbf{u}$, calling the subfunction `bitvec_get_value` to read each bit of $\mathbf{u}$ from its packed representation. Repeated across all `nrows` rows, this reads each bit $\mathbf{u}[i]$ a total of `nrows` times, and each read incurs power consumption that depends on the value of $\mathbf{u}[i]$, making the bit distinguishable in the trace, as illustrated in Figure 4 (right). A single trace thus provides `nrows` bit-level leakage samples of the same SIV $\mathbf{u}[i]$.

A specific attack instance. Leveraging these repeated samples, we present a concrete instance, as detailed in Algorithm 6. The difference from instance 1 lies in the dimension over which the score is computed. In instance 1, the distinguisher aggregates leakage samples from multiple independent traces. Here, we instead aggregate the repeated leakage samples corresponding to the same bit $\mathbf{u}[i]$ within a single trace. Let $\widehat{\mathbf{u}}_s[i]$ denote the predicted bit from the s-th access to $\mathbf{u}[i]$. For each hypothesis $\bar{\mathbf{w}}[i] \in \{0, 1\}$, the score is computed by comparing the repeated predictions $\widehat{\mathbf{u}}_s[i]$ with the hypothetical bit $\mathbf{d}[i] \oplus \bar{\mathbf{w}}[i]$ over all accesses $s \in [0, \mathtt{nrows})$. The highest-scoring hypothesis is selected as $\widetilde{\mathbf{w}}[i]$. Repeating this procedure for all $i \in [0, |\mathsf{wit}|_2)$ recovers the witness.

3.5 Summary of discovered attacks

Table 3 summarizes our DPA-style attacks against FAEST and SDitH, covering six classes of attacks on the VOLE-correlated intermediate variables $\mathbf{u}$ and $\mathbf{V}$. Previously reported attacks are also included, most of which fall into these classes, except the byte combine attack of [JD26b], which targets the witness itself. As shown in the preceding leakage analyses, the two schemes exhibit similar leakage points across several key routines, namely VOLE conversion, VOLE transpose, and the VOLE consistency check. As described in Section 2.3, these routines generate the intermediate variables, arrange their representations for subsequent computations, and check the consistency of the VOLE correlations. Their respective roles explain why attack points concentrate where $\mathbf{u}$ and $\mathbf{V}$ are generated or directly manipulated.

VOLE conversion. Generating the VOLE correlations requires explicitly constructing $\mathbf{u}$ and column-wise $\mathbf{V}$ through XOR operations. The corresponding byte-wise writes can leak the Hamming weights of the resulting bytes, making the generation process a direct leakage source for both intermediate variables. Their storage layouts further determine which witness-dependent bits are processed together.

VOLE consistency check. Ensuring the consistency of the combined VOLE instances requires linear compression of $\mathbf{u}$ and $\mathbf{V}$ through finite-field or matrix multiplications. These

Table 3: Summary of various DPA-style attack instances. "Class" specifies the attack class; "HPW" denotes the Hypothesized Partial Witness; and "Type" indicates whether the attack is Vertical (V) or Horizontal (H). In the SIV column, the index sets are $\mathcal{I}_{\mathbf{\Delta}} \stackrel{\text{def}}{=} \{x \in [0,\lambda) : \mathbf{\Delta}[x] = 1\}$, $\mathcal{I}_b \stackrel{\text{def}}{=} [0,\lambda)$, and $\mathcal{I}_B \stackrel{\text{def}}{=} [0,\lambda/8)$.

a) Attacks against FAEST

Reference	Class	Routine	Operation	HPW	SIV	Type
This work	2	VOLE conversion	XOR	$\bar{\mathbf{w}}^{(i)}$	$\mathbf{u}^{(i)}$	V
	5	VOLE conversion	XOR	$\bar{\mathbf{w}}^{(i)}$	$(\mathbf{V}^T[\mathcal{I}_b])^{(i)}$	H
	3	VOLE transpose	bit move	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H
	1	consistency check	mul. in $\mathbb{F}_{2^\lambda}$	$\bar{\mathbf{w}}[i]$	$\mathbf{u}[i]$	V
	3	consistency check	mul. in $\mathbb{F}_{2^\lambda}$	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H
[GAG+26]	3	ZK.EncCstrnts	mul. in $\mathbb{F}_{2^\lambda}$	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H
[JD26b] (v1)	3	VOLE transpose	bit move	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H

