Side-Channel and Fault Injection Attacks on VOLEitH Signature Schemes: A Case Study of Masked FAEST

Sönke Jendral and Elena Dubrova

KTH Royal Institute of Technology, Stockholm, Sweden
{jendral,dubrova}@kth.se

Abstract. Ongoing efforts to transition to post-quantum secure public-key cryptosystems have created the need for algorithms with a variety of performance characteristics and security assumptions. Among the candidates in NIST's post-quantum standardisation process for additional digital signatures is FAEST, a Vector Oblivious Linear Evaluation in-the-Head (VOLEitH)-based scheme, whose security relies on the one-wayness of the Advanced Encryption Standard (AES). The VOLEitH paradigm enables competitive performance and signature sizes under conservative security assumptions. However, since it was introduced recently, in 2023, its resistance to physical attacks has not yet been analysed. In this paper, we present the first security analysis of VOLEitH-based signature schemes in the context of side-channel and fault injection attacks. We demonstrate four practical attacks on a masked implementation of FAEST in ARM Cortex-M4 capable of recovering the full secret key with high probability (greater than 0.87) from a single signature. These attacks exploit vulnerabilities of components specific to VOLEitH schemes and FAEST, such as the all-but-one vector commitments, VOLE generation, and AES proof generation. Finally, we propose countermeasures to mitigate these attacks and enhance the physical security of VOLEitH-based signature schemes.

Keywords: Side-channel analysis · Fault injection · FAEST · Vector Oblivious Linear Evaluation in-the-Head (VOLEitH) · Post-quantum digital signature · Key recovery attack

1 Introduction

In 2024, the National Institute of Standards and Technology (NIST) published the first set of standards from its main competition for post-quantum cryptographic (PQC) algorithms for key-encapsulation mechanism ML-KEM [30] and digital signature schemes ML-DSA [29] and SLH-DSA [32]. In an effort to enable the use of PQC algorithms in a greater variety of use cases, NIST launched a second competition, focusing on additional digital signature schemes based on different underlying problems and with different performance characteristics [28].

Among the submissions selected by NIST [31] for the second round in this competition is FAEST, a digital signature scheme based on the Vector Oblivious

Evaluation in-the-Head (VOLEitH) paradigm introduced by Baum et al. [8] in 2023. FAEST is designed to be existentially unforgeable under chosen message attacks (EUF-CMA) in the (quantum) random oracle model [9]. EUF-CMA security means that an adversary with access to the public key and a signing oracle cannot generate a valid signature for a new message.

The idea behind VOLEitH signature schemes is to use two-party VOLE correlations to facilitate a zero-knowledge proof of knowledge (ZKPoK) of the computation of a one-way function (OWF). The benefit of this approach is that, assuming the correctness of the ZKPoK, the security of the signature scheme relies primarily on the one-wayness of the OWF. In the case of FAEST, the OWF is the Advanced Encryption Standard (AES) and the ZKPoK proves knowledge of the AES state for each encryption round under a public plaintext and secret key, thus providing strong security guarantees based only on symmetric primitives, while offering competitive signing and verification performance and signature sizes.

However, the VOLEitH paradigm was introduced recently and FAEST is the only submission in the NIST competition for additional digital signatures making use of this paradigm. Thus, FAEST has not received as much attention as other submissions based on more mature approaches, including with respect to the resistance to physical attacks, such a side-channel analysis and fault injection. Previous attacks on PQC algorithms [3,13,25,5,6,26] have shown that even if algorithms are theoretically secure, their implementations might be vulnerable to physical attacks. To allow developers to design and implement effective countermeasures to such attacks, it is therefore important to identify the types of physical attacks that can be conducted on FAEST.

Contributions: In this paper, we present the first physical security analysis of VOLEitH-based signature schemes. The main contributions are:

i) We provide a high-level evaluation of the susceptibility of VOLEitH-based signature schemes to side-channel and fault injection attacks, including the analysis of the underlying VOLE correlations and all-but-one vector commitments that establish these correlations.
ii) We present two single-trace deep learning-assisted power-based side-channel attacks and two single-execution first-order voltage fault injection attacks on the masked implementation of FAEST by Degn, Eilath and Nielsen [14].
iii) We validate the feasibility of the attacks on an ARM Cortex-M4 processor and empirically evaluate their success rate.
iv) We propose countermeasures to mitigate these attacks.

The presented physical security analysis helps identify key vulnerable components in VOLEitH-based signature schemes. Our experimental results show that, for side-channel attacks, such vulnerable components are operations involving witness bits and VOLE tags, while for fault injection attacks, they are operations related to the computation of VOLE and vector commitments. These results are expected to offer guidance to implementers of VOLEitH-based signature schemes and designers of countermeasures, helping them focus their efforts in the right direction.

It is also worth noting that, as the FAEST implementation [14] is still under active development, not all its components are protected by masking yet. To ensure the long-term relevance of the presented analysis, our side-channel attacks target components already protected by masking.

Organisation of the paper: The rest of this paper is organised as follows. Section 2 describes previous work. Section 3 provides background information on the VOLEitH paradigm and the FAEST algorithm. Section 4 outlines the experimental setup. Section 5 discusses the susceptibility of VOLEitH-based signature algorithms to physical attacks. Sections 6 and 7 present the side-channel attacks. Sections 8 and 9 present the fault injection attacks. Section 10 summarises the experimental results. Section 11 discusses potential countermeasures against the attacks. Section 12 concludes the paper.

2 Previous work

To the best of our knowledge, the presented work is the first to evaluate the resistance of FAEST (and the underlying VOLEitH paradigm) to physical attacks, such as side-channel analysis and fault injection. Previous work has focused on the related Multiparty Computation in the Head (MPCitH) approach [22], which differs from VOLEitH (and FAEST) in a number of aspects. This section provides an overview of previous attacks and highlights the differences to the attacks presented in this paper.

Godard et al. [20] describe a single-trace side-channel attack on the reference implementation of the Syndrome Decoding in the Head (SDitH) signature scheme [1]. Specifically, they target the Galois field multiplications during the evaluation of the shares of a secret polynomial S for each of the parties and apply a templated soft-analytical side-channel attack. They practically evaluate their attack on a STM32F407 (ARM Cortex-M4), and show that a number of coefficients of S can be recovered and used to recover the secret key with an enumeration complexity of 70–70^2 for all security levels from a single trace. As a countermeasure, they propose the use of shuffling.

It is unclear how their attack would translate to FAEST, as the finite field multiplications in FAEST are mainly performed in the larger finite field $\mathbb{F}_{2^{128}}$, rather than $\mathbb{F}_{2^8}$ as in SDitH, and thus use a different multiplication algorithm. Note that, while Feneuil and Rivain [17] suggest that the VOLEitH paradigm can be seen as a special case of a variant of the Threshold Computation in-the-Head (TCitH) paradigm with GGM trees, the variant of SDitH that Godard et al. consider uses a different variant of TCitH that does not have such a relation to VOLEitH. The attack they present focuses on the underlying OWF, rather than the broader paradigm.

Gellersen et al. [19] describe two differential power-based side-channel attacks on Picnic [12]. Both attacks involve the LowMC block cipher [2], which is used as the OWF in Picnic. The first attack exploits the secret sharing prior to the multiparty LowMC computation, which leaks the Hamming weight of the unopened key shares. The second attack targets the computation of the S-boxes, which similarly reveals information about the unopened share. They practically evaluate their attacks on a NXP MK66FN2M0VMD18 (ARM Cortex-M4) and demonstrate that it is possible to recover the full secret key from fewer than 1000 traces, obtainable from fewer than 30 signatures. Seker et al. [36] propose a modified variant of Picnic that uses several masking gadgets to prevent these types of attacks. As FAEST is not based on multiparty computation, and uses AES instead of LowMC as the underlying OWF primitive, neither the attacks [19,36] nor the countermeasures are directly applicable.

Aranha et al. [4] also describe two side-channel attacks against Picnic. The first attack applies a similar idea to the previous attacks to the Picnic variant with preprocessing (Picnic3). They observe that information from an offline preprocessing step can be combined with side-channel information about a masked witness wire value to recover a secret key bit. They experimentally validate this attack on a STM32F407G-DISC1 (ARM Cortex-M4) and show that leakage becomes clear after 2,725 traces (and thus signatures). The second attack is an extension of the previous attacks [19,36] to Picnic3. They also propose a masking scheme that can prevent their attacks. Again, the attacks and countermeasures are not directly applicable to FAEST, due to differences in the underlying paradigms. However, the masked implementation of SHAKE which they present is also used in the masked implementation of FAEST [14] targeted in this paper, though none of our attacks focus on this component.

3 Background

3.1 Notation

In this paper, we follow the notation of [9]. Intervals are denoted as $[a..b] = \{a, \dots, b-1, b\}$ and $[a..b) = \{a, \dots, b-1\}$, vectors in bold font $\mathbf{x}$ and matrices in capitals $\mathbf{X}$. Indexing is denoted as $\mathbf{x}[i]$ for the i-th element of a vector, $\mathbf{X}|_i$ for the i-th row of a matrix, while $a \parallel b$ denotes the concatenation of a and b, and $\mathbf{x}_{[a..b]}$ the elements at indices a through b of a vector or list. Additionally, we use the notation $x[\![j]\!]$ to refer to the j-th bit of a field element x.

3.2 VOLEitH signature schemes

The VOLEitH paradigm for constructing secure digital signature schemes from VOLE-based ZKPoKs was first introduced by Baum et al. [8]. This section provides an overview of the main components of the paradigm, primarily based on the FAEST specification [9].

Fundamentally, the VOLEitH approach involves a number of VOLE correlations

$$q_i = \Delta \cdot u_i - v_i \quad \text{with } i \in [0..l) \tag{1}$$

over a finite field $\mathbb{F}_{2^k}$, where $q_i \in \mathbb{F}_{2^k}$ is a public *VOLE key*, $u_i \in \mathbb{F}_2$ is a secret random bit, $\Delta \in \mathbb{F}_{2^k}$ is the public *global key*, and $v_i \in \mathbb{F}_{2^k}$ is a secret *VOLE tag*.

These correlations are computed using all-but-one vector commitments based on a construction by Goldreich, Goldwasser, and Micali [21], which is referred to as a GGM tree. In the construction, a length-doubling pseudorandom generator (PRG) is used to compute a tree with pseudorandom seeds $(\mathsf{sd}_i)_{i\in[0..N)}$ as leaves, such that all-but-one of these seeds can be efficiently revealed (i.e. with logarithmic communication cost, by revealing full subtrees instead of individual leaves) through opening the vector commitment. From the seeds, a set of random bit strings $(\mathbf{r}_i)_{i\in[0..N)}$ are derived, again using a PRG. This allows computing a vector $\mathbf{u} \in \mathbb{F}_2^l$ of random bits and a vector $\mathbf{v} \in \mathbb{F}_{2^k}^l$ of VOLE tags as

$$\mathbf{u} = \sum_{i=0\in\mathbb{F}_{2^k}}^{N-1} \mathbf{r}_i, \quad \mathbf{v} = \sum_{i=0\in\mathbb{F}_{2^k}}^{N-1} i \cdot \mathbf{r}_i$$

and, by revealing seeds sd_i for all $i \neq \Delta$ for random global key $\Delta \in \mathbb{F}_{2^k}$, allows computing a vector $\mathbf{q} \in \mathbb{F}_{2^k}^l$ of VOLE keys as

$$\begin{aligned}\mathbf{q} &= \sum_{i=0\in\mathbb{F}_{2^k}}^{N-1} (\Delta - i) \cdot \mathbf{r}_i \\ &= \Delta \cdot \sum_{i=0\in\mathbb{F}_{2^k}}^{N-1} \mathbf{r}_i - \sum_{i=0\in\mathbb{F}_{2^k}}^{N-1} i \cdot \mathbf{r}_i \\ &= \Delta \cdot \mathbf{u} - \mathbf{v}\end{aligned}$$

for the final correlation.

A useful property of the VOLE correlations is that they are linearly homomorphic, thus one can perform arithmetic with them. For example, given two VOLE correlations $(\Delta, q_0), (u_0, v_0)$ and $(\Delta, q_1), (u_1, v_1)$ for a global key Δ, their sum can be represented by the VOLE correlation $(\Delta, q_0 + q_1), (u_0 + u_1, v_0 + v_1)$. This approach can also be extended to constant values and other linear operations, see [9].

