Exploiting Matrix-Vector Multiplication in AIMer: Single-Trace Attacks on Key Generation and Signing

Süleyman Emir Akın[1][0009−0002−7684−4763],
Abdullah Talayhan[2][0009−0001−3252−5245], and
Özcan Öztürk[1][0000−0002−6870−8430]

[1] Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Türkiye
{emir.akin, ozcan.ozturk}@sabanciuniv.edu
[2] Computer Engineering Department, Bilkent University, Ankara, Türkiye
talayhan@bilkent.edu.tr

Abstract. This paper presents two side-channel attacks against AIMer v3.0, one of the two signature schemes selected in the Korean Post-Quantum Cryptography (KpqC) competition. Both are single-trace attacks that recover the full secret key without a profiling phase, one against key generation and one against signing. Both target the same routine: the matrix-vector multiplication invoked by either phase expands each bit of its secret operand into an all-zero or an all-one mask, which then gates the accumulation of the corresponding public matrix row. A single execution therefore leaks one distinguishable segment per secret bit. What makes the attacks non-profiled is that each attacked trace carries its own labelled training set. During key generation, creating the dense invertible public matrices invokes the matrix-vector multiplication routine hundreds of times, on operands that are public. In signing the same derivation takes place, and in addition the signature itself opens all but one of the N parties of every repetition, so the bits of nearly every multiplication in the trace are public. In both attacks we use a supervised classifier to recover the unknown value. In key generation that value is the private key itself, whereas in signing there are three distinct routes to it: the key itself, a party's share of the key, and a party's share of an S-box output. Recovering any one of them is enough to recover the private key. We validate both attacks on an ARM Cortex-M4: one trace recovers the exact secret key from every capture we took, 100 key generations for each of AES-128, AES-192 and AES-256, and 100 signatures for each of AES-128f, AES-192f and AES-256f. We also discuss possible countermeasures.

Keywords: Side-channel attacks · Post-quantum cryptography · AIMer

1 Introduction

With the rise of quantum computers, traditional signature schemes such as RSA [32], DSA [27], and ECDSA [17] have become vulnerable because Shor's algorithm [33,34] can efficiently solve the integer factorization and discrete logarithm

problems on which they rely. To address this threat, South Korea launched its Korean Post-Quantum Cryptography (KpqC) standardization process in 2021.

In January 2025, KpqC announced its final winners: two key-encapsulation mechanisms (KEMs), SMAUG-T [6] and NTRU+ [18], and two digital signature schemes, HAETAE [5] and AIMer [9]. HAETAE is a lattice-based signature scheme, whereas AIMer is obtained from a non-interactive zero-knowledge proof of knowledge (NIZKPoK) of a preimage for a tweakable one-way function, following the MPC-in-the-Head (MPCitH) paradigm [15]. AIMer's security depends on the one-wayness of the underlying primitive rather than on a hard lattice or code-based problem.

AIMer v1.0 [7], built on the one-way function AIM, was also submitted to NIST's call for additional post-quantum digital signature schemes. After algebraic weaknesses were identified in AIM [24,35], it did not advance to the second round. The same weaknesses motivated the AIMer team to redesign the primitive as AIM2.

The AIM2 primitive underlying AIMer has been the target of purely mathematical cryptanalysis. [4] proposed a new algebraic elimination framework, based on partial linearization over a univariate polynomial ring and a new method for determining a null vector of a polynomial matrix, and applied it to full AIM2. In the standard setting, their attack reduces AIM2's claimed security level; in the reused-key and related-key settings, key recovery becomes practical. In response, the AIMer team introduced AIM3 and AIMer v3.0 [8].

Our contributions are as follows:

- We present two non-profiled, single-trace attacks against AIMer v3.0 that recover the full secret key, one against key generation and one against signing.
- We validate both attacks on an ARM Cortex-M4 microcontroller and recover the exact secret key from a single power trace in all 100 key generations we captured for each of AES-128, AES-192 and AES-256, and in all 100 signatures we captured for each of AES-128f, AES-192f and AES-256f[3].
- We discuss possible countermeasures.

2 Related Work

AIMer has already been the subject of two physical-attack studies, both short papers presented in Korean at KIISC conferences. [13] reports side-channel and fault-injection attacks on AIMer as a KpqC candidate, and is cited by the AIMer specification itself where it discusses side-channel security [9]. [12] later reports a secret-key-recovery side-channel analysis of the Round 2, AIM2-based version, the predecessor of the version we target. Their papers are not available online, and therefore we do not compare our results with theirs in detail. The leakage we exploit and the type of function we target have, in turn, been studied in the broader literature.

[3] The AES-128, AES-192 and AES-256 parameter sets correspond to NIST Security Categories 1, 3 and 5, respectively.

[11] attacks the field multiplication `gf_mul` used inside the vector-matrix multiplication in SDitH [26], targeting the threshold (TCitH) variant of the first-round submission. Like AIMer, SDitH is built on the MPCitH paradigm, and it has since advanced to the third round of NIST's call for additional digital signature schemes [2]. Their multiplication leaks similarly to our matrix-vector multiplication, since both are branch-free schoolbook implementations in which one operand is scanned bit by bit and a bitmask gates the accumulation of the other; in their case, however, the bit-scanned operand is the public one and the secret share is manipulated as a full byte. Moreover, each individual leak in their attack only constrains a byte to one of nine Hamming weight classes, whereas our attack recovers a specific bit directly. Their attack is a profiled, single-trace attack that builds Hamming weight templates on the multiplication's first operand and output, and combines them by belief propagation to recover the key. Our attacks likewise classify with a supervised model, but need no profiling phase to obtain one: the trace under attack already discloses its own labels, from the derivation of the public linear layer in key generation and from the opened parties in signing.

Similar bitwise-masking leaks are not specific to MPCitH-based schemes. [16] introduces a side-channel attack on a masked implementation of FAEST, built on the VOLEitH paradigm [3], and shows that the bitwise-masking leak in the implementation's bit-multiplication routine is enough to recover the secret key. Similarly, [30] introduces a single-trace side-channel attack on FALCON (now FN-DSA), a lattice-based (NTRU) signature scheme, showing that the same bitwise-masking leak occurs in the discrete Gaussian sampling routine used during its key generation and is sufficient to recover the private key.

Now that we have talked about the leakages related to our attack, it is time to shift the axis and talk about how we exploit the leakage. As we have said, our attack is single trace with no profiling required, and we exploit it in a supervised manner, using the labelled traces in a given key generation or signing trace from public data. We are not the first to exploit labels from public data. [29] exploits it using public data from the signature. They call this method Public Template Attack (PTA), and they also note that their prior attack [25] meets the definition of PTA. Our attack also meets the definition of PTA: we capture traces and classify them using public information, then build our template, and then use that template to classify the segments that carry the secret. But what if the public data is absent, or there is no public information at all because of the construction of the algorithm? Luckily, there exist unsupervised non-profiled single trace attacks too [14]; we will talk about this paper in depth in Section 4.2, where we used their search algorithm when the recovered values are not all correct.