b) Attacks against SDitH

Reference	Class	Routine	Operation	HPW	SIV	Type
This work	2	VOLE conversion	XOR	$\bar{\mathbf{w}}^{(i)}$	$\mathbf{u}^{(i)}$	V
	5	VOLE conversion	XOR	$\bar{\mathbf{w}}^{(i)}$	$(\mathbf{V}^T[\mathcal{I}_b])^{(i)}$	H
	3	VOLE transpose	bit move	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H
	4	consistency check	XOR	$\bar{\mathbf{w}}[i]$	$(\mathbf{V}[i])^{(\mathcal{I}_B)}$	H
	6	consistency check	mul. in $\mathbb{F}_2$	$\bar{\mathbf{w}}[i]$	$\mathbf{u}[i]$	H
	4	build lines	$\mathbb{F}_2^{7\kappa} \to \mathbb{F}_{2^\lambda}$	$\bar{\mathbf{w}}[i]$	$(\mathbf{V}[i])^{(\mathcal{I}_B)}$	H
[GAG+26]	3	PIOP.Prover.Run	mul. in $\mathbb{F}_{2^\lambda}$	$\bar{\mathbf{w}}[i]$	$\mathbf{V}[i][\mathcal{I}_{\mathbf{\Delta}}]$	H

computations directly process both intermediate variables. Multiplication involves bit extraction and mask generation, or XOR accumulation, which can expose individual operand bits or Hamming weights of accumulated results. Processing these inputs across multiple field operations or matrix rows can also provide multiple observations associated with the same witness hypothesis. Together, this fine-grained processing and repeated use create multiple opportunities for witness-dependent leakage within a single signing execution.

Similarly, VOLE transpose reorganizes the tag matrix assembled from the small VOLE instances into row-wise storage, allowing each row to be read as one field element without altering the underlying correlation. This reorganization involves bit extraction and movement, creating further leakage opportunities. Related operations also occur in ZK constraint computations and SDitH's "build lines" procedure.

These similarities are structural rather than incidental. The routines implement the same underlying VOLE-correlated computations in both schemes, so the identified attack points are rooted in the computational structure shared by VOLEitH signature schemes. This observation suggests that the framework can extend beyond FAEST and SDitH to other VOLEitH-based schemes such as PERK. Although we only present representative attack points here, $\mathbf{u}$ and $\mathbf{V}$ are involved in many other operations during signing. Further leakage points involving them are therefore expected to be analyzable within the same framework, with the corresponding attacks instantiated in an analogous manner.

4 Evaluation

In this Section, we first evaluate the proposed attacks in simulations under different noise conditions, and then validate the practical exploitability of the targeted leakage on a real ARM Cortex-M4 device.

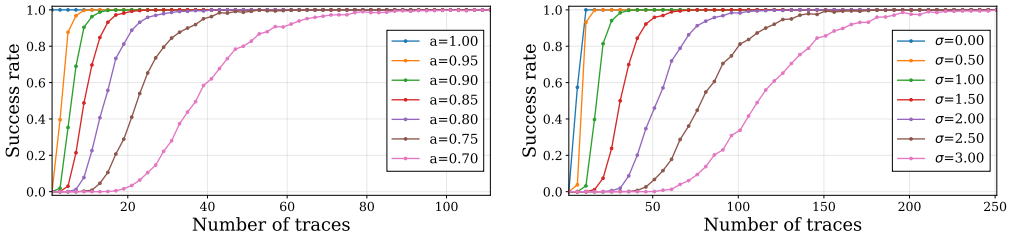

Figure 8: Success rate of the two vertical DPA-style attack instances as a function of the number of traces. Instance 1 under bit leakage for different classification accuracies α with step size two (left). Instance 2 under Hamming weight leakage for different noise levels σ with step size five (right).