To be able to use such VOLE correlations to construct a proof with specific witness bit values $\mathbf{w} \in \mathbb{F}_2^l$ instead of the random bits $\mathbf{u}$, the prover can compute a vector $\mathbf{d} \coloneqq \mathbf{w} - \mathbf{u}$, such that

$$\mathbf{q}' \coloneqq \mathbf{q} + \Delta \cdot \mathbf{d} = \Delta \cdot (\mathbf{u} + \mathbf{d}) - \mathbf{v} = \Delta \cdot \mathbf{w} - \mathbf{v} \tag{2}$$

can be used to form valid VOLE correlations for $\mathbf{w}$ instead [9].

As a simplification, the FAEST specification [9] and a number of algorithms in this paper occasionally interpret the vectors of the form $\mathbb{F}_{2^k}^l$ given in Eq. 2 instead as matrices of the form $\{0,1\}^{l\times k}$. This yields the notions of a VOLE key matrix $\mathbf{Q}$ and a VOLE tag matrix $\mathbf{V}$, where each row corresponds to a single correlation of the form in Eq. 1, i.e. where the columns correspond to the bit decomposition of the field element. Clearly both representations are interchangeable.

To simplify the explanations given in later parts of this paper, we use this matrix representation to introduce the notion of active columns. Specifically, we say column i of the VOLE tag or key matrix is *active* if the i-th bit of the global key Δ, $\Delta[\![i]\!] = 1$. From Eq. 2, it follows that the structure of the VOLE correlations is such that the active columns are the only columns where the VOLE key contains information about the witness bit. This notion is useful when attempting to recover information about the witness from the VOLE tag and key matrices.

As a further optimisation, Baum et al. [8] extend a method described for the SoftSpokenOT protocol of Roy [35] that allows constructing a VOLE for a larger field from multiple VOLEs over smaller fields. As the precise details of this construction are not relevant here, we instead refer to [8].

However, in practical terms, this optimisation means that τ individual VOLEs over smaller fields $\mathbb{F}_{2^{k_0}}$ and $\mathbb{F}_{2^{k_1}}$ are combined together to form a VOLE over the larger field $\mathbb{F}_{2^k}$. This is only possible if all τ VOLEs correspond to the same random bits $\mathbf{u}$, so prior to the adjustment for the witness bit values in Eq. 2, a number of correction values $\mathbf{c}_i = \mathbf{u}_i - \mathbf{u}_0$ are computed for $i \in [1..\tau)$ by the prover (and later applied by the verifier in the same way as before), to ensure that all VOLE correlations use the same set of random bits (i.e. the random bits $\mathbf{u}_0$ computed for the first VOLE correlation). Additionally, this requires a consistency check to ensure that the prover does not cheat when computing the correction values, which we again do not show here and which can instead be found in the specification [9].

Based on the VOLE correlations, Baum et al. [8] then use the interactive QuickSilver ZKPoK protocol [40] to prove an arbitrary arithmetic circuit. Concretely, the specification [9] describes the computation of a linear gate a, b, c as $w_\gamma := a \cdot w_\alpha + b \cdot w_\beta + c$ and $v_\gamma := a \cdot v_\alpha + b \cdot v_\beta$ by the prover and $q_\gamma := a \cdot q_\alpha + b \cdot q_\beta + c \cdot \Delta$ by the verifier, which does not require any communication. Further, it describes the computation of a multiplication gate a, b, c as $w_\gamma := w_\alpha \cdot w_\beta$ and $d_\gamma = w_\gamma - u_i$ by the prover (i.e. the prover commits to the output bit w_γ), and $q_\gamma := q_\gamma + d_\gamma \cdot \Delta$ by the verifier (i.e. the verifier verifies the validity of the output bit). The verifier is then able to check the multiplication by computing and checking

$$\begin{aligned} b_\gamma &:= q_\alpha \cdot q_\beta - q_\gamma \cdot \Delta \\ &= v_\alpha \cdot v_\beta + (w_\alpha \cdot v_\beta + w_\beta \cdot v_\alpha - v_\gamma) \cdot \Delta + \underbrace{(w_\alpha \cdot w_\beta - w_\gamma) \cdot \Delta^2}_{\text{Vanishes if prover is honest}} \\ &\overset{!}{=} a_0 + a_1 \cdot \Delta \end{aligned} \tag{3}$$

using values $a_0 := v_\alpha \cdot v_\beta$ and $a_1 := w_\alpha \cdot v_\beta + w_\beta \cdot v_\alpha - v_\gamma$ computed by the prover. Note that a_0 and a_1 are not revealed to the verifier directly, but are instead combined across all multiplication gates and then masked using another VOLE, as described in [9].

Using this approach, it is possible to prove an arbitrary arithmetic circuit corresponding to the evaluation of a OWF. A potential problem for using such a

Table 1. Overview of the main parameter sets for FAEST. Values $\lambda = \{128, 192, 256\}$ correspond to security levels 1, 3, and 5, respectively. Each security level features both a small (s) and fast (f) variant. For details, see [9].

Scheme	λ	l	τ	τ_0	τ_1	k_0	k_1	pk (bytes)	sig (bytes)
FAEST-128s	128	1600	11	7	4	12	11	32	5006
FAEST-128f	128	1600	16	0	16	8	8	32	6336
FAEST-192s	192	3264	16	0	16	12	12	64	12744
FAEST-192f	192	3264	24	0	24	8	8	64	16792
FAEST-256s	256	4000	22	14	8	12	11	64	22100
FAEST-256f	256	4000	32	0	32	8	8	64	28400

construction in the context of a signature scheme is that it provides a designated verifier proof, i.e. a proof which requires interaction between the prover (signer) and verifier to establish the VOLE correlations from the all-but-one vector commitments [8]. The central insight by Baum et al. [8] is that it is not necessary that the prover is unable to learn the global key Δ that the verifier chooses. Instead, it is sufficient to ensure that the prover is committed to their proof prior to learning the value of Δ, as this still prevents the prover from cheating in the proof. Then, by applying a Fiat-Shamir transform [18], the prover is able to simulate the protocol in their head and to compute an opening of the all-but-one vector commitment, such that the protocol can be used non-interactively, as a signature scheme.

3.3 FAEST algorithm

Based on the VOLEitH paradigm described in the previous section, Baum et al. [8] construct the FAEST signature scheme which uses AES as the underlying OWF.

In FAEST, the witness bits that the signer proves knowledge of using the QuickSilver proof are the secret key and the state of each round of the AES encryption. More specifically, the so-called *extended witness* contains the AES cipher key and, for each round of the key schedule, the bits that form the output of the SubWord operation, and for each round of the encryption, the bits that form the output of the ShiftRows operation.

Then, for the key schedule and for the encryption, the algorithm performs two passes each: a forward pass, which computes the bits (or their corresponding VOLE tags or keys) that form the input for each S-box, and a backward pass, which computes the bits (or their corresponding VOLE tags or keys) that form the output for each S-box, by applying or reversing certain AES steps. For example, the backward pass for the encryption takes the bits of the extended witness (or their corresponding VOLE tags or keys) that correspond to the output of the ShiftRows operation and applies an inverse ShiftRows operation to derive the S-box output.

Algorithm 1 FAEST.KeyGen() [9]

```
Output: Public key pk, secret key sk
1: while true do
2:     $\mathbf{x} \leftarrow \{0,1\}^{\beta \cdot 128}$                    ▷ OWF input
3:     $\mathbf{k} \leftarrow \{0,1\}^{\lambda}$                        ▷ OWF key
4:     $\mathbf{y} := F_{\mathbf{k}}(\mathbf{x})$                        ▷ OWF output
5:     if no SubBytes or SubWords has input byte {00} then
6:         return $(\mathsf{sk} := \mathbf{k}, \mathsf{pk} := (\mathbf{x}, \mathbf{y}))$
```

These inputs and outputs are then used to compute or verify multiplicative constraints, using the approach described in Eq. 3. Specifically, for S-box input a and output b, the constraint ensures that $a \cdot b = 1$, which works because the AES S-box computes a field inversion, meaning that the output b is the inverse of a. This has an added benefit of simplifying the multiplicative constraints because the values w_γ and d_γ can be omitted, as the result is constant. If during the verification all multiplicative constraints are correct, the signature is valid.

In the following we provide an overview of the key generation, signing, and verification algorithms in FAEST. As the signing and verification algorithms each make use of a larger number of subprocedures, we are not able to show these algorithms in their entirety. For a full description of all involved algorithms, we instead refer to the specification [9]. An overview of the main parameters of the algorithm is shown in Tab. 1. This paper focuses on FAEST-128f, though other variants can be approached similarly.

Key generation (Alg. 1) The key generation algorithm computes a random OWF input and OWF key. It then computes the output of the OWF under this input and key. As it is not possible to prove a multiplication constraint for an input that is zero using the degree-2 constraints used in FAEST[1], the algorithm ensures that no S-box in the key schedule or encryption has an input byte {00}, otherwise a new input and key is chosen. Once a suitable key and input are found, the secret key consists of the OWF key (i.e. the cipher key), and the public key consists of the OWF input and output.

Signing (Alg. 2) The signing algorithm first binds the public key to the message by deriving the value to sign, μ. This string is used alongside the secret key and additional randomness to derive the VOLEs. In addition to the VOLE random bits $\mathbf{u}$ and VOLE tag matrix $\mathbf{V}$, the VOLE commitment also computes a hash h_{com}, full decommitments decom_i for all τ parallel VOLEs and correction values $\mathbf{c}_i$ to fix all VOLE correlations to the same random bits. The first challenge chall_1 and the responses $\tilde{\mathbf{u}}$ and h_V are used to ensure the consistency of

[1] Baum et al. [7] later showed that multiplications with zero inputs can be proven using higher-degree constraints. However, FAEST has not been updated to take advantage of such constraints.

Algorithm 2 FAEST.Sign(sk, M) [9]

```
Input: Secret key sk, message $M$
Output: Signature $\sigma$
1: $\rho \leftarrow \{0,1\}^\lambda$                    ▷ Deterministic variant: $\rho \leftarrow \{0\}^\lambda$
2: $\mu := \mathsf{H}_1(\mathsf{pk} \parallel M)$
3: $(r, \mathsf{iv}) := \mathsf{H}_3(\mathsf{sk} \parallel \mu \parallel \rho)$
4: $(h_{\mathsf{com}}, (\mathsf{decom}_i)_{i\in[0,\tau)}, (\mathbf{c}_i)_{i\in[1,\tau)}, \mathbf{u}, \mathbf{V}) := \mathsf{FAEST.VOLECommit}(r, \mathsf{iv})$
5: $\mathsf{chall}_1 := \mathsf{H}_2^1(\mu \parallel h_{\mathsf{com}} \parallel \mathbf{c}_1 \parallel \cdots \parallel \mathbf{c}_{\tau-1} \parallel \mathsf{iv})$
6: $\tilde{\mathbf{u}} := \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{u})$
7: $\tilde{\mathbf{V}} := \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{V})$
8: $h_V := \mathsf{H}_1(\tilde{\mathbf{V}})$
9: $\mathbf{w} := \mathsf{AES.ExtendWitness}(\mathsf{sk}, \mathsf{pk})$
10: $\mathbf{d} := \mathbf{w} \oplus \mathbf{u}_{[0..l)}$
11: $\mathsf{chall}_2 := \mathsf{H}_2^2(\mathsf{chall}_1 \parallel \tilde{\mathbf{u}} \parallel h_V \parallel \mathbf{d})$
12: $\mathbf{u} := \mathbf{u}_{[0..l+\lambda)}, \mathbf{V} := \mathbf{V}|_{[0..l+\lambda)}$   ▷ Impl. may transpose internal representation of $\mathbf{V}$
13: $(\tilde{a}, \tilde{b}) := \mathsf{FAEST.AES.AESProve}(\mathbf{w}, \mathbf{u}, \mathbf{V}, \mathsf{pk}, \mathsf{chall}_2)$
14: $\mathsf{chall}_3 := \mathsf{H}_2^3(\mathsf{chall}_2 \parallel \tilde{a} \parallel \tilde{b})$
15: for $i \in [0..\tau)$ do
16:     $\mathbf{s}_i := \mathsf{ChalDec}(\mathsf{chall}_3, i)$
17:     $(\mathsf{pdecom}_i) := \mathsf{VC.Open}(\mathsf{decom}_i, \mathbf{s}_i)$
18: return $\sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \tilde{\mathbf{u}}, \mathbf{d}, \tilde{a}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{chall}_3, \mathsf{iv})$
```

the VOLEs. The algorithm then computes the extended witness $\mathbf{w}$ by running the AES encryption and storing certain bits, and, from it, the masked witness $\mathbf{d}$, before deriving a second challenge chall_2 used to commit to the witness. It then computes the QuickSilver proof of the encryption and uses it to derive the final challenge chall_3, which reveals the global key. Finally, using the global key, the vector commitments are opened to partial decommitments that allow reconstructing all-but-one vectors. The signature consists of the correction values $\mathbf{c}_i$, the VOLE hash $\tilde{\mathbf{u}}$, the masked witness bits $\mathbf{d}$, part of the QuickSilver proof $\tilde{a}$, the partial decommitments pdecom_i, the third challenge chall_3, and an initialisation vector iv.