3 Background

In this section, we first provide an overview of side-channel analysis, then introduce the notation and mathematical functions used in AIMer's key generation

and signing procedures, describe the key generation and signing processes, and finally conclude with our threat model.

3.1 Side-Channel Analysis

Side-channel analysis (SCA) recovers the secret by looking at other measurements instead of attacking it purely mathematically. Most commonly, this is done by analyzing the power consumption [22] of a device captured as a time series, known as a power trace. Since the power drawn at each cycle depends on the data being manipulated, traces leak information about the data. Beyond power consumption, other side-channel vectors include execution timing [23], electromagnetic emanations [31], branch-prediction behavior [1], and cache-timing effects [28].

Power SCA attacks can be categorized into profiled and non-profiled attacks. In profiled attacks, the attacker is assumed to have access to a profiling device where they can execute the target implementation with known inputs, allowing them to build a stronger statistical leakage model compared to non-profiled attacks. In non-profiled attacks, the attacker does not rely on such a profiling phase and instead directly analyzes leakage from the target device.

Power SCA can also be categorized into vertical and horizontal attacks [10]. In vertical attacks, multiple traces are captured under the same setup to reduce noise, whereas in horizontal attacks, multiple leakage points within a single trace are exploited. The attacks presented in this paper are non-profiled, horizontal, and single-trace.

3.2 Notation and Mathematical Preliminaries

We first fix the notation used throughout the paper and recall the few facts about $\mathbb{F}_{2^n}$ that our attack relies on.

Let $\mathbb{F}_2 = \{0, 1\}$ denote the binary field, whose addition and multiplication are bitwise XOR and logical AND. We write $[m]$ for the set $\{0, \ldots, m-1\}$. The security parameter λ equals the extension degree n of the field AIMer operates over, so we use the two interchangeably, and $n \in \{128, 192, 256\}$. AIMer works over $\mathbb{F}_{2^n} = \mathbb{F}_2[X]/\langle f_n(X)\rangle$, with the irreducible polynomials f_n fixed by the specification [8]. Its design mixes bitwise operations with field arithmetic, so we identify an n-bit object a with the polynomial $\sum_i a_i X^i \in \mathbb{F}_{2^n}$ and with the column vector $(a_0, \ldots, a_{n-1})^\top \in \mathbb{F}_2^n$, write a_i for its i-th bit throughout, and let context disambiguate which representation is meant.

The nonlinear layer of AIM3 is built from the *enhanced Mersenne S-box* $S[e] : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$,

$$S[e](x) = x^{2^e-1} + x^{2^n-2},$$

whose exponent e is public. $S[e]$ is not invertible, but an output of it has at most two preimages and both computeable. For details on the S-box we refer the reader to Section 3.2 of the specification [8].

Finally, we describe the linear layer of AIMer's design explicitly in terms of matrix–vector multiplication over $\mathbb{F}_2$, in order to fix the bit-ordering conventions used later in the paper. For a matrix $A \in \mathbb{F}_2^{n \times n}$ and a vector $x \in \mathbb{F}_2^n$, the product $Ax \in \mathbb{F}_2^n$ is the XOR of the columns of A selected by the nonzero bits of x,

$$Ax = \bigoplus_{c\,:\,x_c=1} A[\cdot][c],$$

where x_c denotes the c-th entry of x and $A[\cdot][c]$ the c-th column of A. This is the form the reference implementation computes, and the form our attack exploits.

3.3 AIMer Key Generation

The key generation algorithm of AIMer (Algorithm 1) takes a parameter λ which specifies the security level. It first samples a secret key pt and a public initialization vector iv uniformly at random from $\{0,1\}^\lambda$, and executes the AIM3 function on this pair to obtain the public-key component ct (Algorithm 2). AIM3 returns $\perp$ if any of its S-boxes receives 0 as input, in which case the sampling is repeated.

The AIM3 algorithm first calls AIM3_GenerateLinear(iv), which, from the public iv, samples 2ℓ pairs of unit lower triangular and upper triangular matrices from $\mathbb{F}_2^{\lambda\times\lambda}$ and $\ell+1$ vectors $(b_{iv,j})_{j\in[\ell+1]}$ from $\mathbb{F}_2^\lambda$, and sets each matrix $A_{iv,j}$ to the product $L_{iv,j} \cdot U_{iv,j}$ of one such pair. Sampling two unit triangular matrices rather than a single general matrix guarantees invertibility, as required by the specification [8]. The reference implementation forms each of these products one row at a time, with the same matrix–vector routine the rest of the scheme uses.

Next, a for loop runs over the ℓ S-box branches, $j \in [\ell]$, where $\ell = 2$ for AES-128 and AES-192 and $\ell = 3$ for AES-256. The iterations are independent of one another, and within each iteration two operations are carried out in sequence. First, the secret key pt is multiplied by the public matrix $A_{iv,j}$ and the public vector $b_{iv,j}$ is added, with the result stored in x_j, the j-th S-box input. Then, x_j is passed through the S-box $S[e_j]$, where e_j is public, with the output stored in y_j, the j-th S-box output.

Once the loop ends, the ℓ S-box outputs are combined by a second affine layer into $x_\ell \leftarrow b_{iv,\ell} + \sum_{j\in[\ell]} A_{iv,j+\ell} \cdot y_j$. This value is passed through the S-box $S[e_\ell]$, where e_ℓ is likewise public, and the secret key pt is XORed in once more. The resulting value is the output ct, which is then returned. Thus, the secret key consists of (pt, iv, ct), while the public key consists of (iv, ct). Nevertheless, the only truly private component is pt, since both iv and ct are also part of the public key.

3.4 AIMer Signing

The signing algorithm AIMer_sign(sk, M, ctx) prepends the context string and its length to the message, samples the randomness ρ, and passes both to the internal signing algorithm AIMer_sign_internal(sk, M', ρ) (Algorithm 3), which

Algorithm 1: AIMer_keygen(λ)

repeat
  $pt \xleftarrow{\$} \{0,1\}^\lambda$;
  $iv \xleftarrow{\$} \{0,1\}^\lambda$;
  $ct \leftarrow \mathrm{AIM3}(pt, iv)$;
until $ct \neq \perp$;
$sk \leftarrow (pt, iv, ct)$;
$pk \leftarrow (iv, ct)$;
return (sk, pk)

Algorithm 2: AIM3(pt, iv)

$\big((A_{iv,j})_{j\in[2\ell]}, (b_{iv,j})_{j\in[\ell+1]}\big) \leftarrow$ AIM3_GenerateLinear(iv);
for $j \in [\ell]$ **do**
  $x_j \leftarrow A_{iv,j} \cdot pt + b_{iv,j}$;
  $y_j \leftarrow S[e_j](x_j)$;
$x_\ell \leftarrow b_{iv,\ell} + \sum_{j\in[\ell]} A_{iv,j+\ell} \cdot y_j$;
$y_\ell \leftarrow S[e_\ell](x_\ell)$;
$ct \leftarrow pt \oplus y_\ell$;
if $x_j = 0$ *for some* $j \in [\ell+1]$ **then return** $\perp$;
return ct

proves knowledge of pt by splitting it into shares among N virtual parties, simulating an MPC protocol in which they jointly verify $\mathrm{AIM3}(pt, iv) = ct$, and repeating that simulation τ times. The internal algorithm runs in five phases; only the first, in which the prover commits to the seeds and to the parties' execution views, is relevant to us, and the only one we describe in detail. We refer the reader to the specification [8] for the remaining four.