4.1 Simulation

Experimental setup. For instances based on bit leakage, we use the classification accuracy α to model the probability that a leaked bit is correctly recovered. For instances based on Hamming weight leakage, we adopt the standard Hamming weight model with additive Gaussian noise, as in [HWZY26]. More precisely, the leakage of each byte is modeled as its Hamming weight, taking values from 0 to 8, plus a zero-mean Gaussian perturbation with standard deviation σ. Each success rate is averaged over 1000 independent tests.

We evaluate vertical and horizontal attacks separately. For vertical attacks, we examine how the success rate grows with the number of traces. For horizontal attacks, which expose multiple leakage points within a single execution, we focus on the single-trace setting. In both cases, we further study how the success rate varies with the classification accuracy α for instances based on bit leakage and with the noise level σ for instances based on Hamming weight leakage.

Experimental results: vertical attacks. The two vertical attacks, instance 1 and instance 2, target $\mathbf{u}$ under bit leakage and Hamming weight leakage, respectively. Since vertical attacks aggregate leakage over many independent executions, the success rate is reported as a function of the number of traces in Figure 8. For instance 1, the classification accuracy α of the bit predictor is varied, whereas for instance 2, the noise level σ of the Hamming weight model is varied.

Figure 8 (left) shows the success rate of instance 1 for different accuracies α, with the number of traces increased in steps of two. When the classification accuracy is perfect ($\alpha = 1.00$), a single trace suffices to recover the full witness, since each bit of $\mathbf{u}$ is predicted without error. As α decreases, more traces are required to average out the mispredictions: full recovery requires 11 traces at $\alpha = 0.95$ and the required number grows for lower accuracies. Even at a relatively low accuracy of $\alpha = 0.70$, the attack still succeeds, reaching a success rate of 1.0 at 119 traces, although it remains near 0 with only 11 traces. Figure 8 (right) reports the success rate of instance 2 under different noise levels σ, where the number of traces is increased in steps of five. The results exhibit the same general trend as instance 1. Without noise ($\sigma = 0.00$), full recovery is achieved with 11 traces. For $\sigma = 0.50$ and $\sigma = 1.00$, full recovery requires 21 and 41 traces, respectively. The attack also remains effective under stronger noise. When $\sigma = 3.00$, the success rate exceeds 0.9 once the number of traces is above 166, and reaches 1.0 with 256 traces. Overall, given enough traces, instance 2 recovers the full witness across all tested noise levels, with the required number increasing as the noise grows.

Experimental results: horizontal attacks. Instance 3 and instance 6 rely on bit leakage and are evaluated over a range of classification accuracies α (Table 4), whereas instance 4

Table 4: Success rate of horizontal DPA-style attack instances under different accuracies.

Accuracy α	0.95	0.90	0.85	0.80	0.75	0.70	0.65	0.60
Instance 3	1.0	1.0	1.0	1.0	0.997	0.953	0.397	0
Instance 6	1.0	1.0	1.0	1.0	1.0	1.0	0.903	0.002

Table 5: Success rate of horizontal DPA-style attack instances under different noise conditions.

Noise σ	0.50	1.00	1.50	2.00	2.50	3.00
Instance 4	1.0	1.0	1.0	1.0	1.0	1.0
Instance 5	1.0	0.996	0.578	0.051	0	0

and instance 5 rely on Hamming weight leakage and are evaluated over a range of noise levels σ (Table 5). For the two attacks based on bit leakage, instance 3 and instance 6, the success rate declines as the classification accuracy α drops, yet both remain highly effective over a wide range of accuracies, staying close to 1.0 for $\alpha \geq 0.75$. Instance 3 achieves a success rate of 0.953 at $\alpha = 0.70$, which falls to 0.397 at $\alpha = 0.65$ and reaches 0 at $\alpha = 0.60$. Instance 6 retains a success rate of 1.0 down to $\alpha = 0.70$, declining to 0.903 at $\alpha = 0.65$ and falling to 0.002 at $\alpha = 0.60$. For instance 4 and instance 5, which rely on Hamming weight leakage, the success rate decreases as noise increases. Instance 4 maintains a success rate of 1.0 across all tested noise levels, up to $\sigma = 3.00$. Instance 5 achieves a success rate of 1.0 at $\sigma = 0.50$ and 0.996 at $\sigma = 1.00$, but the success rate drops to 0.578 at $\sigma = 1.50$ and further to 0.051 at $\sigma = 2.00$, with no successful recovery observed at higher noise levels.