Verification (Alg. 3) The verification algorithm extracts the individual components of the signature. It then recomputes the signed value μ from the public key and the message, before reconstructing the VOLEs using the third challenge, the partial decommitments, and the initialisation vector. This allows the first challenge chall_1 to be recomputed. Next, the VOLE key matrix $\mathbf{Q}$ is derived by applying the correction values $\mathbf{c}_i$. To recompute the response h_V, the VOLE key column hashes must be adjusted to the witness bits by adding the VOLE hash $\tilde{\mathbf{u}}$ to the active columns. Then, the second challenge chall_2 and, from it, the second part of the QuickSilver proof, $\tilde{b}$, can be recomputed. Finally, the third challenge chall_3' can be recomputed. If the recomputed value matches the value in the signature, the signature is valid.

Algorithm 3 $\mathsf{FAEST.Verify}(M, \mathsf{pk}, \sigma)$ [9]

```
Input: Message $M$, public key $\mathsf{pk}$, signature $\sigma$
Output: Boolean
1: $((\mathbf{c}_i)_{i\in[1..\tau)}, \tilde{\mathbf{u}}, \mathbf{d}, \tilde{a}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{chall}_3, \mathsf{iv}) = \sigma$
2: $\mu \coloneqq \mathsf{H}_1(\mathsf{pk} \parallel M)$
3: $(h_{\mathsf{com}}, \mathbf{Q}'_0, \ldots, \mathbf{Q}'_{\tau-1}) \coloneqq \mathsf{FAEST.VOLEReconstruct}(\mathsf{chall}_3, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv})$
4: $\mathsf{chall}_1 \coloneqq \mathsf{H}_2^1(\mu \parallel h_{\mathsf{com}} \parallel \mathbf{c}_1 \parallel \cdots \parallel \mathbf{c}_{\tau-1} \parallel \mathsf{iv})$
5: $\mathbf{Q}_0 \coloneqq \mathbf{Q}'_0$
6: for $i \in [0..\tau)$ do
7:     $b \coloneqq 0$ if $i < \tau_0$ else $1$
8:     $(\delta_0, \ldots, \delta_{k_b-1}) \coloneqq \mathsf{ChalDec}(\mathsf{chall}_3, i)$
9:     $\tilde{\mathbf{D}}_i \coloneqq [\delta_0 \cdot \tilde{\mathbf{u}} \cdots \delta_{k_b-1} \cdot \tilde{\mathbf{u}}]$
10:    if $i > 0$ then
11:        $\mathbf{Q}_i \coloneqq \mathbf{Q}'_i \oplus [\delta_0 \cdot \mathbf{c}_i \cdots \delta_{k_b-1} \cdot \mathbf{c}_i]$
12: $\mathbf{Q} \coloneqq [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}]$
13: $\tilde{\mathbf{Q}} \coloneqq \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{Q})$
14: $h_V \coloneqq \mathsf{H}_1(\tilde{\mathbf{Q}} \oplus \left[\tilde{\mathbf{D}}_0 \cdots \tilde{\mathbf{D}}_{\tau-1}\right])$
15: $\mathsf{chall}_2 \coloneqq \mathsf{H}_2^2(\mathsf{chall}_1 \parallel \tilde{\mathbf{u}} \parallel h_V \parallel \mathbf{d})$
16: $\tilde{b} \coloneqq \mathsf{FAEST.AES.AESVerify}(\mathbf{d}, \mathbf{Q}|_{[0..l+\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}, \mathsf{pk})$
17: $\mathsf{chall}'_3 \coloneqq \mathsf{H}_2^3(\mathsf{chall}_2 \parallel \tilde{a} \parallel \tilde{b}, \lambda)$
18: return true if $\mathsf{chall}_3 = \mathsf{chall}'_3$ else false
```

4 Experimental setup

This section describes the equipment and the target implementation of FAEST used in the experiments.

4.1 Equipment

The target board is a CW308-STM32F4 that contains an ARM Cortex-M4 STM32F415RGT6 processor running at a frequency of 24 MHz. It is mounted on a CW313 adapter board. Power traces are captured and voltage fault injection is performed using a ChipWhisperer-Husky.

For triggering, ARM CoreSight DWT watchpoints are used, which avoids changing the assembly instructions or register allocation of the firmware. In a real attack, other trigger sources, such as communication with peripheral devices or the sum of absolute differences between the power consumption and a reference waveform could be used instead.

4.2 Target implementation

In our experiments, we use the masked FAEST implementation by Degn et al. [14], specifically commit `b2503db`, which was the most recent at the time of the experiments. At the time of writing, two further commits have introduced changes to the computation of the extended witness (which is not targeted in this paper), certain field operations, and parts of the vector commitment and VOLE operations. However, these changes do not address the attacks presented in this paper and we therefore expect the attacks to translate to the most recent version (commit `93e746d`) directly.

The implementation is compiled using `arm-none-eabi-gcc` with the highest optimization level `-O3`.

5 Physical attacks on VOLEitH signature schemes

This section describes the high-level considerations that apply to physical attacks (both side-channel analysis and fault injection) on VOLEitH-based schemes from an algorithmic perspective. Specifically, it explains how knowledge or control of certain internal variables can be utilised to recover the secret key of the signature algorithm.

Recall that the idea behind VOLEitH signature schemes is to prove the evaluation of an OWF for a known output. In the case of FAEST, the secret key is therefore simply an AES cipher key and the signature generation process is a ZKPoK of the encryption of a public plaintext under this key. The goal of an attacker is to recover the input to the OWF by means of a physical attack.

Fundamentally, there are two components of the signature generation that can be attacked. The first is the computation of the extended witness. In FAEST, this is the encryption of the public value under the cipher key where specific bits of the AES state are recorded as the witness for each round. In practice, this is implemented as a straightforward AES encryption and, as such, known attacks and countermeasures can be applied in this context, such as [27,34]. Furthermore, this component appears to be under active development in the implementation by Degn et al. [14], and future changes may affect in which way physical attacks can be applied to it, if at all. We therefore do not consider such attack points in this paper and leave analysis of this component of FAEST for future work.

In other VOLEitH schemes, such as MandaRain and KuMQuat [7], the computation of the extended witness is conceptually similar and involves recording the state of the Rain block cipher [15] (MandaRain) or a solution to a system of multivariate quadratic equations (KuMQuat). In all cases, information leakage about the extended witness may be sufficient to perform secret key recovery. However, as the extended witness does not change across multiple invocations of the signing procedure, it is also possible to interpret the computation of the extended witness as a key expansion or decoding step, as found in other algorithms, such as ML-DSA or MAYO [29,10]. Future algorithms may therefore choose to only perform this step once, which limits the window of opportunity for an attacker.

The second component is the ZKPoK of the evaluation of the OWF, particularly the computations involving the VOLE correlations that encode information about the witness. The idea is to recover a subset $\{w_i \mid i \in S\}$ of the witness bits and use them to recover the OWF input. The specific witness bits that must be recovered depend on the approach used to extract the OWF input from the witness. For FAEST, this means using knowledge of parts of the AES state to recover the cipher key. For example, the first 128 bits of the witness encode the cipher key directly, thus recovering all of these bits reveals the cipher key without further processing. The next 320 bits of the witness encode the S-box output of the key schedule. These bits can be combined to recover a full round key and thus reveal the cipher key with some additional processing. The remaining 1152 bits encode the S-box output of the individual AES encryption rounds and can also be used to recover the cipher key with additional processing. In all cases, the number of witness bits required to recover the cipher key for FAEST cannot be smaller than 128 bits, regardless of the chosen method. For other VOLEitH schemes, the process is similar.

To illustrate the secret key recovery from the VOLE correlations, let $S \subseteq [0..l)$ be a subset of indices for which the corresponding witness bits $\{w_i \mid i \in S\}$ should be recovered and assume that a method for deriving the secret key from the witness bits is known. The structure given by Eq. 2 now constrains the possible attacks that result in key recovery. Clearly, revealing witness bits directly (i.e. via side-channel analysis) is sufficient to perform key recovery, while fixing their values to known values (via a fault) cannot reveal new information. However, it is also possible to recover the witness bits indirectly, i.e. through components u_i or v_i, and then derive the key. For example, recovering components u_i or fixing their values to known values allows computing $w_i \coloneqq d_i + u_i$ from the public values d_i. Similarly, it is also possible to recover or fix components v_i to then compute $q'_i - v_i = \Delta \cdot w_i$ and recover w_i from the active columns. This means that all components of the VOLE correlation are relevant in the context of physical attacks.

In this paper, we present four attacks that target different components of the VOLE correlations. Tab. 2 provides an overview. The first attack, the byte combine attack in Section 6, is a side-channel attack that directly recovers a subset of witness bits. The second attack, the VOLE transpose attack in Section 7, is a side-channel attack that recovers a subset of VOLE tags that can be used to recover the witness bits. The third attack, the counter increment skipping attack in Section 8, is a fault injection attack that recovers a subset of random bits or a subset of VOLE tags by fixing a variable used to derive them. The fourth attack, the VOLE conversion abort attack in Section 9, is a fault injection attack that fixes a subset of random bits or a subset of VOLE tags to known values directly.

6 Byte combine attack

The first attack, which we call the *byte combine* attack, exploits information leakage in the `bf128_byte_combine_bits` procedure. This procedure is called from the `aes_enc_forward_128_1_round` procedure, which is part of the VOLE protocol proof for the encryption. Recall that the forward pass derives the input values to the S-boxes for each of the AES rounds and their VOLE tags. In the initial round, these input values are the XOR of the plaintext and the initial round key, which is the cipher key. The `bf128_byte_combine_bits` procedure is

Table 2. Overview of presented attacks and their targeted components; SC = side-channel, FI = fault injection.

Type	VOLE correlation component		
	w_i	u_i	v_i
Recover	Byte combine SC attack	Counter increment skipping FI attack	VOLE transpose SC attack, Counter increment skipping FI attack
Fix	N/A[a]	VOLE conversion abort FI attack	VOLE conversion abort FI attack

[a] It is not possible to reveal new information about witness bits w_i by fixing their values to known values.

```
bf128_t bf128_mul_bit(bf128_t lhs, uint8_t rhs) {
  return bf128_and_64(lhs, -((uint64_t)rhs & 1));
}

bf128_t bf128_byte_combine_bits(uint8_t x) {
  bf128_t out = bf128_from_bit(x & 1);
  for (unsigned int i = 1; i < 8; ++i) {
    out = bf128_add(out, bf128_mul_bit(alpha[i - 1], (x >> i) & 1));
  }
  return out;
}
```

Listing 1: The C code of the `bf128_byte_combine_bits` and `bf128_mul_bit` procedures. Sections corresponding to the assembly code in Listing 2 are highlighted in color.

```
bf128_mul_bit:
  ...
  ldrb    r1, [sp, #20]
  ldr     r4, [sp, #8]
  sbfx    ip, r1, #0, #1
  and     r2, r2, ip
  str     r2, [r0, #0]
  ldr     r2, [sp, #12]
  and     r2, ip, r2
  ...

bf128_byte_combine_bits:
  ...
  str     r0, [sp, #20]
  and     r9, r1, #1
  mov     r8, r6
  ...
loop:
  subs    r1, r4, #1
  mov     r0, r5
  bl      get_alpha
  asr     r3, fp, r4
  and     r3, r3, #1
  str     r3, [sp, #8]
  ...
  bl      bf128_mul_bit
  ...
  bne     loop
  ...
```

Listing 2: Simplified excerpts of the assembly code of the `bf128_mul_bit` and `bf128_byte_combine_bits` procedures. Sections corresponding to the C code in Listing 1 are highlighted in color.

