The AIMer Signature Scheme

Submission to the NIST PQC project

Version 1.0

Principal Submitter:

Seongkwang Kim
Samsung SDS
sk39.kim@samsung.com
+82-10-9930-6241
56, Seongchon-gil, Seocho-gu, Seoul 06765, Republic of Korea

Auxiliary Submitter:

Jihoon Cho
Mingyu Cho
Jincheol Ha
Jihoon Kwon
Byeonghak Lee
Joohee Lee
Jooyoung Lee
Sangyub Lee
Dukjae Moon
Mincheol Son
Hyojin Yoon

Inventors:

Jincheol Ha
Seongkwang Kim
Jihoon Kwon
Byeonghak Lee
Joohee Lee
Jooyoung Lee
Sangyub Lee
Dukjae Moon
Mincheol Son

Developers/Owners: All listed submitters

Homepage: https://www.aimer-signature.org

Alternative Point of Contact:

Jooyoung Lee
KAIST
hicalf@kaist.ac.kr
+82-10-8757-7831
291, Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea

Thursday 1st June, 2023

Seongkwang Kim (Signature)

Contents

1 Introduction

AIMer is a signature scheme which is obtained from a zero-knowledge proof of preimage knowledge for a certain one-way function. AIMer consists of two parts: a non-interactive zero-knowledge proof of knowledge (NIZKPoK) system, and a one-way function. The security of both parts solely depends on the security of the underlying symmetric primitives.

The NIZKPoK system in AIMer can be viewed as a customized version of the BN++ proof system [KZ22]. BN++ is a NIZKPoK system based on the MPC-in-the-Head (MPCitH) paradigm [IKOS07], which efficiently proves large-field arithmetic. The difference between our system and BN++ is as follows.

- Our system integrates Commit and ExpandTape to CommitAndExpand. It reduces significant amount of signing and verification time without loss of security in the random oracle model.
- Hash functions and extendable-output functions used in our system are domain-separated for stronger concrete security.

The one-way function of AIMer is AIM [KHS+22], which is a tweakable one-way function dedicated to the BN++ system. AIM has been designed to have strong security against algebraic attacks producing short signatures when combined with BN++. The AIM function fully exploits the optimization techniques of BN++ using *repeated multipliers* for checking multiplication triples and *locally computed output shares* to reduce the overall signature size.

1.1 Overview of the Algorithm

The AIMer signature algorithm consists of key generation, signing, and verification algorithms. To provide an intuitive understanding of the AIMer signature scheme, we will briefly describe the three algorithms below. The detailed specification for mathematical understanding and implementation is given in Section 4 and Section 7, respectively.

KEY GENERATION. The key generation is simply a computation of AIM, which proceeds as follows.

1. A tweak iv and a plaintext pt are sampled uniformly at random.
2. $\mathsf{ct} = \mathsf{AIM}(\mathsf{iv}, \mathsf{pt})$ is computed.
3. The secret key is set to $sk = \mathsf{pt}$, and the corresponding public key is defined as $pk = (\mathsf{iv}, \mathsf{ct})$.

SIGNING ALGORITHM. The signing algorithm is a virtual MPC simulation of AIM. The multiple parties involved in the MPC evaluation are not real participants, but a simulation by the signer (MPCitH). As both signing and verification algorithms are non-interactive, random challenges are computed by hash functions (via the Fiat-Shamir transform). The signing algorithm proceeds as follows.

1. The signer prepares the MPC simulation; it generates seeds for each party, and shares of the input and intermediate values appearing in the computation of AIM from each seed. The signer commits each seed.
2. The signer computes a multiplication-checking protocol from a challenge.
3. The signer opens all the views except one determined by another challenge.

VERIFICATION ALGORITHM. The verification algorithm is a recomputation of the signing algorithm to check whether the MPC simulation has been faithfully executed or not. The verification algorithm mainly checks two steps: preparation of the MPC simulation, and the multiplication-checking protocol. The verification algorithm proceeds as follows.

1. The verifier recomputes shares of all the parties except the unopened one, and computes the first challenge.
2. The verifier recomputes the multiplication-checking protocol, and computes the second challenge.
3. The verifier checks whether the opened views of the MPC simulation are consistent or not.

1.2 Notation

Unless stated otherwise, all logarithms are to the base 2. For two vectors a and b over a finite field, their concatenation is denoted by $a\|b$. For a positive integer n, $\mathsf{hw}(n)$ denotes the Hamming weight of n in its binary representation, and we write $[n] = \{1, \cdots, n\}$. We will write $a \leftarrow b$ to denote assignment of b to a. For a set S, $a \rightarrow S$ denotes that a is added to S as an element and $a \xleftarrow{\$} S$ denotes that a is chosen uniformly at random from S.

In this document, additions are usually operated on a binary field, in which case additions are exclusive-OR (XOR). Nevertheless, when we want to emphasize that an addition is actually XOR, we denote the addition by $\oplus$. In the multiparty computation setting, $x^{(i)}$ denotes the i-th party's additive share of x, which implies that $\sum_i x^{(i)} = x$. We summarize some notations of parameters and non-conventional notations in Table 1.

In Section 7, we follow some notations from programming languages. For a vector (array) `vec`, the notation $\mathtt{vec}[n]$ is used in two different meanings according to its context:

- declaration of a length-n array `vec`,
- the n-th element of the array `vec`.

For a vector `vec`, $\mathtt{vec}[\mathtt{a} : \mathtt{b}]$ denotes a sub-vector of $\mathtt{b} - \mathtt{a} + 1$ elements $(\mathtt{vec}[\mathtt{a}], \ldots, \mathtt{vec}[\mathtt{b}])$. Similarly for a matrix `mat`, $\mathtt{mat}[\mathtt{a} : \mathtt{b}][\mathtt{i}]$ (resp. $\mathtt{mat}[\mathtt{i}][\mathtt{a} : \mathtt{b}]$) denotes a vector of $\mathtt{b} - \mathtt{a} + 1$ elements $\mathtt{mat}[\mathtt{a}][\mathtt{i}], \ldots, \mathtt{mat}[\mathtt{b}][\mathtt{i}]$ (resp. $\mathtt{mat}[\mathtt{i}][\mathtt{a}], \ldots, \mathtt{mat}[\mathtt{i}][\mathtt{b}]$). For a bitstring `str`, we write $\mathtt{str}_{[a:b]}$ to denote a substring from bit-position a to b (both-inclusive). For an integer a and b, we denote $a \ll b$ (resp. $a \gg b$) the left (resp. right) shift of a by b bits.

λ	Security parameter
n	S-box size of AIM
ℓ	Number of S-boxes in front of the linear layer
τ	Number of the parallel repetitions
N	Number of the parties
H	Hash function
XOF	Extendable-output function

Table 1: The notation used in the document.

2 Background

2.1 MPC-in-the-Head Paradigm

The MPC-in-the-Head (MPCitH) paradigm, proposed by Ishai et al. [IKOS07], allows one to construct a zero-knowledge proof (ZKP) system from a multi-party computation (MPC) protocol. Consider an MPC protocol where N parties collaborate to securely evaluate a function f on an input x with perfect correctness. Suppose that the views of k parties leak no information on x. Then, one can build a ZKP from the MPC protocol as follows.