4.2 Practical validation of leakage exploitability

The simulations above assume that the bit values and byte Hamming weights leaked by the target operations can be predicted from a power trace. To confirm that these leakage sources are indeed exploitable on a real device, we collect side-channel measurements for these operations, train classifiers for their leakage labels, and evaluate their accuracy.

Experimental setup. We target the reference implementations of FAEST and SDitH submitted to the second round[2], both compiled with `arm-none-eabi-gcc` version `13.2.1` using `-Os` . To enable practical evaluation, we use three CW308T-STM32F4 target boards each hosting an STM32F415-RGT6 chip based on the ARM Cortex-M4 32-bit RISC core and plugged into a CW308 UFO main board. We connect a ChipWhisperer-Lite to the CW308 UFO main board to measure the power consumption of the target. The operating frequency is set to 24 MHz, with a sampling rate of 24 MS/s. For profiling, we use a CNN trained on a PC with an Intel Core i7-14700KF CPU, an NVIDIA GeForce RTX 4090 GPU, and 64GB of RAM, with the batch size set to 4096.

The practical validation targets three functions from the reference implementations: `bf128_mul`, `matrix_vector_product_f2_ref`, and `xor_u8_array`. These functions cover the leakage sources considered in our analysis, namely bit leakage from multiplication over $\mathbb{F}_{2^\lambda}$ and $\mathbb{F}_2$, and Hamming weight leakage from byte-wise XOR operations. We use `xor_u8_array` as a representative implementation of the latter.

On each target board, the procedure is as follows. For each target function, we collect 10,000 raw traces. Each raw trace is split into N segments, one per iteration of the loop surrounding the target operation, with $N = 128$ for `bf128_mul`, 210 for `xor_u8_array`, and 616 for `matrix_vector_product_f2_ref`, which yields $N \times 10{,}000$ labeled segments per

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

Table 6: Classification accuracy of the CNN classifiers in the same-device and cross-device settings. Underlined entries indicate the lowest accuracy for each function.

a) `bf128_mul`

Same-device	D_1	D_2	D_3	**avg**
Accuracy	0.999930	0.999938	0.999984	**0.999951**

Cross-device	Target			**avg**
Profiling	D_1	D_2	D_3	
D_1	–	0.999898	0.999922	0.999910
D_2	0.999680	–	0.999859	0.999770
D_3	<u>0.999648</u>	0.999852	–	0.999750
avg	0.999664	0.999875	0.999891	**0.999810**

b) `matrix_vector_product_f2_ref`

Same-device	D_1	D_2	D_3	**avg**
Accuracy	0.999984	0.999982	0.999995	**0.999987**

Cross-device	Target			**avg**
Profiling	D_1	D_2	D_3	
D_1	–	0.999753	0.999948	0.999851
D_2	<u>0.999417</u>	–	0.999869	0.999643
D_3	0.999959	0.999977	–	0.999968
avg	0.999688	0.999865	0.999909	**0.999821**

c) `xor_u8_array`

Same-device	D_1	D_2	D_3	**avg**
Accuracy	0.999724	0.999724	0.999614	**0.999687**

Cross-device	Target			**avg**
Profiling	D_1	D_2	D_3	
D_1	–	0.973276	0.993167	0.983221
D_2	0.995752	–	0.971724	0.983738
D_3	0.983548	<u>0.968352</u>	–	0.975950
avg	0.989650	0.970814	0.982445	**0.980970**