Phase 1 begins by hashing the message together with the public key into μ, and by deriving the linear components L from the public iv. These are the same 2ℓ matrices $(A_{iv,j})_{j\in[2\ell]}$ and $\ell+1$ vectors $(b_{iv,j})_{j\in[\ell+1]}$ that key generation uses, derived in the same way. Phase 1 then computes the $\ell+1$ S-box outputs $(y_j)_{j\in[\ell+1]} \leftarrow$ AIM3_SboxOutputs(L, pt, ct) (Algorithm 4). These are exactly the values y_j of Algorithm 2: signing recomputes the same intermediates as key generation, and in doing so multiplies the unshared secret key by the ℓ matrices of the input affine layer. Finally, a salt and one root seed per repetition are squeezed from $H_3(\mu, pt, \rho)$.

Each of the τ repetitions that follow produces its own independent sharing of the same values. In each of them, the root seed is expanded into a binary tree whose N leaves are the parties' seeds, and every leaf is expanded in turn into that party's commitment and random tape. The tape carries the party's shares of the secret key and of the S-box outputs, $pt_k^{(i)}$ and $(y_{k,j}^{(i)})_{j\in[\ell]}$, where $y_{k,j}^{(i)}$ denotes the share of y_j held by party i in repetition k, together with the further

values consumed by the multiplication check of a later phase. The sharing is additive over $\mathbb{F}_{2^n}$: a single share, or any $N-1$ of them, is indistinguishable from random, and the secret appears only in their sum,

$$pt = \sum_{i\in[N]} pt_k^{(i)}, \qquad y_j = \sum_{i\in[N]} y_{k,j}^{(i)} \qquad \text{for every } k \in [\tau].$$

Drawn straight from the tapes the shares do not satisfy this, so the prover computes the offset $\Delta y_{k,j} = y_j - \sum_i y_{k,j}^{(i)}$, adds it to the last party's share; the shares of pt are corrected in the same way. The shares are not fully secret: for every repetition the finished signature carries the salt and all the seeds required to produce every share but one. Each party then runs AIM3_MPC on its own shares, with the constant vectors $(b_{iv,j})_{j\in[\ell+1]}$ added by the last party only.

AIM3_MPC (Algorithm 5) is the shared-data counterpart of AIM3. A party first applies the input affine layer to its share of the secret key, setting $x_j^{(\cdot)} \leftarrow A_{iv,j} \cdot pt^{(\cdot)}$ for $j \in [\ell]$, and then the output affine layer to its shares of the S-box outputs, accumulating $x_\ell^{(\cdot)} \leftarrow \sum_{j\in[\ell]} A_{iv,j+\ell} \cdot y_j^{(\cdot)}$. It computes $z_j^{(\cdot)} = \left(x_j^{(\cdot)}\right)^{2^{e_j}}$; squaring is $\mathbb{F}_2$-linear, so a party can compute this from its own share alone. It then outputs the $\ell+1$ triples $(x_j^{(\cdot)}, y_j^{(\cdot)}, z_j^{(\cdot)})$ that the multiplication check consumes. The two affine layers are where a public matrix meets a secret share, and these multiplications, $A_{iv,j} \cdot pt_k^{(i)}$ and $A_{iv,j+\ell} \cdot y_{k,j}^{(i)}$, are our target.

A party therefore performs 2ℓ matrix-vector multiplications, as many as key generation. Together with the ℓ that AIM3_SboxOutputs performs on the unshared secret key, one signature thus contains three kinds of multiplication whose operand leads back to pt: pt itself, a party's share of pt, and a party's share of an S-box output y_j. Recovering the operand of any one of them is enough. The two kinds of share are recovered in the same way, through the sums above: a pt share yields pt directly, and a y_j share yields y_j, from which pt follows by inverting the S-box and the input affine layer. Since every repetition shares the same values, the one share that is secret in any single repetition is enough to recover them.

For more information on AIMer, the reader is referred to the following references [9, 19–21].

3.5 Threat Model

The adversary is assumed to have physical access to the target device and to be able to measure its power consumption. We further assume that the adversary knows the software executed by the device, and can approximate the timing of the computations. Finally, the adversary is assumed to know the public key (iv, ct).

During the attack phase, the adversary observes a single execution on that device and must recover the secret key pt from the resulting trace: in the first attack, a key generation with an unknown seed; in the second, the signing of a single message, whose signature the adversary also obtains, as any verifier would. The adversary does not choose this message. Our attacks require no profiling, neither on this device nor on any other.

Algorithm 3: AIMer_sign_internal($sk = (pt, iv, ct), M', \rho$) - AIMer signature scheme, internal signing algorithm (phase 1).

// *Phase 1: Committing to the seeds and the execution views.*;
Compute the hash of the message: $\mu \leftarrow H_0(iv, ct, M')$;
Derive the linear components: $L \leftarrow$ AIM3_GenerateLinear(iv);
Compute the S-boxes' outputs: $(y_j)_{j\in[\ell+1]} \leftarrow$ AIM3_SboxOutputs(L, pt, ct);
Compute salt and root seeds: $(salt, (seed_k)_{k\in[\tau]}) \leftarrow H_3(\mu, pt, \rho)$;
for *each repetition* $k \in [\tau]$ **do**
    Compute parties' tape and commit to it:;
        $nodes_k \leftarrow$ ExpandTree($salt, k, seed_k$);
    for *each party* $i \in [N]$ **do**
        $seed_k^{(i)} \leftarrow nodes_k[N-1+i]$;
        $\left(com_k^{(i)}, pt_k^{(i)}, (y_{k,j}^{(i)})_{j\in[\ell]}, (a_{k,j}^{(i)})_{j\in[\ell+1]}, c_k^{(i)}\right) \leftarrow H_5(salt, k, i, seed_k^{(i)})$;
    Compute offsets and adjust last shares for witness shares:;
    $\Delta pt_k \leftarrow pt - \sum_i pt_k^{(i)}, \quad pt_k^{(N-1)} \leftarrow pt_k^{(N-1)} + \Delta pt_k$;
    for $j \in [\ell]$, $\Delta y_{k,j} \leftarrow y_j - \sum_i y_{k,j}^{(i)}, \quad y_{k,j}^{(N-1)} \leftarrow y_{k,j}^{(N-1)} + \Delta y_{k,j}$;
    $\Delta c_k \leftarrow \sum_{j\in[\ell+1]}\left(\sum_i a_{k,j}^{(i)} \cdot y_j\right) - \sum_i c_k^{(i)}, \quad c_k^{(N-1)} \leftarrow c_k^{(N-1)} + \Delta c_k$;
    for *each party* $i \in [N]$ **do**
        Prepare the multiplication check inputs by MPC simulation:;
            $(x_{k,j}^{(i)}, y_{k,j}^{(i)}, z_{k,j}^{(i)})_{j\in[\ell+1]} \leftarrow$ AIM3_MPC$\left(L, ct, pt_k^{(i)}, (y_{k,j}^{(i)})_{j\in[\ell]}, i\right)$;
Set $\sigma_1 \leftarrow \left(salt, ((com_k^{(i)})_{i\in[N]}, \Delta pt_k, (\Delta y_{k,j})_{j\in[\ell]}, \Delta c_k)_{k\in[\tau]}\right)$;
// *Phases 2–5 follow.*;