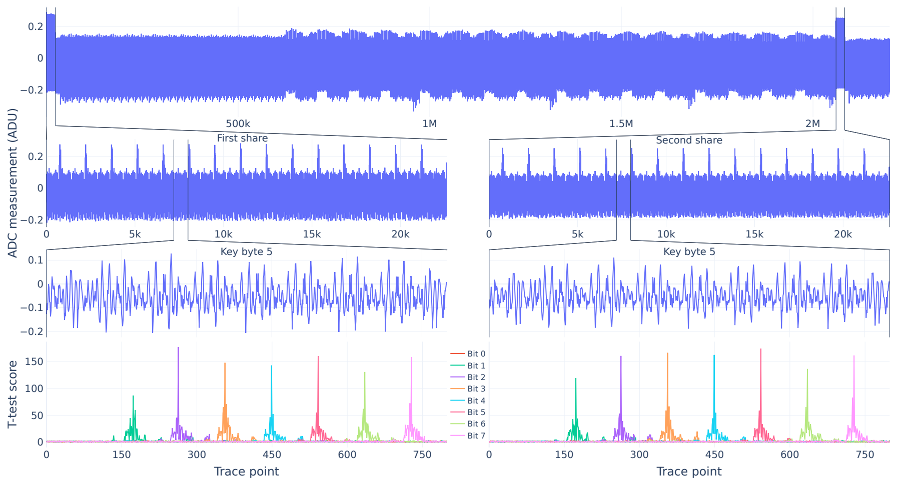

Fig. 1. Trace segmentation process and t-test results for the `bf128_byte_combine_bits` procedure, computed for 1000 traces. Note that bit 0 of each share does not produce leakage and is therefore not visible in the t-test results.

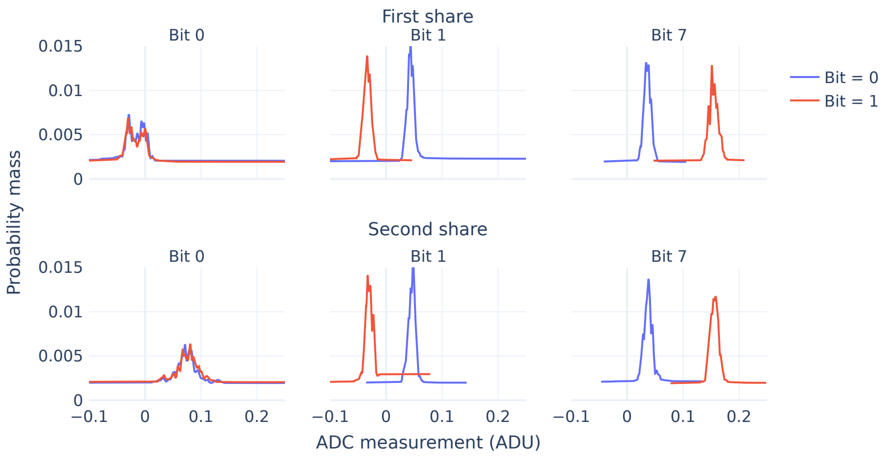

Fig. 2. Distributions of power consumption (average over 1000 traces) for bits 0, 1, and 7 at the point of maximum t-test score during the processing by `bf128_byte_combine_bits`. Note that bit 0 does not produce information leakage as the power consumption is independent of the bit value, while the distributions for bits 1 and 7 are clearly and similarly separated between the bit values.

used to lift individual bits of the key and plaintext into the field $\mathbb{F}_{128}$, before performing the addition (i.e. XOR) to derive the S-box input values.

In the implementation of Degn et al. [14], the `bf128_byte_combine_bits` procedure uses the `bf128_mul_bit` procedure, which is the main source of leakage. In the masked version, both of these procedures are implemented in hand-written assembly, though they appear to have been derived from the C code of the reference implementation with minor modifications. For the sake of readability, this C code is shown in Listing 1 instead, alongside excerpts of the actual assembly code in Listing 2.

For each invocation, the input to the `bf128_byte_combine_bits` procedure is a share byte of the first round key (i.e. the cipher key). The procedure first extracts bit 0 of this value in line 6 of Listing 1, which does not produce exploitable leakage. Line 15 of Listing 2 shows that the extracted bit is kept in a register, but not written to memory (including in the parts of the assembly code that are omitted in Listing 2), which explains the lack of leakage. In the context of the attack, the least significant bit of each byte of the key thus cannot be recovered using this method and these bits are instead enumerated (at a complexity of 2^{16}).

The remaining 7 bits of each byte are then extracted one-by-one from the byte in line 8 of Listing 1, and multiplied with constant values α_i, using the `bf128_mul_bit` procedure. In the C code, this multiplication is realised by first computing a 64-bit mask, whose bits are set if the extracted bit is set or cleared if the extracted bit is not set, and then performing a bitwise AND of the mask and the value to multiply. As the Cortex-M4 is a 32-bit processor, the assembly code in lines 6 and 9 of Listing 2 instead performs two bitwise AND operations on a 32-bit mask computed in line 5 using a `sbfx` (signed bit field extract) instruction. The issue here is obvious: The Hamming weight of the mask value is 0 when the bit value is 0 and 32 when the bit value is 1. This large difference in Hamming weight is clearly visible in the power consumption of the device, as shown in Fig. 2 and in the t-test results in Fig. 1. By observing the power consumption, it is therefore possible to extract the witness bits (and thus the cipher key) directly.

This type of leakage was first pointed out by Amiet et al. in their attack on the key encapsulation mechanism NewHope [3], and it is also found in other PQC algorithms [37,39,23]. In all cases, the behaviour being realised is a bit-multiplication with a secret bit, which is then leaked.

A similar attack can be performed on the `aes_key_schedule_128_masked` procedure, which applies the `bf128_byte_combine_bits` procedure to derive the S-box input from the previous round key, thereby revealing the round key. As this is essentially the same attack with the same underlying leakage, we did not pursue this direction further, though we believe that such an attack would succeed with a similar probability and that it can be mitigated through countermeasures in the underlying `bf128_byte_combine_bits` procedure.

Table 3. Neural network architectures used for the byte combine and VOLE transpose attacks.

Layer type	Output shape	
	Byte combine attack	VOLE transpose attack
Batch Normalization 1	340	512
Dense 1	128	128
Batch Normalization 2	128	128
ReLU	128	128
Dense 2	64	64
Batch Normalization 3	64	64
ReLU	64	64
Dense 3	32	32
Batch Normalization 4	32	32
ReLU	32	32
Dense 4	2	2

6.1 Trace preprocessing and neural network training

For the profiling phase, the dataset contains 560, 000 trace segments, obtained by capturing $t = 5000$ traces $\{T_1, \ldots, T_t\}$ for known secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_t$ selected at random and applying the cut-and-join technique of [33]. For each trace T_i, two segments T_i^{Ψ}, $\Psi \in \{0, 1\}$, are extracted, containing the processing of the first and second share. Each segment is then divided into 16 subsegments $T_i^{\Psi,\eta}$, $\eta \in [0..16)$, covering the processing of one byte of the cipher key. This process is illustrated in Fig. 1 for key byte 5. Finally, each subsegment is divided into 7 further segments $T_i^{\Psi,\eta,\kappa}$, $\kappa \in [1..8)$, covering the processing of one bit of the cipher key (recall that the first bit, bit 0, does not produce leakage), before concatenating the segments across both shares

$$\hat{T}_i^{\eta,\kappa} := T_i^{0,\eta,\kappa} \parallel T_i^{1,\eta,\kappa}.$$

To simplify the notation, in the following, we use $\hat{T}_i^{\phi}$ to refer to segment $\hat{T}_i^{\eta,\kappa}$ with $\phi = 8\eta + \kappa$, i.e. we flatten the byte and bit dimensions into a single dimension. We also use the set $R := \{i \in [0..128) \mid 8 \nmid i\}$ of bits that are recoverable using this method. Each segment $\hat{T}_i^{\phi}$ is labelled with the value of the bit of the cipher key being processed, $\mathrm{sk}_i[\![\phi]\!]$, which is the XOR of the bits being processed in each share.

The model is a multilayer perceptron (MLP) neural network with the architecture shown in Tab. 3. The network is of type $\mathcal{N} : \mathbb{R}^{340} \to \{0, 1\}$, where 340 is the number of samples in each segment. Note that Fig. 1 shows traces captured with one sample per clock cycle, but in this attack, traces were captured with four samples per clock cycle, hence the segments are longer than they appear. The network maps each segment $\hat{T}_i^{\phi}$ into a score vector $S_{i,\phi} = \mathcal{N}(\hat{T}_i^{\phi})$, such that

Algorithm 4 RecoverSecretKey($\sigma, (S_i)_{i \in R}$)

```
1: for $k \in \{0,1\}^{16}$ do
2:     for $i \in [0..\lambda)$ do
3:         if $i \bmod 8 \neq 0$ then
4:             $\mathsf{sk}[\![i]\!] \leftarrow \operatorname{argmax} S_i$
5:         else
6:             $\mathsf{sk}[\![i]\!] \leftarrow k[\![i/8]\!]$                ▷ Enumerate bit
7:     yield sk
```

```c
#define ptr_get_bit(v, i) (v[i / 8] >> (i % 8)) & 1
#define ptr_set_bit(d, v, i) d[i / 8] |= v << (i % 8)

bf128_t* get_vole_aes_128_share(vbb_t* vbb, int idx, int share) {
  memset(vbb->v_buf, 0, 16);
  uint8_t* src = share ? vbb->v_mask_cache : vbb->vole_cache;
  uint32_t strd = share ? 1872 : 1872 / 8;
  // Transpose on the fly into v_buf
  for (unsigned int col = 0; col != 128; ++col) {
    ptr_set_bit(vbb->v_buf, ptr_get_bit(src + col * strd, idx), col);
  }
  return (bf128_t*)vbb->v_buf;
}
```

Listing 3: Simplified C code of the `get_vole_aes_128_share` procedure. The section corresponding to the assembly code in Listing 4 is highlighted in colour.

$S_{i,\phi}[\alpha]$ represents the probability that bit $\mathrm{sk}_i[\![\phi]\!]$ of the secret key takes value $\alpha \in \{0,1\}$:

$$S_{i,\phi}[\alpha] = \Pr[\mathsf{sk}_i[\![\phi]\!] = \alpha].$$

The model is trained for a maximum of 100 epochs with early stopping at a patience of 15 using the Nadam optimiser with a learning rate of 0.01 and a numerical stability constant $\epsilon = 10^{-8}$. The batch size is 1024. Of the data 70% is used for training and 30% for validation.

6.2 Partial enumeration and secret key recovery method

As the leakage exploited in this attack directly reveals most bits of the secret key, the secret key recovery is straightforward and only requires enumeration of the missing bits, which can be done with a complexity of 2^{16}. Alg. 4 shows the main steps.

7 VOLE transpose attack

The second attack, which we call the *VOLE transpose* attack, exploits leakage during the transposition of the VOLE tag matrix in the `get_vole_aes_128_share`

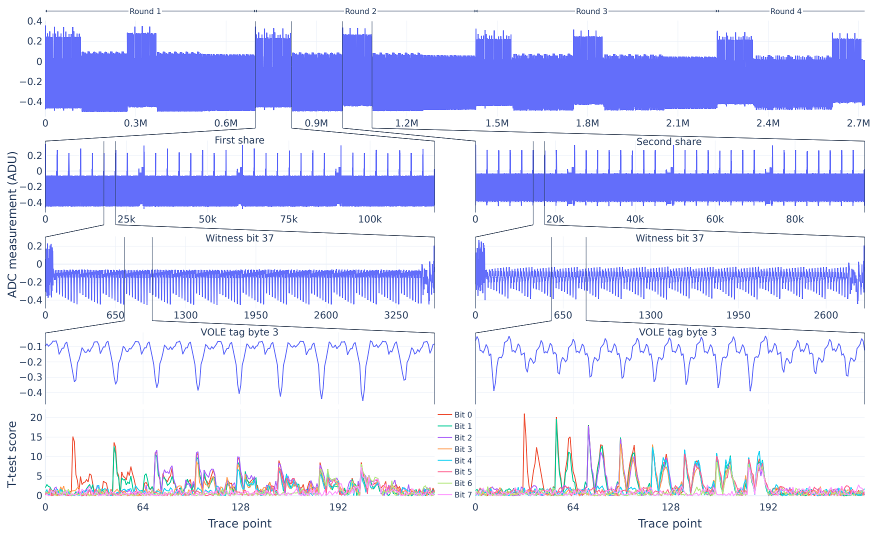

Fig. 3. Trace segmentation process and t-test results for the `get_vole_aes_128_share` procedure, computed for 1000 traces. Note the decreasing amount of leakage from bit 0 to bit 7 for each share and the overall lower amount of leakage for the first share, as indicated by the t-test results.