function. The raw traces are split into training, validation, and test sets in a ratio of 8:1:1. The complete execution of `matrix_vector_product_f2_ref` spans more than one million clock cycles and exceeds the acquisition capacity of our setup. We therefore restrict the measurement to lines 10–14 of Figure 7 for `i=0`. This segment covers the accumulation over all column indices `j`. In particular, the highlighted call to `bitvec_get_value` in line 12 repeatedly accesses the bits of the input vector, which directly corresponds to the leakage source targeted by our analysis. For `bf128_mul` and `matrix_vector_product_f2_ref`, we train binary classifiers to distinguish bit values 0 and 1. For `xor_u8_array`, we train a nine-class classifier for Hamming weight from 0 to 8. Each classifier is trained on and applied to all relevant positions of its target function, so that the reported accuracy is the average over all targeted bit positions for `bf128_mul` and `matrix_vector_product_f2_ref`, and over all targeted byte positions for `xor_u8_array`. As the procedure is carried out on all three boards, we obtain three classifiers per function, and each of them is evaluated on the test sets of all three boards.

Experimental results. Table 6 summarizes the classification results, both when profiling and attack are performed on the same device and when they are performed on different devices. In the same-device setting, all classifiers exceed an accuracy of 0.9995 on every

device. In the cross-device setting, the accuracy of all three classifiers decreases. This is expected, as device-level variations from fabrication and packaging as well as differences in the board and measurement chain typically alter the power consumption characteristics of nominally identical devices [CK14, BCH+20].

Specifically, the average accuracies of `bf128_mul` and `matrix_vector_product_f2_ref` drops only marginally, from 0.999951 and 0.999987 to 0.999810 and 0.999821, respectively, and the lowest accuracy over all six directions remains above 0.9994. The Hamming weight classifier for `xor_u8_array` is more sensitive to the device change, with the average accuracy dropping from 0.999687 to 0.980970 and a minimum of 0.968352 in the direction $D_3 \to D_2$. This is expected, as the nine-class Hamming weight problem is inherently harder than binary classification. Such accuracy is therefore acceptable for the attacks.

We then evaluate the complete attack procedures in the cross-device setting. Each success rate is evaluated over 1000 randomly generated secret keys and their corresponding witnesses. For a conservative evaluation, each instance uses the device pair with the lowest classification accuracy shown in Table 6, which is underlined. For FAEST attack instance 1, we directly apply the classifier profiled on D_3 to the traces collected from D_1, and use the resulting bit predictions to perform the attack described in Algorithm 1. Specifically, for each signing execution, the classifier predicts the targeted bits of $\mathbf{u}$ from the corresponding trace segments, and the predictions from multiple executions are combined by the distinguisher to recover the witness bits. With this practical setting, instance 1 reaches a success rate of 1.0 with 3 traces. For FAEST attack instance 2, we apply the classifier profiled on D_3 to traces collected from D_2. The success rate reaches 1.0 with 18 traces.

For the remaining attack instances, the acquisition window of our measurement setup cannot cover all leakage points required by their complete attack procedures. We therefore use the empirical confusion matrix of the corresponding classifier on the selected device pair. This follows the prediction-matrix approach of [GMGL24, GAG+25], which provides repeatable "real-case-like" simulations by sampling each prediction from the distribution actually produced by the classifier for the corresponding true label, rather than from an idealized leakage model. In this setting, instances 3–6 all reach a 1.0 success rate with a single trace.

These results show that the target leakage can be predicted reliably on a real device and is sufficient to support all of the proposed attacks, even in the cross-device setting. Note that our cross-device evaluation applies the classifiers to the target device without any adaptation. Techniques such as trace normalization [MBTL13], multi-device profiling [BCH+20], and domain adaptation [CZLG21] could further improve the attacks in the cross-device setting. We leave the optimization of the cross-device setting to future work.

5 Applicability to the third-round variants

Our analysis and experiments target the version 2 submissions of FAEST and SDitH. On September 22, 2026, NIST posted updated submission packages for the 3rd Round onramp candidates, including new versions of both schemes. In this section, we examine the resulting changes in the specifications and reference implementations and discuss the applicability of our framework to the new versions.