4 The Proposed Attack

The attack we present recovers the input of each matrix-vector multiplication, those of Section 3.3 in key generation and those of Section 3.4 in signing, via a single-trace power side-channel attack on that multiplication, without requiring a profiling phase.

For each security parameter set, the AIMer reference implementation provides a separate, independently optimized C routine for gf_mat_vec_mul; nevertheless, all of these routines rely on the same bitwise masking technique and the same underlying algorithm, differing only in low-level implementation choices such as loop granularity. Without loss of generality, we therefore present our attack against the gf_mat_vec_mul code written for the AES-128 security parameter set.

4.1 Matrix-Vector Multiplication

In this section we will first describe C code specifically written for AES-128 in detail. After that, we will describe how the code changes at different parameter sizes.

Algorithm 4: AIM3_SboxOutputs($L = ((A_{iv,j})_{j\in[2\ell]}, (b_{iv,j})_{j\in[\ell+1]})$, pt, ct) - compute the S-boxes' outputs.

for $j \in [\ell]$ **do**
  Set $x_j \leftarrow A_{iv,j} \cdot pt + b_{iv,j}$;
  Set $y_j \leftarrow S[e_j](x_j)$;
Set $y_\ell \leftarrow pt + ct$;
Output $(y_j)_{j\in[\ell+1]}$;

Algorithm 5: AIM3_MPC($L = ((A_{iv,j})_{j\in[2\ell]}, (b_{iv,j})_{j\in[\ell+1]})$, ct, $pt^{(\cdot)}$, $(y_j^{(\cdot)})_{j\in[\ell]}$, i) - run the MPC simulation.

for $j \in [\ell]$ **do**
  Set $x_j^{(\cdot)} \leftarrow A_{iv,j} \cdot pt^{(\cdot)}$;
Set $x_\ell^{(\cdot)} \leftarrow \sum_{j\in[\ell]} A_{iv,j+\ell} \cdot y_j^{(\cdot)}$;
Set $y_\ell^{(\cdot)} \leftarrow pt^{(\cdot)}$;
if $i = N - 1$ **then**
  for $j \in [\ell + 1]$ **do**
    $x_j^{(\cdot)} \leftarrow x_j^{(\cdot)} + b_{iv,j}$;
  $y_\ell^{(\cdot)} \leftarrow y_\ell^{(\cdot)} + ct$;
for $j \in [\ell + 1]$ **do**
  Set $z_j^{(\cdot)} \leftarrow \left(x_j^{(\cdot)}\right)^{2^{e_j}}$;
  if $i = N - 1$ **then**
    $z_j^{(\cdot)} \leftarrow z_j^{(\cdot)} + 1$;
Output $(x_j^{(\cdot)}, y_j^{(\cdot)}, z_j^{(\cdot)})_{j\in[\ell+1]}$;

Algorithm 6 gives gf_mat_vec_mul (for AES-128), which computes the matrix-vector product over $\mathbb{F}_2^{128}$ defined in Section 3.2. The type gf holds an element of $\mathbb{F}_2^{128}$ in two `uint64_t` entries; a matrix in $\mathbb{F}_2^{128\times128}$ is simply passed as an array of such rows. Notice that the matrix is given in transposed form, as the specification specifies [8]. Given $a \in \mathbb{F}_2^{128}$ and $b \in \mathbb{F}_2^{128\times128}$, the function writes $c = a \cdot b$ to the output.

The implementation is branchless (constant-time): rather than testing bits of a with a conditional, each bit is expanded into a full all-zero or all-one mask via `mask = 0U - ((index » p) & 1)`, and the corresponding row of b is XORed into the accumulator under this mask.

For AES-128, the outer loop runs twice, taking one entry of a at a time. The inner loop runs eight times per entry, consuming it one byte at a time: each iteration shifts it right by 8 bits to bring the next byte into position, and advances the matrix pointer `b_ptr` by 8 rows to keep it aligned with the corresponding rows of b. Rather than looping over the eight bits of each byte, the body is unrolled by hand into eight blocks, one per bit position $p \in \{0, \ldots, 7\}$; each block computes

the mask for bit p and accumulates the masked row into `temp_c0` and `temp_c1`, the two 64-bit halves of c. Once both loops finish, the accumulators are written back to c.

Algorithm 6: GF Matrix-Vector Multiplication

```
void gf_mat_vec_mul(gf c, const gf a, const gf b[128]) {
  const uint64_t *a_ptr = a; const gf *b_ptr = b;
  uint64_t temp_c0 = 0; uint64_t temp_c1 = 0; uint64_t mask;

  for (size_t i = 2; i; --i, ++a_ptr) {
    uint64_t index = *a_ptr;

    for (size_t j = 64; j; j -= 8, index >>= 8, b_ptr += 8) {

      mask = 0U - (index & 1);
      temp_c0 ^= (b_ptr[0][0] & mask);
      temp_c1 ^= (b_ptr[0][1] & mask);

      mask = 0U - ((index >> 1) & 1);
      temp_c0 ^= (b_ptr[1][0] & mask);
      temp_c1 ^= (b_ptr[1][1] & mask);

      // ... identical blocks for bits 2-7 ...
    }
  }
  c[0] = temp_c0;
  c[1] = temp_c1;
}
```

For AES-192 and AES-256, a is held in three and four `uint64_t` entries instead of two, so the outer loop runs three and four times, taking one entry per iteration; b has λ rows of λ bits. The inner loop processes 4 bits at a time instead of 8. Within each block, the masked row is XORed into three (`temp_c0`–`temp_c2`) or four (`temp_c0`–`temp_c3`) accumulators instead of two, matching the wider output.

4.2 The Leakage and Key Recovery

Leakage. As discussed in Section 4.1, Algorithm 6 processes its input vector bit by bit, and this bitwise structure introduces several points at which each processed bit can influence the device's power consumption. For the p-th bit of the value being processed, the algorithm first isolates it via `index » p`, masks it with `1` to obtain a single bit, and then negates it (`mask = 0U - ((index » p) & 1)`) to expand it into a full-width mask that is either all-zero or all-one. Because

the all-zero and all-one masks have maximally different Hamming weights, this negation induces a power consumption difference that depends on the value of the p-th bit.