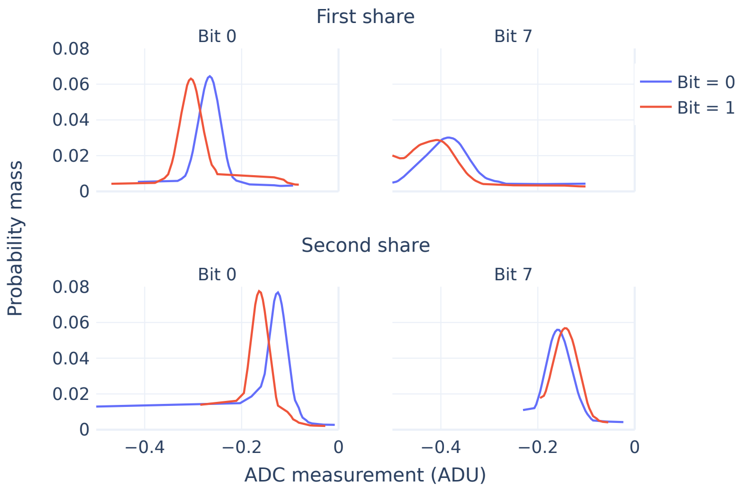

Fig. 4. Distributions of power consumption (average over 1000 traces) for bits 0 and 7 at the point of maximum t-test score during the processing by `get_vole_aes_128_share`. Note the greater separation of the distributions for bit 0 in both shares. The distributions for bit 7 of the first share are partially affected by clipping of the ADC occurring at −0.5 ADU, as a limitation of the capturing equipment.

```
loop:
  ldr     r2, [r4, #16]
  ldr     r3, [r4, #40]
  mla     r2, r2, r1, r5
  lsrs    r7, r2, #3
  lsrs    r0, r1, #3
  ldrb    r3, [r3, r7]
  ldr     r7, [r4, #52]
  and.w   r2, r2, #7
  asrs    r3, r2
  and.w   ip, r1, #7
  ldrb    r2, [r7, r0]
  and.w   r3, r3, #1
  lsl.w   r3, r3, ip
  adds    r1, #1
  orrs    r3, r2
  cmp r1, r6
  strb    r3, [r7, r0]
  bne.n   loop
```

a) First share

```
loop:
  ldr     r3, [r4, #64]
  ldr     r5, [r4, #52]
  ldrb    r3, [r3, r1]
  lsrs    r0, r2, #3
  asr.w   r3, r3, lr
  ldrb.w  ip, [r5, r0]
  and.w   r8, r2, #7
  and.w   r3, r3, #1
  lsl.w   r3, r3, r8
  adds    r2, #1
  orr.w   r3, r3, ip
  cmp r2, r6
  strb    r3, [r5, r0]
  add r1, r7
  bne.n   loop
```

b) Second share

Listing 4: Simplified excerpt of the assembly code of the `get_vole_aes_128_share` procedure. Sections corresponding to the C code in Listing 3 are highlighted in colour.

procedure, called from the `aes_key_schedule_backward_128_vbb_vk_round_share` procedure, which is also part of the VOLE protocol proof for the encryption. Recall that the backward pass derives the output values of the S-boxes (and their corresponding VOLE tags) of each round of the AES key schedule.

In the implementation of Degn et al. [14], the `get_vole_aes_128_share` procedure transposes individual entries of the VOLE tag matrix $\mathbf{V}$ dynamically, using the `ptr_get_bit` and `ptr_set_bit` macros. The code for this procedure is shown in Listings 3 and 4.

For each invocation of the `get_vole_aes_128_share` procedure, the 128 bits that form the share of the VOLE tag are read from the VOLE tag matrix in non-linear order and written to the output in transposed (i.e. linear) order. For each share bit of the VOLE tag, the procedure thus reads a byte from the matrix, extracts the corresponding bit using a bitwise AND operation, and updates a byte of the output using a bitwise OR operation. These steps are also visible in the highlighted lines of Listing 4. The information leakage is primarily caused by the incremental update of the output bytes. Each `strb` operation either increases the Hamming weight of the byte by one if the corresponding bit is set or does not change the Hamming weight of the byte if the corresponding bit is not set. This slight difference in Hamming weight is visible in the power consumption of the device, as shown in Fig. 4 and in the t-test results in Fig. 3. Notably, the difference in power consumption depending on the bit value is largest for the first bit that is written and decreases for all subsequent bits as the Hamming weight (and thus the power consumption) depends on all of the previously written bits. By observing the power consumption, it is possible to extract the shares of the VOLE tags belonging to the S-box outputs, and, from a combination of both shares, compute the corresponding witness bits and, finally, the cipher key.

In practice, we observe that the leakage of the first share is slightly weaker than that of the second share. The assembly code in Listing 4 shows that, while the realised behaviour is identical between both shares, the individual instructions are similar, but not identical. Given that it is still possible to recover the bits from the first share, we did not further investigate this difference.

We further observe that the duration of the loop for the processing of the first share decreases by one clock cycle from the 74th bit onwards. This is caused by the memory layout of the STM32F415RGT6, which maps the available 128KB SRAM into a 112KB SRAM1 and a 16KB SRAM2 chip [38, p. 71, Table 3]. The `vbb->vole_cache` buffer, from which the VOLE tags for the first share are read, is allocated across this boundary, causing reads from the 74th bit onwards to be performed from SRAM2 instead. Our measurements indicate that reads from SRAM2 take one clock cycle longer than reads from SRAM1, but we hypothesise that the loop duration decreases, because, when reading from SRAM2 (and only then), the `ldrb` instruction in line 7 of the first share of Listing 4 can be pipelined together with the succeeding `ldr` instruction in line 8, which always reads from SRAM2.

In the context of the attack, it is therefore necessary to consider bits 0–72 and bits 73–127 separately. We further found that bits 0–7 of the first byte, bits 120–

127 of the last byte, as well as bits 72–79 of the tenth byte (i.e. the byte where the loop duration changes) exhibit slightly different leakage characteristics. As the number of VOLE tag bits is quite large and recovering even just a single of these bits in an active column is sufficient to recover the corresponding witness bit, we elected to discard the bytes with different leakage characteristic and instead focus on two groups of bytes: bytes 1–8 with a longer loop duration for the first share and bytes 10–14 with a shorter loop duration. Note that the second share is not affected by the changes in the loop duration.

It is worth noting that the `get_vole_aes_128_share` procedure is also used by the `bf128_sum_poly_vbb_share` procedure, which is called during the computation of v^* in the last steps of the AES proof, thereby leaking information about the last 128 VOLE tags that are used to mask the vectors of multiplicative constraints $\mathbf{a}_0$ and $\mathbf{a}_1$. The description of the multiplicative constraints in the specification [9, Section 2.3.2] states that "[...] revealing a_0 and a_1 to the verifier leaks information about the circuit values used to compute them." However, in FAEST the multiplication checks are performed using a linear hash function, instead of using $\mathbf{a}_0$ and $\mathbf{a}_1$ directly. The signature therefore contains neither $\mathbf{a}_0$ nor $\mathbf{a}_1$, nor VOLE correlations directly involving either vector. At present, it is thus unclear how knowledge of v^* (or the VOLE tags used to compute it) can be used to recover $\mathbf{a}_0$ or $\mathbf{a}_1$. It is further also unclear whether knowledge of $\mathbf{a}_0$ or $\mathbf{a}_1$ is sufficient to reveal the secret key or to reduce the size of the search space to allow for enumeration. As the underlying leakage for this approach is the same as in the VOLE transpose attack, we expect countermeasures to be applicable to both and thus did not pursue this approach further.

7.1 Trace preprocessing and neural network training

The trace preprocessing for this attack is similar to that in the byte combine attack.

For the profiling phase, the dataset contains two groups with $5,120,000$ respective $3,200,000$ trace segments, obtained by capturing $t = 5000$ traces $\{T_1, \ldots, T_t\}$ for known VOLE tag matrices $\mathbf{V}_1, \ldots, \mathbf{V}_t$ selected at random. For each trace T_i, four segments T_i^{η}, $\eta \in [0..4)$ are extracted, each containing a single round of the key schedule. Note that the AES-128 key schedule actually performs a total of 10 rounds. For the attack, it is only important that four consecutive rounds are recovered (to give a total of $4 \cdot 32 = 128$ bits of information), so an attacker can theoretically make multiple attempts at recovering the secret key by recovering different sets of consecutive rounds from a single trace. In practice, we found it sufficient to recover only the first four rounds. The segments are then split into segments $T_i^{\eta,\Psi}$, $\Psi \in \{0, 1\}$, containing the processing of the first and second share within each round. Each segment is then divided into 32 further segments $T_i^{\eta,\Psi,\kappa}$, $\kappa \in [0..32)$, covering the processing of one witness bit. Finally, each segment is divided into 16 segments $T_i^{\eta,\Psi,\kappa,\nu}$, $\nu \in [0..16)$, covering the processing of one byte of the VOLE tag.

This process is illustrated in Fig. 3. As the leakage is clearly dependent on the value of previous bits, individual bits are not extracted into their own segments.

Instead, segments

$$\hat{T}_i^{\eta,\kappa,\nu} \coloneqq T_i^{\eta,0,\kappa,\nu} \parallel T_i^{\eta,1,\kappa,\nu}$$

are again concatenated across shares. To simplify the notation, we use $\hat{T}_i^{\phi,\pi} \coloneqq \hat{T}_i^{\eta,\kappa,\nu}$ with $\phi = 32\eta + \kappa$ and $\pi = \nu$, i.e. we flatten the round and witness dimensions into a single dimension. Recall that the rounds of the key schedule leak the S-box output for the computation of the round keys, which, for the first four rounds, is stored in bits 128–255 of the extended witness. The corresponding VOLE tags thus occupy rows 128–255 of the VOLE tag matrix, so in a further simplification of notation, we consider vectors $\hat{\mathbf{v}}_i$ of VOLE tags with $\hat{\mathbf{v}}_i[\iota] \coloneqq \mathsf{ToField}(\mathbf{V}_i|_{128+\iota})$ for $\iota \in [0..256)$ (i.e. we interpret rows 128–255 as a vector of field elements). Each segment $\hat{T}_i^{\phi,\pi}$ is labelled with the 8 bits of the VOLE tag being processed, $\hat{\mathbf{v}}_i[\phi][\![8\pi..8\pi+7]\!]$, which is the XOR of the 8 bits being processed in each share.

As explained previously, we discard bytes 0, 9, and 15, as they exhibit different leakage characteristics. We thus form groups

$$G_0 \coloneqq \{\hat{T}_i^{\phi,\pi} \mid \pi \in [1..8]\},$$
$$G_1 \coloneqq \{\hat{T}_i^{\phi,\pi} \mid \pi \in [10..14]\}.$$

In the following, we will also use the set

$$R \coloneqq \{8x + y \mid x \in [0..16), y \in [0..8), x \notin \{0, 9, 15\}\}$$

of VOLE tag matrix columns that are recoverable using this method.

In total, we train 16 MLP neural networks $\mathcal{N}_{g,n} : \mathbb{R}^{512} \to \{0,1\}$, $g \in \{0,1\}, n \in [0..8)$, with the architecture shown in Tab. 3, one for each bit of a byte in both of the groups. The networks map each segment $\hat{T}_i^{\phi,\pi} \in G_g$ into a score vector $S_{i,\phi,8\pi+n} = \mathcal{N}_{g,n}(\hat{T}_i^{\phi,\pi})$, such that $S_{i,\phi,8\pi+n}[\alpha]$ represents the probability that the VOLE tag bit $\hat{\mathbf{v}}_i[\phi][\![8\pi + n]\!]$ takes value $\alpha \in \{0,1\}$:

$$S_{i,\phi,8\pi+n}[\alpha] = \Pr[\hat{\mathbf{v}}_i[\phi][\![8\pi + n]\!] = \alpha]. \tag{4}$$

The models are trained using the same strategy and parameters as in the byte combine attack.