SDitH. SDitH version 3 [ABB+26] fixes inconsistencies between the specification and the reference implementation and adds salt to the expansion of leaf seeds. It also updates the proof system parameters and shortens secret keys and signatures by removing unused or redundant data. These changes do not alter the VOLE relations underlying our attacks. In particular, Equation 2 and Equation 3 continue to relate witness hypotheses to the corresponding values of $\mathbf{u}$ and $\mathbf{V}$. Moreover, the functions analyzed in our attacks and

```
#define ptr_get_bit(value, index) (((value)[(index) / 8] >> ((index) % 8)) & 1)

    memset(uh_0, 0, uh_0_bytes);
    for (unsigned int bit_idx = 0; bit_idx < ell; ++bit_idx) {
      // [u[0],..0 lambda times, u[1], 0 lambda times, ...]
      uh_0[bit_idx * lambda_bytes] = ptr_get_bit(u, bit_idx);
    }
    memcpy(uh_0 + lambda_bytes * ell, u_bar,
           (D_ZK - 1 + 4) * lambda_bytes);

    vole_hash(signature_u_tilde(sig, params),
              chall_1, uh, ell, D_ZK, lambda);
```

Figure 9: Reference code showing the extraction and embedding of individual bits of **u** before VOLEHash in `faest_sign` of FAEST version 3.

the key operations giving rise to the targeted leakage remain unchanged in the version 3 reference implementation. Therefore, after updating the relevant parameters and public values, our analysis and attacks carry over directly to version 3.

FAEST. FAEST version 3 [BBB+26] makes a more substantial change to the VOLE consistency check. It introduces a new method for committing to λ-bit VOLE masks and requires a different VOLEHash check. In version 2, VOLEHash is linear over $\mathbb{F}_2$ and is applied separately to the packed bit string **u** and each column of **V**. In version 3, VOLEHash is linear over $\mathbb{F}_{2^\lambda}$ and takes a sequence of field elements as input. Each bit of **u** corresponding to the witness is embedded into $\mathbb{F}_{2^\lambda}$, while each row of **V** is represented as a field element. The corresponding VOLE masks and tags are included in the inputs to VOLEHash. The hash evaluates the input coefficients as a polynomial at five challenge points, combines the resulting evaluations into four field elements, and masks them using the final four input elements.

The updated implementation nevertheless introduces several candidate leakage points. When preparing the input to VOLEHash, version 3 extracts each bit of **u** corresponding to the witness from the array `u` using `ptr_get_bit` and stores it as a separate field element, as shown in Figure 9. Unlike version 2, where the packed bit string is processed directly, this additional step uses the same shift and mask pattern as `bitvec_get_value` in Figure 7, providing a candidate leakage point for the extracted bit. Additional leakage can arise from the field multiplications in the new VOLEHash. The reference implementation evaluates the input polynomial using Horner's method, so each coefficient added to the accumulator affects the subsequent field multiplications. For the 128-bit parameter sets, these multiplications call `bf128_mul_inplace`, with `lhs` holding the accumulator and `rhs` holding a public evaluation point derived from `chall_1`. During field reduction, `bf128_bit_to_uint64_mask` expands one bit of the shifted and reduced copy of `lhs` into a mask consisting entirely of zeros or ones, which can therefore produce bit leakage from the accumulator.

These leakage points remain directly related to witness hypotheses. In version 3, **V**, **Q**, and **Δ** denote the values after the public linear mapping into $\mathbb{F}_{2^\lambda}$, with **Q** including the public witness correction. By Equation 2 and Equation 3, a hypothesis for a witness bit determines the corresponding bit of **u** or the corresponding row of **V**, and therefore the corresponding coefficient processed by VOLEHash. Given the previously recovered witness bits and the public values, each hypothesis for the current witness bit further determines the intermediate values in the subsequent field multiplications. These predictions can then be compared with side-channel observations to distinguish the correct hypothesis and recover the witness bit by bit, following the same DPA-style principle used in our framework. Despite the substantial redesign of the VOLE consistency check in FAEST version 3, the dependence of the processed intermediate values on the witness therefore

remains. This further confirms that the identified leakage is not specific to a particular implementation, but instead arises from the computational structure shared by VOLEitH signature schemes.