This leakage is not confined to the mask computation. The resulting mask is subsequently ANDed with the two row values `b_ptr[p][0]` and `b_ptr[p][1]`, and the result is XORed into the accumulators `temp_c0` and `temp_c1`. Each of these AND and XOR operations again has a power profile that depends on the same p-th bit. Every bit of the input vector is processed this way, so one execution of the routine leaks each of its n input bits separately, whatever value it was called on.

Leakage Segmentation. Because our attack operates on a single trace, we cannot average away noise across many executions the way vertical attacks do; instead, we must recover the value of each secret bit from a single observation of the corresponding leakage. To do so, we first gather every trace segment that exhibits this leakage within one execution. Every call to Algorithm 6 processes its n-bit input one bit at a time and therefore yields n such segments; what differs between the two phases is only how many calls there are, and on what.

Before either phase can use them, the linear components have to be built, and this is where most of the calls are. As described in Section 3.3, each of the 2ℓ dense matrices $A_{iv,j}$ is the product $L_{iv,j} \cdot U_{iv,j}$ of two triangular matrices squeezed from iv, and the reference implementation forms that product one row at a time, calling the very routine we attack once per row of $U_{iv,j}$. The operand it scans bit by bit is therefore a row of $U_{iv,j}$, which anyone holding the public iv can recompute, so these $2\ell n$ calls leak exactly as Section 4.2 describes, on values that are not secret. Both phases derive the linear components this way, so both traces contain them.

In key generation, the routine is then called 2ℓ times on secret values: ℓ times by the input affine layer, on pt itself, and ℓ times by the output affine layer, on the S-box outputs y_j. A key generation therefore yields $2\ell n + 2\ell$ calls to segment, of which only 2ℓ have a secret operand – 512 against 4 at AES-128.

In signing, AIM3_SboxOutputs runs the routine ℓ times on the unshared pt, and AIM3_MPC runs it 2ℓ times per party and per repetition, on that party's share of pt and on its shares of the y_j. Since the signature opens all but one party of every repetition, the operands of nearly all of those calls are public as well.

Pinpointing Points of Interest. Locating the leakage described above within a captured trace splits into two distinct sub-problems: locating each innermost-loop iteration of Algorithm 6 in time, and, within one such iteration, locating the samples that leak the value of each individual bit.

The first sub-problem does not depend on the secret being processed, since the control flow of Algorithm 6 is identical for every execution regardless of its input. We resolved it by visual inspection of a captured trace: we determined the

start of the first outermost-loop iteration, the number of cycles one innermost-loop iteration takes, and the number of cycles separating consecutive outermost-loop iterations.

The second sub-problem, locating the point at which each bit is processed, is considerably harder. Here the trace answers it for us. As described above, most of the calls it contains have an operand we can recompute from the public key, so for each of them we know the value of every bit that was processed. We therefore pool, over all of those calls, the segments belonging to one bit position, label each segment by the bit it carried, and score every sample of the innermost-loop iteration by how well it separates the segments carrying a 0 from those carrying a 1, using Welch's t. We identified how many samples our setup captures per clock cycle, which is 4, see Section 5.1, so we average the score over each group of four and keep the six most correlated clock cycles for that bit position.

We repeat this for every bit position, and separately for every key generation and every signature we attack: the points of interest are read off the same trace they are then used on, rather than measured once and declared global. Note that this requires no profiling and no labelled data beyond what the attacked execution itself discloses, and that it never needs to know in what order the compiler processes the bits, since the search finds each position wherever it leaks.

One might question why we keep six cycles per bit position across all parameter sets. This is a hyperparameter, and it is probably not optimal either, but although we do have labels we did not want to try different numbers and report the best; we simply wanted one constant value for every parameter set. At AES-128 a bit costs one or two `sbfx` instructions to create the mask, and four `and` instructions to apply it to the accumulator. We did not want to exceed that count, and settled on six.

Recovering the Key from a Key Generation. Having gathered the trace segments of every bit position of the inner loop, we can now separate the two possible bit values. The calls that build the linear components label them for us: their operands are rows of $U_{iv,j}$, so for each of their segments we know the bit it carried. We fit a supervised classifier (nearest-centroid classifier or LDA) on those labelled segments of each bit position and let it classify the rest. Notice that being supervised is not the same as being profiled; our attack is non-profiled. The labels come from the single trace we attack, computed from public data.

We then apply it to the segments of the 2ℓ multiplications whose operand is secret. Each of them recovered bit for bit yields a candidate for the key. The ℓ multiplications of the input affine layer have pt itself as their operand, so they give the secret key directly. The ℓ of the output affine layer give the S-box output y_j, from which the key follows by inverting the S-box and the affine layer, both public; the S-box leaves at most two candidates.

We accept a candidate when recomputing AIM3(pt', iv) with the public iv reproduces the public ct, at one evaluation of AIM3 per candidate. Each deter-

mines the secret key on its own, so they are separate attempts and one success suffices.

But what if none of the candidates were successful, and we were only one bit mistake away from finding the key? What can we do? [14] introduced a clever search mechanism that can be used in our setup too. Their paper's methodology is similar to ours, where they use clustering algorithms k-means to mount non-profiled single-trace (single-execution) attacks. They target exponentiation algorithms, as used in RSA and elliptic curve cryptography, whose loops process independent exponent bits with an identical operation sequence per iteration. They cut a single trace into one segment per processed exponent bit and separate these segments into two clusters. This looks like our case, except that we have labelled data, so we use a nearest-centroid classifier or linear discriminant analysis instead of k-means, which is unsupervised. They also show that the clustering itself makes the remaining search efficient when some bits are misclassified: its posterior class-membership probabilities rank the bits by confidence, so the least confident ones can be enumerated first, rather than the entire exponent. This is what we do when none of the candidates is successful.

The search buys the missing bits with computation, each extra window approximately doubles the number of candidates to check. A stronger classifier costs nothing. We therefore report linear discriminant analysis alongside the nearest centroid.

Recovering the Key from a Signing. In signing the labels come from the signature itself. For every repetition it opens all but one of the N parties, so the shares those parties multiplied are public, and we fit the same classifiers on their segments and let them classify the segments of the party that stays closed.

As Section 3.4 describes, three kinds of multiplication lead back to pt, and recovering the operand of any one of them bit for bit is enough. The multiplications of AIM3_SboxOutputs have the unshared pt as their operand, so they give the key itself. A closed party's share of pt gives the key once the $N-1$ opened shares of that repetition are XORed onto it. A closed party's share of y_j gives y_j in the same way, and the key follows from it as in key generation.

Every candidate is checked against the public key exactly as in key generation. A signature offers many more attempts than a key generation does, since every repetition provides its own, and one success suffices.

5 Results

5.1 Setup

We target the official C reference implementation of AIMer[4], compiled with `arm-none-eabi-gcc -O2` for an `STM32F303RCT7` ARM Cortex-M4 clocked at 7.38 MHz. Power traces are captured with a ChipWhisperer-Lite (CW1173) at

[4] The code is available at `github.com/samsungsds-opensource/AIMer`.