7.2 Partial enumeration and secret key recovery method

Unlike in the byte combine attack, knowledge of the VOLE tags does not reveal the secret key directly. In this section, we describe an approach for secret key recovery that combines information about several VOLE tags and uses the maximum predicted class probability to select certain bits for enumeration, thereby increasing the success probability of the attack.

Alg. 5 shows the steps of the approach. As input, the algorithm receives the signature σ and a list of predictions $(S_{i,j})_{i\in[0..\lambda),j\in R}$ (note that Eq. 4 implicitly flattens the VOLE tag byte and bit dimensions into a single dimension indexed

Algorithm 5 RecoverSecretKey($\sigma, (S_{i,j})_{i\in[0..\lambda),j\in R}$)

```
1: $((\mathbf{c}_i)_{i\in[1..\tau)}, (\mathsf{pdecom}_i)_{i\in[1..\tau)}, \mathsf{chall}_3, \mathsf{iv}) = \sigma$
2: $(\mathbf{Q}, \Delta) \leftarrow \mathsf{VOLEReconstructAndCorrect}(\mathsf{chall}_3, \mathsf{iv}, (\mathsf{pdecom}_i)_{i\in[1..\tau)}, (\mathbf{c}_i)_{i\in[1..\tau)}, \hat{l})$
3: $R_\Delta \leftarrow R \cap \{i \in [0..\lambda) \mid \Delta[\![i]\!] = 1\}$    ▷ Valid active columns
4: for $i \in [0..\lambda)$ do
5:     $\mathbf{q}[i] \leftarrow \mathsf{ToField}(\mathbf{Q}|_{\lambda+i})$
6: for $i \in [0..\lambda)$ do
7:     for $j \in R_\Delta$ do
8:         $\mathbf{w}[j] \leftarrow \mathbf{q}[i][\![j]\!] \oplus \operatorname{argmax}(S_{i,j})$
9:     $m_0 \leftarrow \sum_{j\in R_\Delta} (1 - \mathbf{w}[j]) \cdot \max(S_{i,j}) + \mathbf{w}[j] \cdot \min(S_{i,j})$
10:    $m_1 \leftarrow \sum_{j\in R_\Delta} \mathbf{w}[j] \cdot \max(S_{i,j}) + (1 - \mathbf{w}[j]) \cdot \min(S_{i,j})$
11:    $\mathbf{b}[i] \leftarrow \operatorname{argmax}(\{m_0, m_1\})$    ▷ Witness bit prediction
12:    $\mathbf{p}[i] \leftarrow \max(\{m_0, m_1\})/(m_0 + m_1)$    ▷ Prediction probability
13: $e \leftarrow \operatorname{argsort}(\mathbf{p})_{[0..32)} \cap \{i \in [0..\lambda) \mid \mathbf{p}[i] \le 0.9\}$
14: for $k \in \{0,1\}^{|e|}$ do
15:     for $i \in [0..|e|)$ do
16:         $\mathbf{b}[e[i]] \leftarrow k[\![i]\!]$    ▷ Enumerate bits
17:     $k'_4 \leftarrow \mathbf{b}_{[0..32)}, k'_8 \leftarrow \mathbf{b}_{[32..64)}, k'_{12} \leftarrow \mathbf{b}_{[64..96)}, k'_{16} \leftarrow \mathbf{b}_{[96..128)}$
18:     $k'_7 \leftarrow \mathsf{SubWord}^{-1}(\mathsf{RotWord}^{-1}(k'_8 \oplus k'_4 \oplus \mathtt{Rcon}[1]))$
19:     $k'_{11} \leftarrow \mathsf{SubWord}^{-1}(\mathsf{RotWord}^{-1}(k'_{12} \oplus k'_8 \oplus \mathtt{Rcon}[2])))$
20:     $k'_{15} \leftarrow \mathsf{SubWord}^{-1}(\mathsf{RotWord}^{-1}(k'_{16} \oplus k'_{12} \oplus \mathtt{Rcon}[3]))$
21:     $k'_{10} \leftarrow k'_{11} \oplus k'_7$
22:     $k'_{14} \leftarrow k'_{15} \oplus k'_{11}$
23:     $k'_{13} \leftarrow k'_{14} \oplus k'_{10}$    ▷ Fourth round key fully recovered
24:     for $i \in [2..0]$ do    ▷ Run key schedule in reverse to get original cipher key
25:         $k'_{4i+0} \leftarrow k'_{4(i+1)+0} \oplus \mathsf{SubWord}(\mathsf{RotWord}(k'_{4(i+1)+3} \oplus k'_{4(i+1)+2} \oplus \mathtt{Rcon}[i]))$
26:         $k'_{4i+1} \leftarrow k'_{4(i+1)+1} \oplus k'_{4(i+1)+0}$
27:         $k'_{4i+2} \leftarrow k'_{4(i+1)+2} \oplus k'_{4(i+1)+1}$
28:         $k'_{4i+3} \leftarrow k'_{4(i+1)+3} \oplus k'_{4(i+1)+2}$
29:     yield $\mathsf{sk} := k'_0 \parallel k'_1 \parallel k'_2 \parallel k'_3$
```

by j and that i now indexes the witness bit, not the trace). From there, the VOLE key matrix $\mathbf{Q}$ is reconstructed, using the VOLEReconstructAndCorrect procedure, which is equivalent to steps 3 to 12 of Alg. 3.

Then, using the models' predictions $S_{i,j}$, for each witness bit, the predicted class probabilities for witness bit value 0 and 1 are added up across all VOLE tag bits in the active columns. This requires adjusting the predicted class probabilities such that they correspond to the resulting witness bit, not the VOLE tag. For example, a VOLE tag prediction of 1 for a VOLE key of 1 corresponds to a witness bit prediction of 0, not 1, due to Eq. 2. The final witness bit prediction is then given by whether the total predicted class probability for the witness bit is larger for bit value 0 or 1. We found that this approach better accounts for VOLE tag predictions with lower maximum predicted class probability, which would otherwise be overrepresented in a majority vote on bit values alone. A normalised probability of the witness bit prediction (i.e. a probability in $(0.5, 1.0]$ indicating the overall confidence in the witness bit prediction) is also

```
void prg(uint8_t* key, uint8_t* iv, uint8_t* out, size_t outlen) {
  aes128ctx_publicinputs ctx128;
  aes128_ecb_keyexp_publicinputs(&ctx128, key);
  for (; outlen >= 16; outlen -= 16, out += 16) {
    aes128_ecb_publicinputs(out, iv, 1, &ctx128);
    aes_increment_iv(iv);
  }
  aes128_ctx_release_publicinputs(&ctx128);
}
```

Listing 5: The C code of the prg procedure. The section corresponding to the assembly code in Listing 6 is highlighted in colour.

```
loop:
  mov r0, r4
  mov r3, r5
  movs    r2, #1
  mov r1, sp
  bl  aes128_ecb_publicinputs
  adds    r4, #16
  mov r0, sp
  bl  aes_increment_iv
  cmp r4, r7
  bne.n   loop
  ...
```

Listing 6: Simplified excerpt of the assembly code of the prg procedure. The section corresponding to the C code in Listing 5 is highlighted in colour.

computed. For enumeration, up to 32 bits with lowest normalised probability (up to a normalised probability of 0.9) are selected and the corresponding witness bits are enumerated. The value 0.9 was chosen empirically to reduce the average enumeration cost.

The resulting witness prediction then gives the words that form the S-box output for round keys 1–4 during the key schedule. From this, all 4 words of round key 4 are recomputed, which allows applying the key schedule in reverse to derive the initial round key, which is the cipher key.

8 Counter increment skipping attack

The third attack, which we call the *counter increment skipping* attack, skips the incrementation step for the counter in the pseudorandom generator prg used to derive the per-tree keys from the root key during the opening of the vector commitment.

Recall that, instead of computing a single large VOLEitH instance, FAEST computes $\tau = 16$ different VOLEitH instances and then combines these to construct the final VOLE correlation. The individual VOLEitH instances each use vector commitments computed using a GGM tree. To derive the keys that are used as the roots of the GGM trees, FAEST expands a so-called *root key* into several individual keys using a pseudorandom generator, which is AES in CTR mode. The pseudorandom generation of the expanded keys is therefore performed by encrypting an incrementing counter under the root key and using each block of output as a root for a GGM tree.

The idea behind the attack is to skip the incrementation of the counter with a single fault during the expansion of the keys. This causes the pseudorandom generator to generate the same output for two blocks, meaning that two neighbouring GGM trees are derived from the same root. As the nodes of the GGM tree are derived pseudorandomly from the root, the leaves of these two trees, i.e. the seeds $\mathtt{sd}_i$, will be the same. For both trees, the value of the global key $\varDelta$ will now choose all-but-one of these seeds to reveal publicly in the signature. If the value that is missing in the first tree is revealed in the second tree, it is possible to recompute the entire GGM tree and thereby break the hiding property of the VOLE commitment. Knowledge of the full tree is sufficient to derive the values u_i or v_i, and, from there, recover the witness bits from the signature. In cases where the same seed $\mathtt{sd}_i$ is missing from both trees (i.e. $\varDelta[\![0..7]\!] = \varDelta[\![8..15]\!]$), it is not possible to recover a tree and the attack fails. As the value of $\varDelta$ is uniformly and randomly distributed, the case where the same seed is selected in both trees occurs with the probability of approximately 2^{-8} and is therefore not a problem in practice.

In the implementation of Degn et al. [14], the computation of the GGM trees is performed twice, once to compute all VOLE tags prior to the VOLE protocol proof using the `partial_vole_commit_column` procedure and once to compute each GGM tree individually during the opening of the vector commitment at the end of the signature generation using the `vector_open_ondemand` procedure. Fixing the roots of the GGM trees during the computation of the VOLE tags therefore does not lead to key recovery, because the partial decommitments in the signature are recomputed from the original root key during the opening of the vector commitment and are not faulty. We instead target the pseudorandom generator procedure `prg` called in the `vector_open_ondemand` procedure. By skipping the branching to the `aes_increment_iv` procedure in line 6 of Listing 5 (i.e. by skipping the `bl` instruction in line 9 of Listing 6), the `prg` procedure will generate two identical blocks of output, leading to two identical GGM trees.

8.1 Secret key recovery method

Recovering the seed from a signature with two identical GGM trees is straightforward. Alg. 6 shows the steps of the approach. If the same seed is missing in both trees, the recovery fails. Otherwise, the seeds of the individual trees are reconstructed, the missing seed in the first tree is taken from the second tree, and the vector $\mathbf{u}$ is recomputed from the now complete set of seeds. Using $\mathbf{u}$ and

Algorithm 6 RecoverSecretKey(σ)

```
 1: $(\mathbf{d}, (\mathsf{pdecom}_i)_{i\in[1..\tau)}, \mathsf{chall}_3, \mathsf{iv}) \leftarrow \sigma$
 2: $(\delta_{0,0}, \ldots, \delta_{0,k_1-1}) \leftarrow \mathsf{ChalDec}(\mathsf{chall}_3, 0)$
 3: $(\delta_{1,0}, \ldots, \delta_{1,k_1-1}) \leftarrow \mathsf{ChalDec}(\mathsf{chall}_3, 1)$
 4: $\Delta_0 \leftarrow \mathsf{NumRec}((\delta_{0,0}, \ldots, \delta_{0,k_1-1}))$
 5: $\Delta_1 \leftarrow \mathsf{NumRec}((\delta_{1,0}, \ldots, \delta_{1,k_1-1}))$
 6: if $\Delta_0 = \Delta_1$ then return $\perp$          ▷ Cannot recover if same seed is missing
 7: $(s_{0,j})_{j\in[0..2^{k_1}),j\neq\Delta_0} \leftarrow \mathsf{VC.Reconstruct}(\mathsf{pdecom}_0, (\delta_{0,0}, \ldots, \delta_{0,k_1-1}), \mathsf{iv})$
 8: $(s_{1,j})_{j\in[0..2^{k_1}),j\neq\Delta_1} \leftarrow \mathsf{VC.Reconstruct}(\mathsf{pdecom}_1, (\delta_{1,0}, \ldots, \delta_{1,k_1-1}), \mathsf{iv})$
 9: for $j \in [1..2^{k_1})$ do $\mathsf{sd}_j \coloneqq s_{0,j\oplus\Delta_0}$
10: $\mathsf{sd}_0 \coloneqq s_{1,\Delta_0}$          ▷ Use missing seed from second tree
11: $\mathbf{u} \leftarrow \mathsf{ConvertToVOLE}(\mathsf{sd}_0, \ldots, \mathsf{sd}_{2^{k_1}-1}, \mathsf{iv}, \hat{l})$
12: $\mathbf{w} \leftarrow \mathbf{d}_{[0..\lambda)} \oplus \mathbf{u}_{[0..\lambda)}$
13: return $\mathsf{sk} \coloneqq \mathbf{w}$
```

the vector $\mathbf{d}$ from the signature, the witness $\mathbf{w}$ is recovered and the secret key (i.e. the first 128 bits) is extracted.