1. The prover generates random secret shares $x^{(1)}, \ldots, x^{(N)}$ such that $x^{(1)} + \cdots + x^{(N)} = x$, and assign them to N parties, say $\mathcal{P}_1, \ldots, \mathcal{P}_N$.
2. The prover simulates the MPC protocol "in her head" by simulating each $\mathcal{P}_i$, $i = 1, \ldots, N$.
3. The prover commits to each party's view which includes its random tape, the secret input share, and the communicated messages from and to the party. She sends the commitments to the verifier.
4. The prover possibly gets random challenges for MPC simulation from the verifier when needed, and conducts local computations on each party. She may repeat this step for several times.
5. The prover completes the MPC simulation and hands over requested output shares of the MPC protocol to the verifier.

Note that the verifier interactively joins the above procedure to provide random challenges to the prover. After that, the verifier selects k parties and asks the prover to open their views. Once the views are received, the verifier checks

1. if the opened views are consistent, i.e., the messages sent from and to a party match and the commitments are correctly evaluated from the resulting views, and
2. if the output recovered from the output share is y.

Since only k views are opened, no information on x is leaked from the revealed views. Also, since the verifier opens the random views, any cheating adversary's winning probability is upper bounded by $(N-k)/N$. We fix $k = N-1$ throughout this proposal.

The practicality of MPCitH is demonstrated by the ZKBoo scheme, the first efficient MPCitH-based proof scheme proposed by Giacomelli et al. [GMO16]. One of the main applications of the MPCitH paradigm is to construct a post-quantum signature. Picnic [CDG+17] is the first and the most famous signature scheme based on the MPCitH paradigm; it combines an MPC-friendly block cipher LowMC [ARS+15] and an MPCitH proof system called ZKB++, which is an optimized variant of ZKBoo. Katz et al. [KKW18] proposed a new proof system KKW by further improving the efficiency of ZKB++ with pre-processing, and updated Picnic accordingly. The updated version of Picnic was the only MPCitH-based scheme that advanced to the third round of the NIST PQC competition. BBQ [dSGMOS19] and Banquet [BSGK+21] are AES-based signature schemes, where BBQ employs the KKW proof system and Banquet improves BBQ by injecting shares for intermediate states.

To fully exploit efficient multiplication over a large field in the Banquet proof system, Dobraunig et al. [DKR+22] proposed MPCitH-friendly ciphers LS-AES and Rain. They are substitution-permutation ciphers based on the inverse S-box over a large field. This design strategy increases the efficiency of the resulting MPCitH-based signature scheme, while the number of rounds should be carefully determined by comprehensive analysis on any possible algebraic attack due to their simple algebraic structures. Kales and Zaverucha [KZ22] proposed several optimization techniques to further improve the efficiency of the Baum and Nof's proof system [BN20], and their variant is called BN++.

2.2 BN++ Proof System

In this section, we briefly review the BN++ proof system [KZ22], one of the state-of-the-art MPCitH zero-knowledge protocols. The BN++ protocol will be combined with our symmetric primitive AIM to construct the AIMer signature scheme. At a high level, BN++ is a variant of the BN protocol [BN20] with several optimization techniques applied to reduce the signature size.

PROTOCOL OVERVIEW. The BN++ protocol follows the MPCitH paradigm [IKOS07]. In order to check C multiplication triples $(x_j, y_j, z_j = x_j \cdot y_j)_{j=1}^{C}$ over a finite field $\mathbb{F}$ in the multiparty computation setting with N parties, *helping triples* $((a_j, b_j)_{j=1}^{C}, c)$ are required, where $a_j \in \mathbb{F}, b_j = y_j$, and $c = \sum_{j=1}^{C} a_j \cdot b_j$. Each party holds secret shares of the multiplication triples $(x_j, y_j, z_j)_{j=1}^{C}$ and the helping triples $((a_j, b_j)_{j=1}^{C}, c)$. Then the protocol proceeds as follows.

- A prover is given random challenges $\epsilon_1, \cdots, \epsilon_C \in \mathbb{F}$.
- For $i \in [N]$, the i-th party locally sets $\alpha_1^{(i)}, \cdots, \alpha_C^{(i)}$ where $\alpha_j^{(i)} = \epsilon_j \cdot x_j^{(i)} + a_j^{(i)}$.
- The parties open $\alpha_1, \cdots, \alpha_C$ by broadcasting their shares.
- For $i \in [N]$, the i-th party locally sets

$$v^{(i)} = \sum_{j=1}^{C} \epsilon_j \cdot z_j^{(i)} - \sum_{j=1}^{C} \alpha_j \cdot b_j^{(i)} + c^{(i)}.$$

- The parties open v by broadcasting their shares and output `Accept` if $v = 0$.

The probability that there exist incorrect triples and the parties output `Accept` in a single run of the above steps is upper bounded by $1/|\mathbb{F}|$.

SIGNATURE SIZE. By applying the Fiat-Shamir transform [DFM20], one can obtain a signature scheme from the BN++ proof system. In this signature scheme, the signature size is given as

$$6\lambda + \tau \cdot (3\lambda + \lambda \cdot \lceil \log_2(N) \rceil + \mathcal{M}(C)),$$

where λ is the security parameter, τ is the number of parallel repetitions of the multiplication checking protocol for reducing the soundness error, C is the number of multiplication gates in the underlying symmetric primitive, and $\mathcal{M}(C) = (2C+1) \cdot \log_2(|\mathbb{F}|)$. In particular, $\mathcal{M}(C)$ has been defined so from the observation that sharing the secret share offsets for $(z_j)_{j=1}^{C}$ and c, and opening shares for $(\alpha_j)_{j=1}^{C}$ occurs for each repetition, using C, 1, and C elements of $\mathbb{F}$, respectively. For more details, we refer to [KZ22].

OPTIMIZATION TECHNIQUES. If multiplication triples use an identical multiplier in common, for example, given (x_1, y, z_1) and (x_2, y, z_2), then the corresponding α values can be batched to reduce the signature size. Instead of computing $\alpha_1 = \epsilon_1 \cdot x_1 + a_1$ and $\alpha_2 = \epsilon_2 \cdot x_2 + a_2$, $\alpha = \epsilon_1 \cdot x_1 + \epsilon_2 \cdot x_2 + a$ is computed, and v is defined as

$$v = \epsilon_1 \cdot z_1 + \epsilon_2 \cdot z_2 - \alpha \cdot y + c,$$

where $c = a \cdot y$. This technique is called *repeated multiplier* technique. Our symmetric primitive design allows us to take full advantage of this technique to reduce the number of α values in each repetition of the protocol.

If the output of the multiplication z_i can be locally generated from each share, then the secret share offset is not necessarily included in the signature.

2.3 Fiat-Shamir Transform

The Fiat-Shamir transform [FS87] is a technique for taking an interactive proof of knowledge and creating a non-interactive counterpart, or a digital signature based on it. The core of the technique is to replace challenges from the verifier by random oracle access which is realized by hashing of the transcript obtained so far.