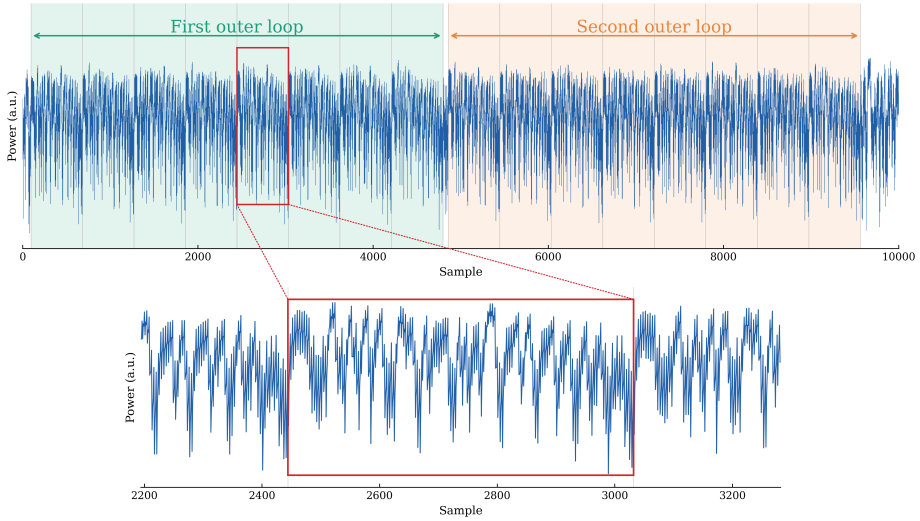


Fig. 1: A representative power trace of the gf_mat_vec_mul routine written for the AES-128 parameter set. The figure highlights two outermost-loop iterations in green and orange. One of the innermost-loop iterations is highlighted with a red rectangle and expanded below in detail.

29.54 MHz, yielding four samples per clock cycle. We attack the AES-128f ($\lambda = 128$, $\ell = 2$, $\tau = 33$, $N = 16$), AES-192f ($\lambda = 192$, $\ell = 2$, $\tau = 49$, $N = 16$) and AES-256f ($\lambda = 256$, $\ell = 3$, $\tau = 65$, $N = 16$) parameter sets. For each of them we acquire 100 key generations, every one with a different secret key, and one signing under each of those keys. Key generation depends only on the field size, so its results hold for the corresponding s parameter sets as well; signing does not, and we evaluate it on the f sets.

To fit the target's memory, the firmware we flash runs only the parts of the scheme our attack touches: key generation, and the first phase of signing reduced to AIM3_SboxOutputs and the per-party evaluations of AIM3_MPC. The commitments, the seed tree and the hashing are removed, and the shares are drawn from the reference implementation's random number generator rather than from the party tapes; both phases, in turn, derive the linear components one at a time instead of holding all 2ℓ matrices at once: key generation holds one matrix at a time, and signing one affine layer. None of this touches the multiplications we attack, including those that build the matrices: they compute exactly the same values as the unmodified code, and are only interleaved differently, since each matrix is now built just before it is used. The signing shares likewise remain pseudorandom field elements summing to y_j, as in a full signature. For every repetition we draw one party uniformly at random as the one whose share stays secret, and take the remaining $N - 1$ shares as known. All parties run the same code on their own share, and the scheme picks the secret one at

random, so our choice cannot favour the attack; the classifiers are fitted only on the multiplications whose shares we know.

Figure 1 shows a representative power trace of the gf_mat_vec_mul routine written for the AES-128 parameter set, captured during key generation.

5.2 Attack Results

We evaluate both attacks on the traces described in Section 5.1, running the full pipeline of Section 4 on each of them. A trace counts as a success only when a recovered candidate pt' is bit-for-bit identical to the secret key; partial recovery earns no credit.

Key Generation. Table 1 gives the results for key generation. The first three columns are the nearest centroid: the multiplications we recover exactly, the keys that follow from them directly, and the keys we obtain with the enumeration technique introduced in [14], with the window it takes to reach that rate in parentheses. The last two are linear discriminant analysis, which needs no enumeration at all.

With the nearest centroid we recover nearly half of the multiplications exactly at AES-128, and that number drops sharply at AES-192 and AES-256. Part of the reason is simply that the classifier has to get more bits right at once. That does not explain why AES-192 scores below AES-256, however, and the points of interest do: at AES-192 bit position 1 separates the two classes far less sharply than the others, so its error rate is correspondingly higher and it alone spoils most multiplications.

The second column is the key recovery rate straight from the recovered values, with no enumeration. We recover 89 keys at AES-128, only 2 at AES-192, and 39 at AES-256. The third column shows what the enumeration buys. A window of three already recovers every key at AES-128, at no more than $2^4 - 1 = 15$ trials per value. A window of ten, at no more than $2^{11} - 1 = 2\,047$ trials per value, recovers 70 and 95 of the keys at AES-192 and AES-256; every key at AES-256 is reached by a window of 26, while one key at AES-192 still resists a window of 32.

Linear discriminant analysis closes all of it. It recovers every multiplication at AES-128, 396 of 400 at AES-192 and 599 of 600 at AES-256, and with them every key at all three parameter sets, without a single enumeration step.

Signing. Table 2 gives the results for signing. The attack recovers the key from every signature at every parameter set, but the rates behind that are worth looking at in detail. The columns are the two classifiers, and for each of them the percentage of target multiplications recovered exactly and the keys that follow from them. Even though the nearest centroid already recovers all of the keys at every parameter set, we report LDA as well, to stay on the same page as key generation and to gauge how the multiplication accuracy rate would change. For the nearest centroid that rate drops as the parameter set increases, which is

Table 1: Key-generation results, over 100 key generations per security level.

	Nearest centroid			LDA	
Parameter set	Multiplications recovered	Keys	Keys (+ search)	Multiplications recovered	Keys
AES-128	193/400	89	100 ($w = 3$)	400/400	100
AES-192	2/400	2	70 ($w = 10$)	396/400	100
AES-256	48/600	39	95 ($w = 10$)	599/600	100

expected: it makes mistakes, and in the larger the parameter set the more values it has to find correctly at the same time. However, LDA is very powerful in our setup, and recovers all of the multiplications regardless of the parameter set.

Table 2: Signing results, over 100 signatures per parameter set.

	Nearest centroid		LDA	
Parameter set	Multiplications recovered	Keys recovered	Multiplications recovered	Keys recovered
AES-128f	98.3%	100	100%	100
AES-192f	78.8%	100	100%	100
AES-256f	57.0%	100	100%	100

The s Parameter Sets. AIMer specifies two variants at every security level, which differ only in how the MPC simulation is parameterized. The f (fast) variant runs $N = 16$ parties per repetition, the s (short) variant $N = 256$. Table 3 lists both. Neither τ nor N enters key generation, so Table 1 holds for the s sets unchanged and only signing is affected.