Note that it is also possible to compute a VOLE tag matrix from the full set of seeds and to recover $\mathbf{w}$ from the VOLE correlations. This approach works in the same cases, because $\Delta_0 \neq \Delta_1$ implies that at least one column in the VOLE tag matrices derived from the first or second tree is active. As it is also possible to take the missing seed in the second tree from the first tree (as opposed to the other way around), both VOLE tag matrices can be computed. Thus, a VOLE tag for an active column is known and can be used to recover $\mathbf{w}$. However, as there is no benefit to this slightly more complicated approach, we only present the recovery from $\mathbf{u}$.

9 VOLE conversion abort attack

The fourth attack, which we call the *VOLE conversion abort* attack, targets the conversion of the first GGM tree seeds to the vector $\mathbf{u}_0$ of random bits and the VOLE tag matrix $\mathbf{V}_0$ during the VOLE commitment.

The idea behind the attack is to abort the conversion of the seeds to the VOLE so that the output is dependent only on the first seed. In cases where this seed is revealed in a partial decommitment in the signature (i.e. where $\Delta[\![0..7]\!] \neq \mathbf{0}$), $\mathbf{u}_0$ and $\mathbf{V}_0$ are thereby fixed to known values that can be recomputed to reveal the secret key. In cases where the first seed is missing (i.e. where $\Delta[\![0..7]\!] = \mathbf{0}$), the attack fails, which occurs with approximate probability 2^{-8} and is therefore not a problem in practice.

In the implementation by Degn et al. [14], the VOLE conversion and commitment are merged into the `partial_vole_commit_column` procedure. By aborting the loop in line 14 of Listing 7 (i.e. by skipping the `bhi` instruction in line 5 of Listing 8) during the VOLE commitment to the first GGM tree, the vector $\mathbf{u}_0$ of random bits and the VOLE tag matrix $\mathbf{V}_0$ depend only on the first seed and can thus be recomputed. Note that it would also be possible to abort the loop

```
void partial_vole_commit_column(uint8_t* rootKey, uint8_t* iv,
                                vole_t* out) {
  uint8_t* expanded_keys[16*16];
  prg(rootKey, iv, expanded_keys, 128, 16*16);
  vec_com_t vec_com;
  int factor_32 = 234 / 4;
  for (unsigned int idx = 0; idx < 16; idx++) {
    vector_commitment(expanded_keys + idx*16, &vec_com);
    vole_t* cur_vole = out + idx*(sizeof vole_t);
    uint8_t* u = cur_vole->u;
    uint8_t* c = cur_vole->c;
    uint8_t* v = cur_vole->v;

    for (unsigned int i = 0; i < 256; i++) {
      extract_sd_com(&vec_com, iv, i, sd, com);
      prg(sd, iv, r, 128, 234);

      // Compute u
      xor_u32_array(u, r, u, factor_32);
      xor_u8_array(u + factor_32*4, r + factor_32*4,
                   u + factor_32*4, 234 - factor_32*4);

      // Compute VOLE tags
      for (unsigned int j = i*8; j < (i+1)*8; j++) {
        unsigned int t_v = j - i*8;
        if ((i >> t_v) & 1) {
          xor_u32_array(v + j, r, v + j, factor_32);
          xor_u8_array(v + j + factor_32*4, r + factor_32*4,
                       v + j + factor_32*4, 234 - factor_32*4);
        }
      }
    }

    if (idx != 0) {
      // Compute correction values
      xor_u8_array(out->u, u, c, 234);
    }
  }
}
```

Listing 7: The C code of the `partial_vole_commit_column` procedure. The section corresponding to the assembly code in Listing 8 is highlighted in colour.

```
...
ldr     r3, [r7, #112]
adds    r4, #1
cmp     r3, r4
bhi.w   loop
mov     fp, r6
ldr     r3, [r7, #124]
...
```

Listing 8: Simplified excerpt of the assembly code of the `partial_vole_commit_column` procedure. The section corresponding to the C code in Listing 7 is highlighted in colour.

Algorithm 7 RecoverSecretKey(σ)

```
1: $(\mathbf{d}, (\mathsf{pdecom}_i)_{i \in [1..\tau)}, \mathsf{chall}_3, \mathsf{iv}) \leftarrow \sigma$
2: $(\delta_{0,0}, \ldots, \delta_{0,k_1-1}) \leftarrow \mathsf{ChalDec}(\mathsf{chall}_3, 0)$
3: $\Delta_0 \leftarrow \mathsf{NumRec}(k_1, (\delta_{0,0}, \ldots, \delta_{0,k_1-1}))$
4: if $\Delta_0 = 0$ then return $\perp$          ▷ Cannot recover if first seed is missing
5: $(s_j)_{j \in [0..2^{k_1}), j \neq \Delta_0} \leftarrow \mathsf{VC.Reconstruct}(\mathsf{pdecom}_0, (\delta_{0,0}, \ldots, \delta_{0,k_1-1}), \mathsf{iv})$
6: $\mathbf{u} \leftarrow \mathsf{PRG}(s_0, \mathsf{iv}, \hat{l})$
7: $\mathbf{w} \leftarrow \mathbf{d}_{[0..\lambda)} \oplus \mathbf{u}_{[0..\lambda)}$
8: return $\mathsf{sk} := \mathbf{w}$
```

during the VOLE commitment for a different GGM tree k and use either its VOLE tag matrix $\mathbf{V}_k$ or recompute the original vector $\mathbf{u}_0$ from the correction values $\mathbf{c}_k = \mathbf{u}_0 \oplus \mathbf{u}_k$ and knowledge of $\mathbf{u}_k$.

We further found that it is occasionally possible to skip the loop entirely, thereby causing the value of $\mathbf{u}_0$ to be zeroised (the implementation explicitly zero-initialises this value). However, this fault only succeeds with very low probability and causes the device to crash if it fails.

9.1 Secret key recovery method

Recovering the secret key from a signature with a partial VOLE is similar to the recovery in the counter increment skipping attack. Alg. 7 shows the steps of the approach. If the first seed of the first GGM tree is missing, the recovery fails. Otherwise, the seeds of the first tree are reconstructed. The vector $\mathbf{u}$ can then be derived from the first seed using the PRG pseudorandom generator, and, like in the previous attack, can be used to recover the witness $\mathbf{w}$ and the secret key. Note that the implementation by Degn et al. [14] does not reorder the seeds during the VOLE commitment. The output of the ConvertToVOLE procedure is independent of the order of the individual seeds if all seeds are provided, thus the lack of reordering has no effect on the generated signature. However, in the context of the attack, it is important that all seeds used prior to the VOLE

Table 4. Empirical success probabilities and enumeration complexities for 1000 secret keys selected at random.

Byte combine attack	VOLE transpose attack	Counter increment skipping attack	VOLE conversion abort attack
0.997 (2^{16})	0.989 ($\leq 2^{32}$)	0.952 (0)	0.871 (0)

conversion being aborted are known. Thus, if the implementation had reordered the seeds, aborting the loop after the first iteration would not have resulted in secret key recovery, because the needed seed (i.e. the seed at index Δ_0) would have been missing from the signature.

As in the counter increment skipping attack, it is also possible to compute the VOLE tag matrix and recover $\mathbf{w}$ from the VOLE correlations. Again, this approach works in the same cases, because $\Delta_0 \neq 0$ implies that at least one column in the VOLE tag matrix derived from the first tree is active. Thus, recomputing the VOLE tag matrix from the first seed gives a VOLE tag in an active column that can be used to recover $\mathbf{w}$. As before, there is no benefit to this slightly more complicated approach, which is why we only present the recovery from $\mathbf{u}$.

10 Experimental results

This section describes the results of the presented side-channel and fault injection attacks. The stated probabilities are empirical probabilities (mean over 1000 secret keys selected at random) for recovering the secret key from a single execution of the signing procedure. The results are summarised in Tab. 4.

For the byte combine attack, we train a neural network as described in Section 6.1. Using this network and the technique described in Section 6.2, we are able to recover the secret key in 99.7% of 1000 attempts, with an enumeration of 2^{16} for the bits that cannot be recovered using the attack.

For the VOLE transpose attack, we train 16 neural networks as described in Section 7.1. Using these networks and the technique described in Section 7.2, we are able to recover the secret key in 98.9% of 1000 attempts, with an average enumeration of $2^{5.21}$ and a maximum enumeration of 2^{32}. Without enumeration, the attack still succeeds with the probability of 91.7%.

For the counter increment skipping attack, we are able to recover the secret key in 95.2% of 1000 attempts using the technique described in Section 8.1. Among the unsuccessful attempts, we encounter the scenario where the same seed is missing from both GGM trees for three signatures.

For the VOLE conversion abort attack, we are able to recover the secret key in 87.1% of 1000 attempts using the technique described in Section 9.1. Among the unsuccessful attempts, we encounter the scenario where the first seed is missing from the first GGM tree for two signatures.

11 Countermeasures

This section discusses potential countermeasures against the presented side-channel and fault injection attacks.

11.1 Countermeasures common to both side-channel attacks

One approach to protect against the byte combine attack and the VOLE transpose attack is to remove the underlying bitwise leakage by parallelising operations. In both attacks, the bit-by-bit processing allows secrets to be extracted, even in the presence of masking. As shown in [16], higher-order masking countermeasures are not effective unless the bitwise leakage is eliminated. This may be achieved by bitslicing the operations which need to be protected, to ensure that multiple bits are processed in parallel.

A different approach is to employ shuffling [11] to randomly reorder operations at runtime. While this approach still allows an attacker to identify the value of bits being processed, it makes it harder for the attacker to map these bits into the secrets. It is important to ensure that the shuffling procedure is itself resistant to physical attacks, such as [24], and that the number of operations being shuffled is sufficiently large to not be enumerable.

11.2 Countermeasures against the byte combine attack

In the paper [3] first identifying this type of leakage in NewHope, several countermeasures were proposed. One approach involves minimising the leakage by reducing the number of bits in the generated mask. This approach is also applicable to the `bf128_byte_combine_bits` procedure, as the left operands of the bit multiplications are constant generator values α_i with known bit-patterns. Though, as pointed out in [3], even a smaller mask can generate sufficient leakage to enable an attack, so this countermeasure alone may not be sufficient in practice.

The authors of [3] also suggest the use of shuffling, though the approach they present for NewHope of reordering the multiplications in the loop cannot be translated to FAEST directly, because the `bf128_byte_combine_bits` procedure processes individual bytes and the seven bit operations that could be reordered in this way are easily enumerable. Instead, the shuffling should be performed at a higher level, across both bytes and bits.

Note that the byte combine attack would not be prevented by using the Even-Mansour (EM) variants of FAEST that instead use AES with a public key and secret input (see [9]), as the implementation by Degn et al. [14] processes the input using the same `bf128_byte_combine_bits` function as the one used for the key.

11.3 Countermeasures against the VOLE transpose attack

As mentioned previously, shuffling the order in which bits are read and written during the transpose prevents the VOLE transpose attack. When implementing such type of shuffling for this procedure, care must be taken to shuffle both the order of witness bits and the order of VOLE tag matrix columns that are being processed, as the Hamming weight of a VOLE tag is sufficient for distinguishing a witness bit, so it is not sufficient to simply reorder the VOLE tag matrix columns.

We also experimented with using the bit-banding feature available in some Cortex-M4 processors [38, Section 2.3.3]. This feature maps part of the original address space into an alias region, such that individual bits in the original address space are word-addressable in the alias region. This allows eliminating the bitwise arithmetic operations used to extract and insert individual bits during the transpose. However, we found that storing individual bits to memory still produced sufficiently strong leakage to distinguish individual bit values.

Note that this attack is also not prevented by using the EM variants of FAEST, because, in the implementation by Degn et al. [14], the combined forward and backward pass for the proof of the encryption also uses the `get_vole_aes_128_share` procedure, which leaks VOLE tags for witness bits of the encryption.