The Fiat-Shamir transform was originally targeted at a Σ-protocol, a three-round interactive proof of knowledge. Let R be a relation such that, for a given x, it is difficult to find an w such that $R(x, w) = 1$. Given public R and x, the value w such that $R(x, w) = 1$ becomes the secret information that a prover P wants to prove the knowledge of to the verifier V. Then, a Σ-protocol proceeds as follows.

1. **Commitment**: a random number r is generated, committed to by the prover, and sent to the verifier.

   $\mathsf{P} \xrightarrow{\text{com}} \mathsf{V}$, where $\text{com} = \mathsf{Commit}(r)$.

2. **Challenge**: on receiving the commitment, the verifier sends a random challenge ch the prover.

   $\mathsf{P} \xleftarrow{\text{ch}} \mathsf{V}$.

3. **Response**: the prover creates an appropriate response corresponding to the challenge.

$$\mathsf{P} \xrightarrow{\text{res}} \mathsf{V}, \quad \text{where res} = \mathsf{Response}(w, r, \text{ch}).$$

Then, the verifier checks the validity of the response together with com and ch. This Σ-protocol is transformed into a non-interactive version, by replacing the challenge sent by the verifier by a random oracle access, using the previous transcript (x, com). Denoting the random oracle as $\mathcal{RO}$, the challenge step of the above procedure is replaced by ch $\leftarrow \mathcal{RO}(x, \text{com})$. This approach can be extended to multi-round proofs. The security loss is known to be linear in the number of attacker's queries to the random oracle [AFK22].

3 Symmetric Primitive AIM

3.1 Specification

AIM is designed to be a "tweakable" one-way function so that it offers multi-target one-wayness. Given input/output size n and an $(\ell + 1)$-tuple of exponents $(e_1, \ldots, e_\ell, e_*) \in \mathbb{Z}^{\ell+1}$, $\mathsf{AIM} : \{0,1\}^n \times \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ is defined by

$$\mathsf{AIM}(\mathsf{iv}, \mathsf{pt}) = \mathsf{Mer}[e_*] \circ \mathsf{Lin}[\mathsf{iv}] \circ \mathsf{Mer}[e_1, \ldots, e_\ell](\mathsf{pt}) \oplus \mathsf{pt}$$

where each function will be described below. See Figure 1 for the pictorial description of AIM with $\ell = 3$.

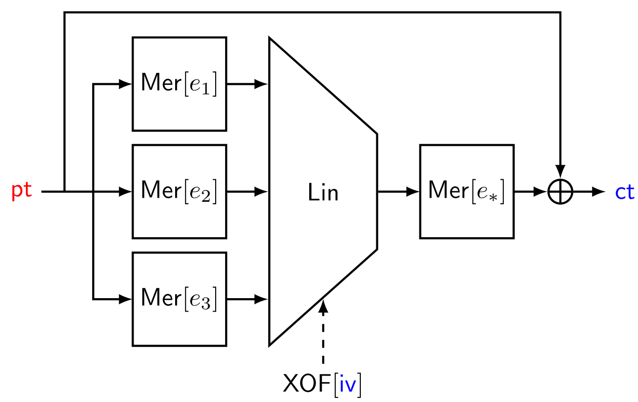

Figure 1: The AIM-V one-way function with $\ell = 3$. The input pt (in red) is the secret key of the signature scheme, and $(\mathsf{iv}, \mathsf{ct})$ (in blue) is the corresponding public key.

S-BOXES. In AIM, S-boxes are exponentiation by Mersenne numbers over a large field. More precisely, for $x \in \mathbb{F}_{2^n}$,

$$\mathsf{Mer}[e](x) = x^{2^e - 1}$$

for some e. Note that this map is a permutation if $\gcd(e, n) = 1$. As an extension, $\mathsf{Mer}[e_1, \ldots, e_\ell] : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}^\ell$ is defined by

$$\mathsf{Mer}[e_1, \ldots, e_\ell](x) = \mathsf{Mer}[e_1](x) \| \ldots \| \mathsf{Mer}[e_\ell](x).$$

LINEAR COMPONENTS. AIM includes two types of linear components: an affine layer and feed-forward. The affine layer is multiplication by an $n \times \ell n$ random binary matrix A_{iv} and addition by a random constant $b_{\mathsf{iv}} \in \mathbb{F}_2^n$. The matrix

$$A_{\mathsf{iv}} = \left[A_{\mathsf{iv},1} \middle| \ldots \middle| A_{\mathsf{iv},\ell} \right] \in (\mathbb{F}_2^{n \times n})^\ell$$

is composed of ℓ random invertible matrices $A_{\mathsf{iv},i}$. The matrix A_{iv} and the vector b_{iv} are generated by an extendable-output function (XOF) with the initial vector iv. Each matrix $A_{\mathsf{iv},i}$ can be equivalently represented by a linearized polynomial $L_{\mathsf{iv},i}$ on $\mathbb{F}_{2^n}$. For $x = (x_1, \ldots, x_\ell) \in (\mathbb{F}_{2^n})^\ell$,

$$\mathsf{Lin}[\mathsf{iv}](x) = \sum_{1 \le i \le \ell} L_{\mathsf{iv},i}(x_i) \oplus b_{\mathsf{iv}}.$$

By abuse of notation, we will denote Ax as the same meaning as $\sum_{1 \le i \le \ell} L_{\mathsf{iv},i}(x_i)$. Feed-forward operation, which is addition by the input itself, makes the entire function non-invertible.

RECOMMENDED PARAMETERS. Recommended sets of parameters for $\lambda \in \{128, 192, 256\}$-bit security are given in Table 2. The number of S-boxes is determined by taking into account the complexity of the XL attack, which is described in Section 6.3.2. Exponents e_1 and e_* are chosen as small numbers to provide smaller differential probability, and exponent e_2 is chosen so that $e_2 \cdot e_* \ge \lambda$, while all the exponents are distinct in the same set of parameters. The irreducible polynomials for extension fields $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$ are the same as those used in Rain [DKR+22].

Scheme	λ	n	ℓ	e_1	e_2	e_3	e_*
AIM-I	128	128	2	3	27	-	5
AIM-III	192	192	2	5	29	-	7
AIM-V	256	256	3	3	53	7	5

Table 2: Recommended sets of parameters of AIM.

3.2 Design Rationale

CHOICE OF FIELD. When a symmetric primitive is built upon field operations, the field is typically binary since bitwise operations are cheap in most of modern architectures. However, when the multiplicative complexity of the primitive becomes a more important metric for efficiency, it is hard to generally specify which type of field has merits with respect to security and efficiency.

Focusing on a primitive for MPCitH-style zero-knowledge protocols, a primitive over a large field generally requires a small number of multiplications, leading to shorter signatures. However, any primitive operating on a large field of large prime characteristic might permit algebraic attacks since the number of variables and the algebraic degree will be significantly limited for efficiency reasons. On the other hand, binary extension fields enjoy both advantages from small and large fields. In particular, matrix multiplication is represented by a polynomial of high algebraic degree without increasing the proof size.