The party count increases from 16 to 256. Only one party is closed, so this changes nothing and provides more labelled multiplications. The repetition count nearly halves, which leaves fewer paths to the key. We therefore simulate an s set on the traces we captured for its f variant: we take only as many repetitions as the s set uses, which reproduces the number of attempts exactly. For the party count we cannot follow the same approach, because an s set uses sixteen times as many parties as we captured, so we keep the 16 parties we have and again decide at random which one stays closed in each repetition. Under this setup we still recover the private key from every signature at every security level.

One might ask why we did not capture the s sets directly. An s signature performs about seven times as many multiplications as its f variant, so its trace is about seven times as large. The hundred signatures behind our AES-256f results take 72 GB; the same hundred for AES-256s would take some 485 GB, more storage than we have. The traces are stored as `int16`, two bytes a sample.

Table 3: Signing parameters of the six parameter sets, given as f/s.

Parameter set	τ	N	ℓ
AES-128 (f/s)	33/17	16/256	2
AES-192 (f/s)	49/25	16/256	2
AES-256 (f/s)	65/33	16/256	3

6 Countermeasures

In this section we give a brief overview of potential countermeasures. The design and implementation of these measures are considered outside the scope of this paper.

One can mask the secret operand into d shares. However this does not seem to help, because the attacker can recover each share independently and then recover the secret. Notice that in signing the MPCitH sharing is itself a masking, and the opened shares are precisely what give us our labels. However, if every call of the routine is masked, including those on public operands such as the rows of $U_{iv,j}$, the self-labelled data vanishes, so for non-profiled attacks masking could make the attack harder. Locating the points of interest, as described in Section 4.2, also relies on these labels, which makes the attack harder still. Nonetheless, one can use unsupervised approaches to distinguish 0 and 1 bits, as [14] did. Therefore, we believe masking alone is not a strong countermeasure. The same holds for blinding.

Shuffling is another common alternative, which permutes the order of operations. In our case, the order in which the bits are processed could be permuted. The leakage remains, but the attacker cannot match bits to their real positions, which is what our attack needs. Furthermore, if every call is shuffled, the self-labelled data vanishes as well, since the attacker no longer knows which bit of a public operand a segment carries. Locating the points of interest, as described in Section 4.2, also relies on these labels, which makes the attack even harder. However, the shuffling mechanism itself must be side-channel resistant. Otherwise, the attacker can first attack the shuffle (most commonly Fisher–Yates), then match the bits to their correct positions and continue. Implementing a side-channel resistant Fisher–Yates shuffle is not trivial, at least in software, and is a dedicated research topic of its own, outside the scope of this paper.

Moreover, the index loads during the processing of the vector could leak parts of the permutation, most directly through the Hamming weight of each index. To prevent this, the indices could be shifted randomly using C's pointer arithmetic. Yet the details do not end here. Because of timing constraints, one might want to shuffle only the inner loop, to balance security and speed. In that case the self-labelled data does not vanish completely, since the attacker still knows the Hamming weight of every group of bits in the public calls. In addition, if the granularity is only 4, the indices have Hamming weights 0, 1, 1 and 2, so the attacker can use this to detect some of them and enumerate the rest. Therefore,

we believe the indices should be shifted as described above, and the granularity should be chosen somewhat higher, at least 8 as in AES-128.

Nonetheless, for signing there is a difference compared to key generation. The attacker has multiple paths to the private key, as many as the number of repetitions, and in each of them the same secret is hidden under a different public offset. Suppose only the inner loop is shuffled, with granularity $m = 8, 4, 4$ for AES-128, AES-192 and AES-256, respectively (Section 4.1). Fix one such group, let $v \in \{0,1\}^m$ be its bits in pt, and $r_k \in \{0,1\}^m$ its bits in the XOR of the $N-1$ shares opened in repetition k, which is public. The closed party's share then carries $v \oplus r_k$ in this group, and although the shuffle hides the order of these bits, the classifier still yields their Hamming weight h_k. Since $\mathrm{HW}(v \oplus r_k) = \mathrm{HW}(v) + \mathrm{HW}(r_k) - 2\,\mathrm{HW}(v \wedge r_k)$, and both $\mathrm{HW}(v)$ and $\mathrm{HW}(v \wedge r_k)$ are sums of bits of v, each repetition gives an equation that is linear in v. Writing v_q and $r_{k,q}$ for the q-th bits of v and r_k, it reads

$$h_k = \sum_q v_q + \mathrm{HW}(r_k) - 2\sum_q r_{k,q}\, v_q \quad \Longrightarrow \quad \sum_q (1 - 2r_{k,q})\, v_q = h_k - \mathrm{HW}(r_k).$$

We stack the τ equations as $Rv = u$, where $R \in \{\pm 1\}^{\tau \times m}$ has the entries $1-2r_{k,q}$ and $u \in \mathbb{Z}^\tau$ the entries $h_k - \mathrm{HW}(r_k)$. Since R is not square, we multiply both sides by $R^\top$; if $\mathrm{rank}(R) = m$, the $m \times m$ matrix $R^\top R$ is invertible and the solution is unique:

$$v = (R^\top R)^{-1} R^\top u.$$

Otherwise, $m - \mathrm{rank}(R)$ entries of v are free variables, which the attacker can enumerate. Since $\tau \geq 17$ at every parameter set, far more than $m \leq 8$, R has full rank in practice, and repeating this for every group recovers pt. These equations assume that every h_k is exact, and our results in Section 5.2 show that the classifier can indeed recover such values exactly. Nonetheless, in practice this may not always hold, because of jitter, misalignment, interrupts or noise etc. The attacker can still cope with this by using the classifier's posterior probabilities rather than its hard decisions: each h_k is replaced by the expected Hamming weight of the group, the sum of the probabilities that its bits carry a 1, and each equation is weighted by how confident the classifier is in it. Solving this weighted least-squares problem and rounding each entry of v to the nearest bit gives a candidate; if the check against ct fails, the entries closest to $1/2$ are flipped first, as in [14]. Moreover, since the granularity is small, the attacker can even skip the linear algebra and try all 2^m candidates of each group, at most $2^8 = 256$, keeping the one that best agrees with the τ observed Hamming weights. The groups are independent, so this costs only $(n/m) \cdot 2^m$ trials in total.

Nonetheless, as we said, this attack only holds for signing, and in signing one could fully shuffle the matrix-vector multiplications. However, even full shuffling does not fully close this path. The equations no longer exactly determine v (in this case private key), but since the shares are uniformly random, R has rank τ in practice, and the attacker is left with $2^{n-\tau}$ candidates to enumerate, which is below the claimed security level at every parameter set. Furthermore, the multiplications of AIM3_SboxOutputs operate on pt itself directly, so attacking

them directly reveals Hamming Weight of private key, which gives one more equation and information and makes the attack easier, having said that we did not use it. Lastly, since the private key is uniformly distributed, the actual cost may be considerably lower.