6 Conclusion

This work provides a comprehensive analysis of side-channel attacks against VOLEitH schemes. By applying our DPA-style attack framework to FAEST and SDitH, we identify multiple attack points throughout the signing procedure. Our analysis shows that several components shared across VOLEitH schemes constitute particularly favorable targets for side-channel analysis in practical implementations. These findings highlight the need for SCA countermeasures in VOLEitH schemes. At the same time, memory constraints already pose challenges for deploying these schemes on embedded devices [KKPY24]. Given this, the design of protected implementations must carefully balance leakage resistance against practical overhead, especially memory usage.

As future work, we plan to extend our framework to MQOM [BBFR25], which is based on Threshold Computation in the Head (TCitH) and has also advanced to the third round. This is a natural extension, since TCitH and VOLEitH are closely related techniques within the broader MPC-in-the-Head paradigm [ABC+26]. We expect several of the DPA-style attack classes identified in this work to have similar counterparts in TCitH schemes.

References

[AAB+25] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath specifications (version 2.1.0), 2025. https://pqc-mirath.org/assets/downloads/mirath_v2.1.0.pdf.

[ABB+25a] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Rainer Esser, Thibauld Feneuil, Philippe Gaborit, Mukul Ramesh Kulkarni, Víctor Mateu Meseguer, Marco Palumbi, Lucas Pandolfo Perin, Matthieu Rivain, Jean-Pierre Tillich, and Keita Xagawa. PERK specifications (version 2.2.0), 2025. https://pqc-perk.org/assets/downloads/perk-v2.2.0.pdf.

[ABB+25b] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. The syndrome decoding in the head (SD-in-the-Head) signature scheme algorithm specifications and supporting documentation (version 2.0), 2025. https://sdith.org/docs/sdith-v2.0.pdf.

[ABB+25c] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Romaric Neveu, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vincotte. RYDE signature scheme (version 2.1.0), 2025. https://pqc-ryde.org/assets/downloads/ryde_specification_v2.1.0.pdf.

[ABB+26] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hüls-

ing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. The syndrome decoding in the head (SD-in-the-Head) signature scheme algorithm specifications and supporting documentation (version 3.0), 2026. https://sdith.org/docs/sdith-v3.0.pdf.

[ABC+26] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, Quynh Dang, Thinh Hung Dang, John Kelsey, 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 8610, 2026. https://doi.org/10.6028/NIST.IR.8610.

[BBB+25] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v2: Algorithm specifications (version 2.0), 2025. https://faest.info/faest-spec-v2.0.pdf.

[BBB+26] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v3: Algorithm specifications (version 3.0), 2026. https://faest.info/faest-spec-v3.0.pdf.

[BBFR25] Ryad Benadjila, Charles Bouillaguet, Thibauld Feneuil, and Matthieu Rivain. MQOM: MQ on my mind, algorithm specifications and supporting documentation (version 2.1), 2025. https://mqom.org/docs/mqom-v2.1.pdf.

[BCH+20] Shivam Bhasin, Anupam Chattopadhyay, Annelie Heuser, Dirmanto Jap, Stjepan Picek, and Ritu Ranjan Shrivastwa. Mind the portability: A warriors guide through realistic profiled side-channel analysis. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2020. https://doi.org/10.14722/ndss.2020.24390.

[CFG+10] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *International Conference on Information and Communications Security (ICICS)*, LNCS 6476, pages 46–61. Springer, 2010. https://doi.org/10.1007/978-3-642-17650-0_5.

[CK14] Omar Choudary and Markus G. Kuhn. Template attacks on different devices. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 8622, pages 179–198. Springer, 2014. https://doi.org/10.1007/978-3-319-10175-0_13.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, pages 13–28. Springer, 2002. https://doi.org/10.1007/3-540-36400-5_3.

[CZLG21] Pei Cao, Chi Zhang, Xiangjun Lu, and Dawu Gu. Cross-device profiled side-channel attack with unsupervised domain adaptation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):27–56, 2021. https://doi.org/10.46586/tches.v2021.i4.27-56.

[FHK+20] 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, specification v1.2, 2020. https://falcon-sign.info/falcon.pdf.

[FIP24a] Module-lattice-based digital signature standard. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 204, 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf.