ALGEBRAICALLY SOUND S-BOXES. In an MPCitH-style zero-knowledge protocol, the proof size of a circuit is usually proportional to the number of nonlinear operations in the circuit. In order to minimize the number of multiplications, one might introduce intermediate variables for some wires of the circuit. For example, the inverse S-box ($S(x) = x^{-1}$) has high (bitwise) algebraic degree $n - 1$, while it can be simply represented by a quadratic equation $xy = 1$ by letting the output from the S-box be a new variable y. When an S-box is represented by a quadratic equation of its input and output, we will say it is *implicitly quadratic*. In particular, we consider implicitly quadratic S-boxes which are represented by a single multiplication over $\mathbb{F}_{2^n}$. This feature makes the proof size short and mitigates algebraic attacks at the same time.

The inverse S-box is one of the well-studied implicitly quadratic S-boxes. The inverse S-box has been widely adopted to symmetric ciphers [DR02, AIK+01, SSA+07] due to its nice cryptographic properties. It is invertible, is of high-degree, has good enough differential uniformity and nonlinearity. Recently, it is used in symmetric primitives for advanced cryptographic protocols such as multiparty computation and zero-knowledge proof [GKR+21, GLR+20, DKR+22].

Meanwhile, the inverse S-box has one minor weakness; a single evaluation of the n-bit inverse S-box as a form of $xy = 1$ produces $5n - 1$ linearly independent quadratic equations over $\mathbb{F}_2$ [CDG06]. The complexity of an algebraic attack is typically bounded (with heuristics) by the degree and the number of equations, and the number of variables. In particular, an algebraic attack is more efficient with a larger number of equations, while this aspect has not been fully considered in the design of recent symmetric ciphers based on algebraic S-boxes. When the number of rounds is small, this issue might be critical to the overall security of the cipher. For more details, see Section 6.3.2.

With the above observation, we tried to find an invertible S-box of high-degree which is moderately resistant to differential/linear cryptanalysis as well as implicitly quadratic, and *producing only a small number of quadratic equations*. Since our attack model does not allow multiple queries to a single instance of AIM, we allow a relaxed condition on the DC/LC resistance, not being necessarily maximal. As a family of S-boxes that beautifully fit all the conditions, we choose a family of Mersenne S-boxes; they are exponentiation by Mersenne numbers $2^e - 1$ such that $\gcd(n, e) = 1$, are invertible, are of high-degree, need only one multiplication for its proof, produce only $3n$ Boolean quadratic equations with its input and output, and provide moderate DC/LC resistance. Furthermore, when the implicit equation $xy = x^{2^e}$ of a Mersenne S-box is computed in the BN++ proof system, it is not required to broadcast the output share since the output of multiplication x^{2^e} can be locally computed from the share of x.

REPETITIVE STRUCTURE. The efficiency of the BN++ proof system partially comes from the optimization technique using *repeated multipliers*. When a multiplier is repeated in multiple equations to prove, the proof can be done in a batched way, reducing the overall signature size. In order to maximize the advantage of repeated multipliers, we put S-boxes in parallel at the first round in order to make the S-box inputs the same. Then, we put only one S-box at the second round with feed-forward. In this way, all the implicit equations have the same multiplier.

AFFINE LAYER GENERATION. The main advantage of using binary affine layers in large S-box-based constructions is to increase the algebraic degree of equations over the large field. Multiplication by a random $n \times n$ binary matrix can be represented as

$$\sum_{i=0}^{n-1} a_i x^{2^i} = a_0 x + a_1 x^{2^1} + a_2 x^{2^2} + \cdots + a_{n-1} x^{2^{n-1}}$$

where $a_0, a_1, \ldots, a_{n-1} \in \mathbb{F}_{2^n}$. Similarly, our design uses a random affine map from $\mathbb{F}_2^{\ell n}$ to $\mathbb{F}_2^n$. In order to mitigate multi-target attacks (in the multi-user setting), the affine map is uniquely generated for each user; each user's iv is fed to an XOF, generating the corresponding linear layer.

4 Mathematical Description of the AIMer Signature Scheme

In this section, we review the mathematical specification of AIMer introduced in [KHS+22]. In order to obtain the AIMer signature scheme, the customized BN++ proof system in Section 2.2 is combined with AIM. The resulting signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ consists of key generation, signing, and verification algorithms.

- $\mathsf{KeyGen}(1^\lambda) \to (sk, pk)$: Sample uniform random $\mathsf{pt} \xleftarrow{\$} \mathbb{F}_{2^n}$, and $\mathsf{iv} \xleftarrow{\$} \{0,1\}^n$. Compute $\mathsf{ct} \leftarrow \mathsf{AIM}(\mathsf{iv}, \mathsf{pt})$ as described in Section 3, and set the public key $pk \leftarrow (\mathsf{iv}, \mathsf{ct}) \in \{0,1\}^n \times \mathbb{F}_{2^n}$ and the private signing key $sk \leftarrow \mathsf{pt} \in \mathbb{F}_{2^n}$.
- $\mathsf{Sign}((sk, pk), m) \to \sigma$: Take as input a pair of signing and public keys ($sk = \mathsf{pt}, pk = (\mathsf{iv}, \mathsf{ct})$) and a message m, and compute the BN++ ZKP π for the AIM one-way function circuit using m as a part of the input to the challenge hash as described in Algorithm 1. Output the corresponding signature $\sigma \leftarrow \pi$.

- $\mathsf{Verify}(pk, \sigma) \to \mathsf{Accept}$ or Reject : Take as input a public key $pk = (\mathsf{iv}, \mathsf{ct})$, a message m and a signature σ and conduct the verification of BN++ ZKP for the AIM one-way function circuit as described in Algorithm 2. Output either Accept or Reject according to the verification result of the ZKP.

The AIMer signing and verification algorithms will be described in detail in the following subsections. The full specification for implementation is given in Section 7.

4.1 Features

The AIM function has been designed to fully exploit the optimization techniques of the BN++ proof system using *repeated multipliers* for checking multiplication triples and *locally computed output shares* to reduce the overall signature size.

EXPLOITING REPEATED MULTIPLIERS. If multiplication triples share the same multiplier, then the α values in the multiplication checking protocol can be batched as mentioned in Section 2.2. The $\ell+1$ S-box evaluations in AIM produce the $\ell + 1$ multiplication triples that need to be verified, reformulated as follows.

$$\mathsf{pt} \cdot t_i = \mathsf{pt}^{2^{e_i}}$$

for $i = 1, \ldots, \ell$, and

$$\mathsf{pt} \cdot \mathsf{Lin}[\mathsf{iv}](t) = (\mathsf{Lin}[\mathsf{iv}](t))^{2^{e_*}} + \mathsf{ct} \cdot \mathsf{Lin}[\mathsf{iv}](t)$$

where t_i, $i = 1, 2, \ldots, \ell$, is the output of the i-th S-box and $t \overset{\text{def}}{=} [t_1|\ldots|t_\ell]$. Since every multiplication triple shares the same multiplier pt, a single value of α is included in the signature instead of $\ell+1$ different values.