To summarize, we believe that shuffling the inner loop is a fast way to protect key generation against the attack presented in this paper, although it still leaks the Hamming weight of every group and therefore reduces the search space. However, for signing, even full shuffling is not enough on its own, and it should be combined with other countermeasures such as masking. Therefore, we believe AIMer needs a dedicated study of how to protect it against this attack, and the implementation of its matrix-vector multiplication may need to be reconsidered.

7 Conclusion

We presented a side-channel analysis of AIMer v3.0, one of the two signature schemes selected in the Korean Post-Quantum Cryptography (KpqC) competition. We described two non-profiled, single-trace attacks, one against key generation and one against signing. Both target the matrix-vector multiplication that each phase invokes, which processes its secret operand one bit at a time and therefore leaks each of those bits separately. On an ARM Cortex-M4 both attacks recovered the exact key from every trace we captured: 100 key generations for each of AES-128, AES-192 and AES-256, and 100 signatures for each of AES-128f, AES-192f and AES-256f. Finally, we discussed possible countermeasures.

References

1. Acıiçmez, O., Koç, Ç.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: Cryptographers' Track at the RSA Conference. pp. 225–242. Springer (2007)
2. Alagic, G., Bros, M., Ciadoux, P., Dang, Q., Dang, T.H., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., et al.: Status report on the second round of the additional digital signature schemes for the nist post-quantum cryptography standardization process. Tech. rep., Tech. Rep. NIST IR 8610, National Institute of Standards and Technology (2026)
3. 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: Annual international cryptology conference. pp. 581–615. Springer (2023)
4. Biryukov, A., García Fernández, P., Udovenko, A.: Magic pot: Cryptanalysis of full aim2 in the standard and related-/reused-key settings using new elimination framework. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 63–90. Springer (2026)
5. Cheon, J.H., Choe, H., Devevey, J., Güneysu, T., Hong, D., Krausz, M., Land, G., Möller, M., Stehlé, D., Yi, M.: Haetae: Shorter lattice-based fiat-shamir signatures. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(3), 25–75 (2024)

6. Cheon, J.H., Choe, H., Hong, D., Hong, J., Seong, H., Shin, J., Yi, M.: Smaug: The key exchange algorithm based on module-lwe and module-lwr. Algorithm Specifications Version **3** (2024)
7. Cho, J., Cho, M., Ha, J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, J., Lee, S., Moon, D.: The aimer signature scheme–submission to the nist pqc project. version 1.0–1st june 2023, 2023
8. Cho, J., Ha, J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, J., Lee, S., Moon, D., Son, M., Yoon, H.: The AIMer signature scheme. Specification, Version 3.0 (Jul 2026), https://aimer-signature.org/docs/AIMer-specification-v3%20(Eng).pdf, version 3.0, 31 July 2026
9. Cho, J., Ha, J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, J., Lee, S., Moon, D., Son, M., et al.: The aimer signature scheme
10. Clavier, C., Feix, B., Gagnerot, G., Roussellet, M., Verneuil, V.: Horizontal correlation analysis on exponentiation. In: International Conference on Information and Communications Security. pp. 46–61. Springer (2010)
11. Godard, J., Aragon, N., Gaborit, P., Loiseau, A., Maillard, J.: Single trace side-channel attack on the mpc-in-the-head framework. In: International Conference on Post-Quantum Cryptography. pp. 267–293. Springer (2025)
12. Han, J., Han, D.G.: Secret key recovery side-channel analysis technique for KpqC round 2 AIMer. In: Conference on Information Security and Cryptography Summer 2024 (CISC-S'24). Korea Institute of Information Security and Cryptology (2024), in Korean; title translated from the original
13. Han, J., Huh, J.W., Lee, S., Kwon, J., Han, D.G.: Side-channel and fault injection attacks on the KpqC digital signature candidate AIMer. In: Conference on Information Security and Cryptography Summer 2023 (CISC-S'23). Korea Institute of Information Security and Cryptology (2023), in Korean
14. Heyszl, J., Ibing, A., Mangard, S., De Santis, F., Sigl, G.: Clustering algorithms for non-profiled single-execution attacks on exponentiations. In: International Conference on Smart Card Research and Advanced Applications. pp. 79–93. Springer (2013)
15. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing. pp. 21–30 (2007)
16. Jendral, S., Dubrova, E.: Side-channel attacks on voleith signature schemes. IACR Communications in Cryptology **2**(4) (2026)
17. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**(1), 36–63 (2001)
18. Kim, J., Park, J.H.: Ntru+: Compact construction of ntru using simple encoding method. IEEE Transactions on Information Forensics and Security **18**, 4760–4774 (2023)
19. Kim, S., Ha, J., Son, M., Lee, B.: Efficacy and mitigation of the cryptanalysis on aim. Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep **1474**, 2023 (2023)
20. Kim, S., Ha, J., Son, M., Lee, B., Moon, D., Lee, J., Lee, S., Kwon, J., Cho, J., Yoon, H., et al.: Aim: symmetric primitive for shorter signatures with stronger security. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 401–415 (2023)
21. Kim, S., Lee, B., Son, M.: Relaxed vector commitment for shorter signatures. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 427–455. Springer (2025)
22. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual international cryptology conference. pp. 388–397. Springer (1999)

23. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Annual international cryptology conference. pp. 104–113. Springer (1996)
24. Liu, F., Mahzoun, M., Øygarden, M., Meier, W.: Algebraic attacks on rain and aim using equivalent representations. IACR Transactions on Symmetric Cryptology **2023**(4), 166–186 (2023)
25. Liu, Y., Zhou, Y., Sun, S., Wang, T., Zhang, R., Ming, J.: On the security of lattice-based fiat-shamir signatures in the presence of randomness leakage. IEEE Transactions on Information Forensics and Security **16**, 1868–1879 (2020)
26. Melchor, C.A., Bettaieb, S., Bidoux, L., Feneuil, T., Gaborit, P., Gama, N., et al.: The syndrome decoding in the head (sd-in-the-head) signature scheme algorithm specifications and supporting documentation–version 2.0. Second round submission to the NIST post-quantum cryptography call for additional digital signature schemes (2025)
27. Nist, C.: The digital signature standard. Communications of the ACM **35**(7), 36–40 (1992)
28. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive (2002)
29. Qiao, Z., Liu, Y., Zhou, Y., Ming, J., Jin, C., Li, H.: Practical public template attacks on crystals-dilithium with randomness leakages. IEEE Transactions on Information Forensics and Security **18**, 1–14 (2022)
30. Qiu, J., Aysu, A.: Shift snare: Uncovering secret keys in falcon via single-trace analysis. arXiv preprint arXiv:2504.00320 (2025)
31. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In: International Conference on Research in Smart Cards. pp. 200–210. Springer (2001)
32. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2), 120–126 (1978)
33. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th annual symposium on foundations of computer science. pp. 124–134. Ieee (1994)
34. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
35. Zhang, K., Wang, Q., Yu, Y., Guo, C., Cui, H.: Algebraic attacks on round-reduced rain and full aim-iii. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 285–310. Springer (2023)