11.4 Countermeasures common to both fault attacks

To protect against both the counter increment skipping attack and the VOLE conversion abort attack, it is possible to check the validity of the generated signature. In both cases, the attacks cause the generated signature to be invalid, which can be detected and used to abort the signing procedure without revealing the signature, thereby preventing the attacks. Note that, for the counter increment skipping attack, this is only possible due to how the implementation by Degn et al. [14] handles the opening of the vector commitments. As described in Section 8, in the implementation, the opening procedure recomputes the expanded keys and GGM trees from a root key. A fault during the opening therefore causes the partial decommitments to not match the VOLE correlations used to compute the rest of the signature. In an implementation that only computes the GGM trees once, the VOLE correlations would match the partial decommitments and the attack could thus not be detected in this way. The VOLE conversion abort attack always causes the generated signature to become invalid, even without repeating computations.

More broadly, both attacks could also be prevented by eliminating the branches that are targeted by the fault injection. For the counter increment skipping attack, this would involve inlining the `aes_increment_iv` procedure into the `prg` procedure (or combining it with the encryption). As the `aes_increment_iv` procedure is relatively simple, this approach would be an effective way of preventing the attack. For the VOLE conversion abort attack, this would require unrolling the loop that is targeted by the fault injection. Due to the number of iterations of this loop (256 for FAEST-128f) being quite large, this approach is likely to be practically infeasible.

11.5 Countermeasures against the counter increment skipping attack

One approach to prevent the counter increment skipping attack is to check for identical GGM trees during the signature generation. This can be accomplished, for example, by explicitly checking the expanded keys r_i used to perform the vector commitments, or by comparing the computed commitments com_i during the VOLE commitment. It is also possible to use the vectors of correction values $\mathbf{c}_i$ to detect identical trees. If the first and second tree are identical, the first set of correction values is all-zero. If two other neighbouring trees are identical, their correction values are identical. If identical trees are detected, the signing should be aborted without revealing the signature, thereby preventing the attack.

A different approach is to eliminate the use of multiple GGM trees entirely by replacing them with a batch all-but-one vector commitment, as proposed in [7]. Their approach instead uses a single larger tree to derive the seeds used to compute the VOLE correlations, thereby eliminating the need to expand a root key into several keys and thus preventing the attack.

11.6 Countermeasures against the VOLE conversion abort attack

As mentioned in Section 9.1, reordering the seeds during the VOLE conversion in the VOLE commitment to the ordering dictated by the specification [9] effectively prevents the attack. The reason for this is that the loop abort causes the faulty VOLE to depend only on the first seed. If this seed is missing from the signature (i.e. if the missing seed is processed first), it is not possible to recompute the faulty VOLE and recover the secret key. Note that shuffling the order in which the VOLEs are processed during the conversion is not enough to prevent the attack, as all-but-one of the possible seeds are included in the signature and the value of the faulty VOLE could thus easily be enumerated.

It is also possible to use a similar approach to the detection of identical GGM trees for the counter increment skipping attack involving the correction values. In the implementation by Degn et al. [14], the buffer containing the correction values is explicitly zero-initialised. Thus, when aborting the loop, the correction values computed in later iterations will instead by zero, which can be detected directly or through the comparison of neighbouring correction values proposed in the previous section. If a loop abort is detected in this way, the signing should be aborted without revealing the signature, thereby preventing the attack.

12 Conclusion

This paper presents the first evaluation of the resistance of VOLEitH-based signature schemes to side-channel and fault injection attacks. We demonstrated how knowledge of individual components of the underlying VOLE correlations can be exploited to recover the secret key. We further presented practical single-trace deep learning-assisted power-based side channel attacks and single-execution first-order voltage fault injection attacks on a masked implementation of FAEST, successfully recovering the full secret key with a probability greater than 87%. We also proposed countermeasures against these attacks.

Our results highlight the importance of protecting operations involving witness bits and VOLE tags from side-channel attacks, as well as the need to ensure that operations involving the computation of VOLE and vector commitments are resilient to fault injection attacks.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and by the Swedish Civil Contingencies Agency (Grant No. 2020-11632).

References

1. Aguilar Melchor, C., Feneuil, T., Gama, N., Gueron, S., Howe, J., Joseph, D., Joux, A., Persichetti, E., H. Randrianarisoa, T., Rivain, M., Yue, D.: The syndrome decoding in the head (SD-in-the-Head) signature scheme (May 2023), https://sdith.org/docs/sdith-package-v1.zip
2. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 430–454. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_17
3. Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.: Defeating NewHope with a single trace. In: Ding, J., Tillich, J. (eds.) Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12100, pp. 189–205. Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_11
4. Aranha, D.F., Berndt, S., Eisenbarth, T., Seker, O., Takahashi, A., Wilke, L., Zaverucha, G.: Side-channel protections for Picnic signatures. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 239–282 (2021). https://doi.org/10.46586/TCHES.V2021.I4.239-282
5. Aulbach, T., Campos, F., Krämer, J., Samardjiska, S., Stöttinger, M.: Separating oil and vinegar with a single trace side-channel assisted Kipnis-Shamir attack on UOV. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(3), 221–245 (2023). https://doi.org/10.46586/TCHES.V2023.I3.221-245
6. Aulbach, T., Marzougui, S., Seifert, J., Ulitzsch, V.Q.: MAYo or MAY-not: Exploring implementation security of the post-quantum signature scheme MAYO against physical attacks. In: Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2024, Halifax, NS, Canada, September 4, 2024. pp. 28–33. IEEE (2024). https://doi.org/10.1109/FDTC64268.2024.00012
7. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In: Chung, K., Sasaki, Y. (eds.) Advances

in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15484, pp. 463–493. Springer (2024). https://doi.org/10.1007/978-981-96-0875-1_15

8. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Orsini, E., Roy, L., Scholl, P.: Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-Head. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V. Lecture Notes in Computer Science, vol. 14085, pp. 581–615. Springer (2023). https://doi.org/10.1007/978-3-031-38554-4_19
9. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Majenz, C., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: FAEST: Algorithm Specifications (July 2023), https://faest.info/faest-spec-v1.1.pdf
10. Beullens, W., Campos, F., Celi, S., Hess, B., Kannwischer, M.J.: MAYO (June 2023), https://pqmayo.org/assets/specs/mayo.pdf
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999). https://doi.org/10.1007/3-540-48405-1_26
12. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1825–1842. ACM (2017). https://doi.org/10.1145/3133956.3133997
13. Chen, Z., Karabulut, E., Aysu, A., Ma, Y., Jing, J.: An efficient non-profiled side-channel attack on the CRYSTALS-Dilithium post-quantum signature. In: 39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021. pp. 583–590. IEEE (2021). https://doi.org/10.1109/ICCD53106.2021.00094
14. Degn, J.T., Eilath, J., Nielsen, K.: pqm4-faest, https://github.com/johandegn/pqm4-faest
15. Dobraunig, C., Kales, D., Rechberger, C., Schofnegger, M., Zaverucha, G.: Shorter signatures based on tailor-made minimalist symmetric-key crypto. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 843–857. ACM (2022). https://doi.org/10.1145/3548606.3559353
16. Dubrova, E., Ngo, K., Gärtner, J., Wang, R.: Breaking a fifth-order masked implementation of CRYSTALS-Kyber by copy-paste. In: Fukumitsu, M., Hasegawa, S. (eds.) Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop, APKC 2023, Melbourne, VIC, Australia, July 10-14, 2023. pp. 10–20. ACM (2023). https://doi.org/10.1145/3591866.3593072
17. Feneuil, T., Rivain, M.: Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. IACR Cryptol. ePrint Arch. p. 1573 (2023), https://eprint.iacr.org/2023/1573

18. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings. Lecture Notes in Computer Science, vol. 263, pp. 186–194. Springer (1986). https://doi.org/10.1007/3-540-47721-7_12
19. Gellersen, T., Seker, O., Eisenbarth, T.: Differential power analysis of the Picnic signature scheme. In: Cheon, J.H., Tillich, J. (eds.) Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12841, pp. 177–194. Springer (2021). https://doi.org/10.1007/978-3-030-81293-5_10
20. Godard, J., Aragon, N., Gaborit, P., Loiseau, A., Maillard, J.: Single trace side-channel attack on the MPC-in-the-Head framework. IACR Cryptol. ePrint Arch. p. 1882 (2024), https://eprint.iacr.org/2024/1882
21. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: 25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984. pp. 464–479. IEEE Computer Society (1984). https://doi.org/10.1109/SFCS.1984.715949
22. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. SIAM J. Comput. **39**(3), 1121–1152 (2009). https://doi.org/10.1137/080725398
23. Jendral, S., Dubrova, E.: Single-trace side-channel attacks on MAYO exploiting leaky modular multiplication. IACR Cryptol. ePrint Arch. p. 1850 (2024), https://eprint.iacr.org/2024/1850
24. Jendral, S., Ngo, K., Wang, R., Dubrova, E.: Breaking SCA-Protected CRYSTALS-Kyber with a single trace. In: IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024. pp. 70–73. IEEE (2024). https://doi.org/10.1109/HOST55342.2024.10545390
25. Karabulut, E., Aysu, A.: FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021. pp. 691–696. IEEE (2021). https://doi.org/10.1109/DAC18074.2021.9586131
26. Krahmer, E., Pessl, P., Land, G., Güneysu, T.: Correction fault attacks on randomized CRYSTALS-Dilithium. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(3), 174–199 (2024). https://doi.org/10.46586/TCHES.V2024.I3.174-199
27. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
28. National Institute of Standards and Technology: NIST announces additional digital signature candidates for the PQC standardization process (June 2023), https://csrc.nist.gov/news/2023/additional-pqc-digital-signature-candidates
29. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. Tech. Rep. NIST FIPS 204, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.204
30. National Institute of Standards and Technology: Module-Lattice-Based Key Encapsulation Mechanism Standard. Tech. Rep. NIST FIPS 203, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.203
31. National Institute of Standards and Technology: NIST announces 14 candidates to advance to the second round of the additional digital signatures for the post-quantum cryptography standardization process (October 2024), https://csrc.nist.gov/news/2024/pqc-digital-signature-second-round-announcement

32. National Institute of Standards and Technology: Stateless Hash-Based Digital Signature Standard. Tech. Rep. NIST FIPS 205, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.205
33. Ngo, K., Dubrova, E., Johansson, T.: Breaking masked and shuffled CCA secure Saber KEM by power analysis. In: Chang, C., Rührmair, U., Katzenbeisser, S., Mukhopadhyay, D. (eds.) ASHES@CCS 2021: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, Virtual Event, Republic of Korea, 19 November 2021. pp. 51–61. ACM (2021). https://doi.org/10.1145/3474376.3487277
34. Patranabis, S., Mukhopadhyay, D.: Fault Tolerant Architectures for Cryptography and Hardware Security. Computer Architecture and Design Methodologies, Springer Singapore Pte. Limited, Singapore, 1st edn. (2018). https://doi.org/0.1007/978-981-10-1387-4
35. Roy, L.: SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the Minicrypt model. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13507, pp. 657–687. Springer (2022). https://doi.org/10.1007/978-3-031-15802-5_23
36. Seker, O., Berndt, S., Wilke, L., Eisenbarth, T.: SNI-in-the-head: Protecting MPC-in-the-head protocols against side-channel analysis. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1033–1049. ACM (2020). https://doi.org/10.1145/3372297.3417889
37. Sim, B., Kwon, J., Lee, J., Kim, I., Lee, T., Han, J., Yoon, H.J., Cho, J., Han, D.: Single-trace attacks on message encoding in lattice-based KEMs. IEEE Access **8**, 183175–183191 (2020). https://doi.org/10.1109/ACCESS.2020.3029521
38. STMicroelectronics: (June 2024), https://www.st.com/resource/en/reference_manual/rm0090-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf
39. Wang, R., Dubrova, E.: A shared key recovery attack on a masked implementation of CRYSTALS-Kyber's encapsulation algorithm. In: Mosbah, M., Sèdes, F., Tawbi, N., Ahmed, T., Boulahia-Cuppens, N., García-Alfaro, J. (eds.) Foundations and Practice of Security - 16th International Symposium, FPS 2023, Bordeaux, France, December 11-13, 2023, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 14551, pp. 424–439. Springer (2023). https://doi.org/10.1007/978-3-031-57537-2_26
40. Yang, K., Sarkar, P., Weng, C., Wang, X.: Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 2986–3001. ACM (2021). https://doi.org/10.1145/3460120.3484556