LOCALLY COMPUTED OUTPUT SHARES. For the above multiplication triples, every multiplication output share on the right-hand side can be locally computed without communication between parties, thanks to the freshman's dream over $\mathbb{F}_{2^n}$ (*i.e.*, the map $x \mapsto x^{2^e}$ is linear over $\mathbb{F}_{2^n}$). Hence, it is possible to remove the offset Δz of the output share in the multiplication triples in the BN++ proof from the signature of AIMer. For the first ℓ multiplications, each party locally computes the output $(\mathsf{pt}^{(i)})^{2^{e_i}}$ from their input share $\mathsf{pt}^{(i)}$ using linear operations. For the last multiplication output, the output is determined as follows.

$$\begin{cases} (A_{\mathsf{iv}} \cdot t^{(i)} + b_{\mathsf{iv}})^{2^{e_*}} + \mathsf{ct} \cdot (A_{\mathsf{iv}} \cdot t^{(i)} + b_{\mathsf{iv}}) & \text{for } i = 1, \\ (A_{\mathsf{iv}} \cdot t^{(i)})^{2^{e_*}} + \mathsf{ct} \cdot (A_{\mathsf{iv}} \cdot t^{(i)}) & \text{for } i \geq 2, \end{cases}$$

where $t^{(i)} \in \mathbb{F}_2^{\ell n}$ is the output shares of the first ℓ S-boxes for the i-th party: $t^{(i)} = [t_1^{(i)}|\ldots|t_\ell^{(i)}]$.

With the above optimization techniques applied, the signature size is

$$6\lambda + \tau \cdot (\lambda \cdot \lceil \log_2(N) \rceil + 2\lambda + (\ell + 3) \cdot n).$$

Since $n = \lambda$ in our recommended sets of parameters, it can be represented as

$$6\lambda + \tau \cdot (\lambda \cdot \lceil \log_2(N) \rceil + (\ell + 5) \cdot \lambda).$$

OTHER SYMMETRIC PRIMITIVES IN USE. The SHAKE128 (resp. SHAKE256) XOF is used to instantiate hash functions $\mathsf{CommitAndExpand}$, H_1, H_2 and a pseudorandom generator Expand in the signature scheme for $\lambda = 128$ (resp. $\lambda \in \{192, 256\}$). $\mathsf{Sample}(\mathsf{tape})$ samples an element from a random tape tape, which is a part of the output of $\mathsf{CommitAndExpand}$, tracking the current position of the tape.

4.2 Signature Generation

In this section, we review the signing algorithm of AIMer. The signing algorithm consists of five phases as commented in Algorithm 1.

Phase 1: Committing to the seeds and the execution views of the parties. It samples a random salt salt, and computes an instance of AIM using the initial vector iv. After that, for each parallel execution $k \in [\tau]$, it does the following.

1. It samples a root seed seed_k for the k-th execution, and computes the parties' seeds $\mathsf{seed}_k^{(1)}, \cdots, \mathsf{seed}_k^{(N)}$ as leaves of a binary tree from seed_k.
2. It commit to each party's seed and expand random tape as
$$(\mathsf{com}_k^{(i)}, \mathsf{tape}_k^{(i)}) \leftarrow \mathsf{CommitAndExpand}(\mathsf{salt}, k, i, \mathsf{seed}_k^{(i)})$$
for $i \in [N]$.
3. It prepares for the multi-party computation among the N parties using the parties' seeds, by generating secret shares $\left(x_{k,j}^{(i)}, \mathsf{pt}_k^{(i)}, z_{k,j}^{(i)}\right)$ of the multiplication triples for each S-box with index j, where $z_{k,j}^{(i)} = (\mathsf{pt}_k^{(i)})^{2^{e_j}}$ and $x_{k,j}^{(i)}$ is the secret share of the injected output of the j-th S-box of the secret key pt for $j \in \{1, \cdots, \ell\}$. Also, for the $(\ell+1)$-th S-box, it formulates the multiplication triple as $\left(x_{k,\ell+1}^{(i)}, \mathsf{pt}_k^{(i)}, z_{k,\ell+1}^{(i)}\right)$ where $z_{k,\ell+1}^{(i)} = (x_{k,\ell+1}^{(i)})^{2^{e_*}} + \mathsf{ct} \cdot x_{k,\ell+1}^{(i)}$, and $x_{k,\ell+1}^{(i)} = A_{\mathsf{iv}} \cdot [t_{k,1}^{(i)} | \cdots | t_{k,\ell}^{(i)}] + b_{\mathsf{iv}}$ for $i = 1$ and $x_{k,\ell+1}^{(i)} = A_{\mathsf{iv}} \cdot [t_{k,1}^{(i)} | \cdots | t_{k,\ell}^{(i)}]$ otherwise. It samples helping triples $\left(a_k^{(i)}, b_k^{(i)}, c_k^{(i)}\right)$ as in BN++ and computes the linear operations over the secret shares.[1]

Since we use the additive secret sharing, the witness share and the secret share of the multiplication triples and the helping triples can be recomputed from the offsets $\Delta\mathsf{pt}_k$ and $(\Delta c_k, (\Delta t_{k,j})_{j\in[\ell]})$, respectively, combined with the parties' random seeds. It outputs $\sigma_1 \leftarrow \left(\mathsf{salt}, ((\mathsf{com}_k^{(i)})_{i\in[N]}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]})_{k\in[\tau]}\right)$.

Phase 2: Challenging the checking protocol. It then computes $h_1 \leftarrow H_1(m, \mathsf{iv}, \mathsf{ct}, \sigma_1)$ and outputs $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_1)$ where $\epsilon_{k,j} \in \mathbb{F}_{2^n}$ is a challenge value for the multiplication checking protocol in BN++.

Phase 3: Committing to the simulation of the checking protocol. It computes the broadcast values $\alpha_k^{(i)}, v_k^{(i)}$ for the multiplication checking protocol of BN++. It outputs $\sigma_2 \leftarrow \left(\mathsf{salt}, ((\alpha_k^{(i)}, v_k^{(i)})_{i\in[N]})_{k\in[\tau]}\right)$.

Phase 4: Challenging the views of the MPC protocol. It computes $h_2 \leftarrow H_2(h_1, \sigma_2)$, and outputs a challenge index $\bar{i}_k \in [N]$ for an unopened view by computing $(\bar{i}_k)_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_2)$.

Phase 5: Opening the views of the MPC and checking protocols. It collects the seeds to open the views of $N-1$ parties as $\mathsf{seeds}_k \leftarrow \{\lceil \log_2(N) \rceil$ (tree) nodes to compute $\mathsf{seed}_k^{(i)}$ for $i \in [N] \backslash \{\bar{i}_k\}\}$ for each repetition k. It then outputs a signature $\sigma \leftarrow (\mathsf{salt}, h_1, h_2, (\mathsf{seeds}_k, \mathsf{com}_k^{(\bar{i}_k)}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}, \alpha_k^{(\bar{i}_k)})_{k\in[\tau]})$.

4.3 Signature Verification

In this section we review the verification algorithm of AIMer. The verification algorithm takes as input $((\mathsf{iv}, \mathsf{ct}), m, \sigma)$, and outputs Accept or Reject. We refer to Algorithm 2 for the detailed description.