[FIP24b] Module-lattice-based key-encapsulation mechanism standard. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 203, 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf.

[FIP24c] Stateless hash-based digital signature standard. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 205, 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf.

[GAA+25] Philippe Gaborit, Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jérôme Lacan, Jean-Marc Robert, Pascal Véron, Paulo L. Barreto, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Jan Richter-Brokmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. Hamming Quasi-Cyclic (HQC), 2025. https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.

[GAG+25] Julie Godard, Nicolas Aragon, Philippe Gaborit, Antoine Loiseau, and Julien Maillard. Single trace side-channel attack on the MPC-in-the-Head framework. In *Post-Quantum Cryptography (PQCrypto)*, LNCS 15578, pages 267–293. Springer, 2025. https://doi.org/10.1007/978-3-031-86602-9_10.

[GAG+26] Julie Godard, Nicolas Aragon, Philippe Gaborit, Antoine Loiseau, and Julien Maillard. Non-supervised SASCA on the VOLEitH framework. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2026(2):468–494, 2026. https://doi.org/10.46586/tches.v2026.i2.468-494.

[GMGL24] Guillaume Goy, Julien Maillard, Philippe Gaborit, and Antoine Loiseau. Single trace HQC shared key recovery with SASCA. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2024(2):64–87, 2024. https://doi.org/10.46586/tches.v2024.i2.64-87.

[HWZY26] Zhuo Huang, Weijia Wang, Xiaogang Zhou, and Yu Yu. DPA-style attacks on HQC. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2026(2):928–952, 2026. https://doi.org/10.46586/tches.v2026.i2.928-952.

[JD26a] Sönke Jendral and Elena Dubrova. Fault attacks on VOLEitH signature schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2026(1):225–249, 2026. https://doi.org/10.46586/tches.v2026.i1.225-249.

[JD26b] Sönke Jendral and Elena Dubrova. Side-channel attacks on VOLEitH signature schemes: Breaking masked FAEST. *IACR Communications in Cryptology (CiC)*, 2(4):24, 2026. https://doi.org/10.62056/aebngyl7s.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_25.

[KKPY24] Matthias J. Kannwischer, Markus Krausz, Richard Petri, and Shang-Yi Yang. pqm4: Benchmarking NIST additional Post-Quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Paper 2024/112, 2024. https://eprint.iacr.org/2024/112.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO)*, LNCS 1109, pages 104–113. Springer, 1996. https://doi.org/10.1007/3-540-68697-5_9.

[MBTL13] David P. Montminy, Rusty O. Baldwin, Michael A. Temple, and Eric D. Laspe. Improving cross-device attacks using zero-mean unit-variance normalization. *Journal of Cryptographic Engineering (JCEN)*, 3(2):99–110, 2013. https://doi.org/10.1007/s13389-012-0038-y.

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 1717, pages 144–157. Springer, 1999. https://doi.org/10.1007/3-540-48059-5_14.

[MOP07] Stefan Mangard, Maria Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. https://doi.org/10.1007/978-0-387-38162-6.

[Nat22] National Institute of Standards and Technology (NIST). Request for additional digital signature schemes for the Post-Quantum cryptography standardization process, 2022. https://csrc.nist.gov/news/2022/request-additional-pqc-digital-signature-schemes.

[Nat24] National Institute of Standards and Technology (NIST). NIST announces 14 candidates to advance to the second round of the additional digital signatures for the Post-Quantum cryptography standardization process, 2024. https://csrc.nist.gov/news/2024/pqc-digital-signature-second-round-announcement.

[Nat26] National Institute of Standards and Technology (NIST). Nine candidates advance to the third round of the additional digital signatures for the PQC standardization process, 2026. https://csrc.nist.gov/News/2026/nist-advances-9-candidates-to-the-3rd-round-of-pqc.

[RBSC22] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. On exploiting message leakage in (few) NIST PQC candidates for practical message recovery attacks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 17:684–699, 2022. https://doi.org/10.1109/TIFS.2021.3139268.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3659, pages 30–46. Springer, 2005. https://doi.org/10.1007/11545262_3.