First, given an input iv, it computes an instance of AIM, i.e., computes a binary matrix A_{iv} and a vector b_{iv}. From the signature parsed as $\sigma = \left(\mathsf{salt}, h_1, h_2, \left(\mathsf{seeds}_k, \mathsf{com}_k^{(\bar{i}_k)}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}, \alpha_k^{(\bar{i}_k)}\right)_{k\in[\tau]}\right)$, it expands hash values h_1 and h_2 to obtain the challenges $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]}$ in Phase 2 and $(\bar{i}_k)_{k\in[\tau]}$ in Phase 4 of the signing algorithm.

[1] We note that, by exploiting repeated multipliers, we need to verify only one multiplication for each repetition.

Algorithm 1: $\mathsf{Sign}((\mathsf{pt}, (\mathsf{iv}, \mathsf{ct})), m)$ - AIMer signature scheme, signing algorithm.

`// Phase 1: Committing to the seeds and the execution views of the parties.`

Sample a random salt $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$.

Compute the first ℓ S-boxes' outputs $t_1, \ldots, t_\ell$.

Derive the binary matrix $A_{\mathsf{iv}} \in (\mathbb{F}_2^{n\times n})^\ell$ and the vector $b_{\mathsf{iv}} \in \mathbb{F}_2^n$ from the initial vector iv.

for *each parallel execution* $k \in [\tau]$ **do**

  Sample a root seed : $\mathsf{seed}_k \xleftarrow{\$} \{0,1\}^\lambda$.

  Compute parties' seeds $\mathsf{seed}_k^{(1)}, \ldots, \mathsf{seed}_k^{(N)}$ as leaves of binary tree from seed_k.

  for *each party* $i \in [N]$ **do**

    Commit to the seed and expand random tape:
    $(\mathsf{com}_k^{(i)}, \mathsf{tape}_k^{(i)}) \leftarrow \mathsf{CommitAndExpand}(\mathsf{salt}, k, i, \mathsf{seed}_k^{(i)})$.

    Sample witness share: $\mathsf{pt}_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

  Compute witness offset and adjust first witness: $\Delta\mathsf{pt}_k \leftarrow \mathsf{pt} - \sum_i \mathsf{pt}_k^{(i)}$, $\mathsf{pt}_k^{(1)} \leftarrow \mathsf{pt}_k^{(1)} + \Delta\mathsf{pt}_k$.

  for *each S-box with index* j **do**

    if $j \leq \ell$ **then**

      For each party i, sample an S-box output: $t_{k,j}^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

      Compute output offset and adjust first share: $\Delta t_{k,j} = t_j - \sum_i t_{k,j}^{(i)}$, $t_{k,j}^{(1)} \leftarrow t_{k,j}^{(1)} + \Delta t_{k,j}$.

      For each party i, set $x_{k,j}^{(i)} = t_{k,j}^{(i)}$ and $z_{k,j}^{(i)} = (\mathsf{pt}_k^{(i)})^{2^{e_j}}$.

    if $j = \ell + 1$ **then**

      For $i = 1$, set $x_{k,j}^{(i)} = A_{\mathsf{iv}} \cdot t_{k,*}^{(i)} + b_{\mathsf{iv}}$ where $t_{k,*}^{(i)} = [t_{k,1}^{(i)} | \ldots | t_{k,\ell}^{(i)}]$ is the output shares of the first ℓ S-boxes.

      For each party $i \in [N] \backslash \{1\}$, set $x_{k,j}^{(i)} = A_{\mathsf{iv}} \cdot t_{k,*}^{(i)}$

      For each party i, set $z_{k,j}^{(i)} = (x_{k,j}^{(i)})^{2^{e_*}} + \mathsf{ct} \cdot x_{k,j}^{(i)}$.

  For each party i, set $a_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

  Compute $a_k = \sum_{i=1}^N a_k^{(i)}$.

  Set $c_k = a_k \cdot \mathsf{pt}$.

  For each party i, set $c_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

  Compute offset and adjust first share : $\Delta c_k = c_k - \sum_i c_k^{(i)}$, $c_k^{(1)} \leftarrow c_k^{(1)} + \Delta c_k$.

Set $\sigma_1 \leftarrow (\mathsf{salt}, ((\mathsf{com}_k^{(i)})_{i\in[N]}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]})_{k\in[\tau]})$.

`// Phase 2: Challenging the checking protocol.`

Compute challenge hash: $h_1 \leftarrow H_1(m, \mathsf{iv}, \mathsf{ct}, \sigma_1)$.

Expand hash: $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_1)$ where $\epsilon_{k,j} \in \mathbb{F}_{2^n}$.

`// Phase 3: Committing to the simulation of the checking protocol.`

for *each repetition* k **do**

  Simulate the triple checking protocol as in Section 2.2 for all parties with challenge $\epsilon_{k,j}$. The inputs are $((x_{k,j}^{(i)}, \mathsf{pt}_k^{(i)}, z_{k,j}^{(i)})_{j\in[\ell+1]}, a_k^{(i)}, b_k^{(i)}, c_k^{(i)})$, where $b_k^{(i)} = \mathsf{pt}_k^{(i)}$, and let $\alpha_k^{(i)}$ and $v_k^{(i)}$ be the broadcast values.

Set $\sigma_2 \leftarrow (\mathsf{salt}, ((\alpha_k^{(i)}, v_k^{(i)})_{i\in[N]})_{k\in[\tau]})$.

`// Phase 4: Challenging the views of the MPC protocol.`

Compute challenge hash: $h_2 \leftarrow H_2(h_1, \sigma_2)$.

Expand hash: $(\bar{i}_k)_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_2)$ where $\bar{i}_k \in [N]$.

`// Phase 5: Opening the views of the MPC and checking protocols.`

for *each repetition* k **do**

  $\mathsf{seeds}_k \leftarrow \{\lceil \log_2(N) \rceil$ nodes to compute $\mathsf{seed}_k^{(i)}$ for $i \in [N] \backslash \{\bar{i}_k\}\}$.

Output $\sigma \leftarrow (\mathsf{salt}, h_1, h_2, (\mathsf{seeds}_k, \mathsf{com}_k^{(\bar{i}_k)}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}, \alpha_k^{(\bar{i}_k)})_{k\in[\tau]})$.

Recomputation of Phase 1 and 2. It does the following for each parallel repetition $k \in [\tau]$:

- Recomputes a random seed for each $i \in [N] \setminus \{\bar{i}_k\}$, and

$$(\mathsf{com}_k^{(i)}, \mathsf{tape}_k^{(i)}) \leftarrow \mathsf{CommitAndExpand}(\mathsf{salt}, k, i, \mathsf{seed}_k^{(i)}).$$

- Recompute all the secret shares for the multiplication triples from the random seed for each $i \in [N] \setminus \{\bar{i}_k\}$.

Then, it recomputes $\sigma_1 \leftarrow \left(\mathsf{salt}, \left((\mathsf{com}_k^{(i)})_{i\in[N]}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}\right)_{k\in[\tau]}\right)$, and the challenge hash $h_1' \leftarrow H_1(m, \mathsf{iv}, \mathsf{ct}, \sigma_1)$.

Recomputation of Phase 3 and 4. For each parallel repetition $k \in [\tau]$, it simulates the multiplication checking protocol in Section 2.2 for each $i \in [N] \setminus \{\bar{i}_k\}$. It recomputes the broadcast values $\alpha_k^{(i)}$ and $v_k^{(i)}$ of the BN++ protocol for each $i \in [N] \setminus \{\bar{i}_k\}$. Also, it computes the remaining share of the output value for the $\bar{i}_k$-th party as $v_k^{(\bar{i}_k)} = 0 - \sum_{i \neq \bar{i}_k} v_k^{(i)}$. Finally, it recomputes $\sigma_2 \leftarrow \left(\mathsf{salt}, ((\alpha_k^{(i)}, v_k^{(i)})_{i\in[N]})_{k\in[\tau]}\right)$, and the challenge hash $h_2' = H_2(h_1, \sigma_2)$.

Comparison of the hash values. It compares the hash values in the input signature and those in the recomputation. It outputs Accept only if both $h_1 = h_1'$ and $h_2 = h_2'$ hold, and outputs Reject, otherwise.

4.4 Recommended Parameters

For security levels L1, L3, and L5, recommended sets of parameters are given in Table 3. For each value of security parameter λ, the corresponding sets of parameters are expected to provide λ-bit security against all classical attacks, and $\lambda/2$-bit security against quantum attacks.

5 Formal Security Analysis

5.1 EUF-CMA Security of AIMer in the Random Oracle Model

In this section, we prove the EUF-CMA (existential unforgeability under adaptive chosen-message attacks [GMR88]) security of AIMer. To prove the EUF-CMA security, we first show that AIMer is secure against key-only attack (EUF-KO) in Theorem 1, where an adversary is given the public key and no access to the signing oracle. Then, we show that AIMer is EUF-CMA secure by proving that the signing can be simulated without using the secret key in Theorem 2. In our security proof, we followed the same arguments as the security proof of BN++ in [KZ22].

Theorem 1 (EUF-KO Security of AIMer)**.** *Assume that* CommitAndExpand*, H_1 and H_2 be modeled as random oracles, Expand be modeled as a random function, and let (N, τ, λ) be parameters of the* AIMer *signature scheme. Let $\mathcal{A}$ be an arbitrary adversary against the EUF-KO security of* AIMer *that makes a total of Q random oracle queries. Assuming that KeyGen is an ϵ_{OWF}-hard one-way function, then $\mathcal{A}$'s advantage in the EUF-KO game is*

$$\epsilon_{\mathsf{KO}} \leq \epsilon_{\mathsf{OWF}} + \frac{(\tau N + 1)Q^2}{2^{2\lambda}} + Pr[X + Y = \tau],$$

where $Pr[X + Y = \tau]$ is as described in the proof.

Algorithm 2: $\mathsf{Verify}((\mathsf{iv},\mathsf{ct}), m, \sigma)$ - AIMer signature scheme, verification algorithm.

Parse σ as $\left(\mathsf{salt}, h_1, h_2, \left(\mathsf{seeds}_k, \mathsf{com}_k^{(\bar{i}_k)}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}, \alpha_k^{(\bar{i}_k)}\right)_{k\in[\tau]}\right)$.

Derive the binary matrix $A_{\mathsf{iv}} \in (\mathbb{F}_2^{n\times n})^\ell$ and the vector $b_{\mathsf{iv}} \in \mathbb{F}_2^n$ from the initial vector iv.

Expand hashes: $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_1)$ and $(\bar{i}_k)_{k\in[\tau]} \leftarrow \mathsf{Expand}(h_2)$.

for *each parallel repetition* $k \in [\tau]$ **do**

    Uses seeds_k to recompute $\mathsf{seed}_k^{(i)}$ for $i \in [N] \setminus \{\bar{i}_k\}$.

    for *each party* $i \in [N] \setminus \{\bar{i}_k\}$ **do**

        Recompute $(\mathsf{com}_k^{(i)}, \mathsf{tape}_k^{(i)}) \leftarrow \mathsf{CommitAndExpand}(\mathsf{salt}, k, i, \mathsf{seed}_k^{(i)})$, $\mathsf{pt}_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

        if $i = 1$ **then**

            Adjust first share: $\mathsf{pt}_k^{(i)} \leftarrow \mathsf{pt}_k^{(i)} + \Delta\mathsf{pt}_k$

        for *each S-box with index* j **do**

            if $j \le \ell$ **then**

                Sample an S-box output: $t_{k,j}^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

                if $i = 1$ **then**

                    Adjust first share: $t_{k,j}^{(1)} \leftarrow t_{k,j}^{(1)} + \Delta t_{k,j}$.

                Set $x_{k,j}^{(i)} = t_{k,j}^{(i)}$ and $z_{k,j}^{(i)} = (\mathsf{pt}_k^{(i)})^{2^{e_j}}$.

            if $j = \ell + 1$ **then**

                if $i = 1$ **then**

                    Set $x_{k,j}^{(i)} = A_{\mathsf{iv}} \cdot t_{k,*}^{(i)} + b_{\mathsf{iv}}$ where $t_{k,*}^{(i)} = [t_{k,1}^{(i)}|\dots|t_{k,\ell}^{(i)}]$ is the output shares of the first ℓ S-boxes.

                else

                    Set $x_{k,j}^{(i)} = A_{\mathsf{iv}} \cdot t_{k,*}^{(i)}$.

                Set $z_{k,j}^{(i)} = (x_{k,j}^{(i)})^{2^{e_*}} + \mathsf{ct} \cdot x_{k,j}^{(i)}$.

        Set $a_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$ and $c_k^{(i)} \leftarrow \mathsf{Sample}(\mathsf{tape}_k^{(i)})$.

        if $i = 1$ **then**

            Adjust first share $c_k^{(i)} \leftarrow c_k^{(i)} + \Delta c_k$.

Set $\sigma_1 \leftarrow \left(\mathsf{salt}, \left((\mathsf{com}_k^{(i)})_{i\in[N]}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]}\right)_{k\in[\tau]}\right)$.

Set $h_1' \leftarrow H_1(m, \mathsf{iv}, \mathsf{ct}, \sigma_1)$.

for *each parallel execution* $k \in [\tau]$ **do**

    for *each party* $i \in [N] \setminus \{\bar{i}_k\}$ **do**

        Simulate the triple checking protocol as defined in Section 2.2 for all parties with challenge $\epsilon_{k,j}$. The inputs are $((x_{k,j}^{(i)}, \mathsf{pt}_k^{(i)}, z_{k,j}^{(i)})_{j\in[\ell+1]}, a_k^{(i)}, b_k^{(i)}, c_k^{(i)})$, where $b_k^{(i)} = \mathsf{pt}_k^{(i)}$, and let $\alpha_k^{(i)}$ and $v_k^{(i)}$ be the broadcast values.

    Compute $v_k^{(\bar{i}_k)} = 0 - \sum_{i\neq\bar{i}_k} v_k^{(i)}$.

Set $\sigma_2 \leftarrow \left(\mathsf{salt}, ((\alpha_k^{(i)}, v_k^{(i)})_{i\in[N]})_{k\in[\tau]}\right)$

Set $h_2' = H_2(h_1', \sigma_2)$.

Output Accept if $h_1 = h_1'$ and $h_2 = h_2'$.

Otherwise, output Reject.

Parameters	λ	n	ℓ	e_1	e_2	e_3	e_*	Hash	N	τ
AIMER_L1_PARAM1	128	128	2	3	27	-	5	SHAKE128	16	33
AIMER_L1_PARAM2	128	128	2	3	27	-	5	SHAKE128	57	23
AIMER_L1_PARAM3	128	128	2	3	27	-	5	SHAKE128	256	17
AIMER_L1_PARAM4	128	128	2	3	27	-	5	SHAKE128	1615	13
AIMER_L3_PARAM1	192	192	2	5	29	-	7	SHAKE256	16	49
AIMER_L3_PARAM2	192	192	2	5	29	-	7	SHAKE256	64	33
AIMER_L3_PARAM3	192	192	2	5	29	-	7	SHAKE256	256	25
AIMER_L3_PARAM4	192	192	2	5	29	-	7	SHAKE256	1621	19
AIMER_L5_PARAM1	256	256	3	3	53	7	5	SHAKE256	16	65
AIMER_L5_PARAM2	256	256	3	3	53	7	5	SHAKE256	62	44
AIMER_L5_PARAM3	256	256	3	3	53	7	5	SHAKE256	256	33
AIMER_L5_PARAM4	256	256	3	3	53	7	5	SHAKE256	1623	25

Table 3: The recommended parameters for AIMer.

Proof. We build an algorithm $\mathcal{B}$ that retrieves a pre-image for the AIM one-way function using the EUF-KO adversary $\mathcal{A}$ as a subroutine. Let H_{c} denote a random oracle (RO) modelling CommitAndExpand. Suppose that all the queries to H_{c}, H_1 and H_2 are listed in $\mathcal{Q}_{\mathsf{c}}$, $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively. We extend the output length of random oracles H_1 and H_2 instead of making calls to Expand() in our analysis, since Expand is a random function used to expand outputs from H_1 and H_2.

Algorithm $\mathcal{B}$ takes the AIM one-way function value $(\mathsf{iv}, \mathsf{ct})$ as an input, and forwards it to $\mathcal{A}$ as a AIMer public key for the EUF-KO game. $\mathcal{B}$ manages a set Bad to keep track of all the answers from the three random oracles and two tables $\mathcal{T}_{\mathsf{sh}}$ and $\mathcal{T}_{\mathsf{in}}$ to record the values derived from $\mathcal{A}$'s RO queries as follows:

- $\mathcal{T}_{\mathsf{sh}}$ to store secret shares of the parties, and
- $\mathcal{T}_{\mathsf{in}}$ to store inputs to the MPC protocol.

We also program the random oracles for $\mathcal{A}$ as follows.

- H_{c} : When $\mathcal{A}$ queries random oracle for H_{c}, $\mathcal{B}$ records the query to match the commitments and expanded random tape with its corresponding seeds. See Algorithm 3.
- H_1 : When $\mathcal{A}$ commits to seeds and sends the offsets for the preimage pt which is the secret key and the multiplication triples, $\mathcal{B}$ check the query list $\mathcal{Q}_{\mathsf{c}}$ to see if the commitments were output by its simulation of H_{c}. If $\mathcal{B}$ finds matching results for all i's in some repetition k, then it can recover pt. See Algorithm 4.
- H_2 : See Algorithm 5.

After $\mathcal{A}$ terminates, $\mathcal{B}$ checks whether there is $\mathsf{pt}_k \in \mathcal{T}_{\mathsf{in}}$ satisfying $\mathsf{AIM}(\mathsf{iv}, \mathsf{pt}_k) = \mathsf{ct}$. If $\mathcal{B}$ finds a match pt_k, $\mathcal{B}$ outputs it as a pre-image for the AIM, otherwise $\mathcal{B}$ outputs $\perp$.

Given the algorithm of $\mathcal{B}$ as above, the probability that $\mathcal{A}$ wins is bounded as below.

$$\begin{aligned}\Pr[\mathcal{A}\text{ wins}] &= \Pr[\mathcal{A}\text{ wins} \wedge \mathcal{B}\text{ aborts}] + \Pr[\mathcal{A}\text{ wins} \wedge \mathcal{B}\text{ outputs } \perp] + \Pr[\mathcal{A}\text{ wins} \wedge \mathcal{B}\text{ outputs pt}] \\ &\leq \Pr[\mathcal{B}\text{ aborts}] + \Pr[\mathcal{A}\text{ wins} \mid \mathcal{B}\text{ outputs } \perp] + \Pr[\mathcal{B}\text{ outputs pt}] \end{aligned} \quad (1)$$

We define Q_{c}, Q_1 and Q_2 as the number of queries made by $\mathcal{A}$ to random oracles H_{c}, H_1 and H_2, respectively. Then we can bound the probability that $\mathcal{B}$ aborts (The first term on the RHS of (1)) as follows.

Algorithm 3: $H_{\mathsf{c}}(q_{\mathsf{c}} = (\mathsf{salt}, k, i, \mathsf{seed}))$:

$r \xleftarrow{\$} \{0,1\}^{2\lambda}$.

if $r \in \mathsf{Bad}$ **then**

  abort.

$r \to \mathsf{Bad}$.

$\left(\mathsf{pt}_k^{(i)}, a_k^{(i)}, c_k^{(i)}, (t_{k,j}^{(i)})_{j\in[\ell]}\right) \xleftarrow{\$} \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \times (\mathbb{F}_{2^n})^{\ell}$

$\left(q_{\mathsf{c}}, r, \mathsf{pt}_k^{(i)}, a_k^{(i)}, c_k^{(i)}, (t_{k,j}^{(i)})_{j\in[\ell]}\right) \to \mathcal{Q}_{\mathsf{c}}$.

Return $\left(r, \mathsf{pt}_k^{(i)}, a_k^{(i)}, c_k^{(i)}, (t_{k,j}^{(i)})_{j\in[\ell]}\right)$.

$$\begin{aligned}
\Pr[\mathcal{B} \text{ aborts}] &= (\#\text{times an } r \text{ is sampled}) \cdot \Pr[\mathcal{B} \text{ aborts at that sample}] \\
&\le (Q_{\mathsf{c}} + Q_1 + Q_2) \cdot \frac{\max|\mathsf{Bad}|}{2^{2\lambda}} = (Q_{\mathsf{c}} + Q_1 + Q_2) \cdot \frac{Q_{\mathsf{c}} + (\tau N + 1)Q_1 + 2Q_2}{2^{2\lambda}} \\
&\le \frac{(\tau N + 1)(Q_{\mathsf{c}} + Q_1 + Q_2)^2}{2^{2\lambda}} = \frac{(\tau N + 1)Q^2}{2^{2\lambda}},
\end{aligned} \tag{2}$$

where $Q = Q_{\mathsf{c}} + Q_1 + Q_2$.

We now analyze $\Pr[\mathcal{A} \text{ wins} \mid \mathcal{B} \text{ outputs } \perp]$ (The second term on the RHS of (1)), which means pt corresponding to $(\mathsf{iv}, \mathsf{ct})$ is not found. We parse it into two cases, which correspond to cheating in the first round and the second round.

Cheating in the first round. Let $q_1 \in \mathcal{Q}_1$ be a query to H_1, and $h_1 = ((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]}$ be its corresponding answer. We collect the set of indices $k \in [\tau]$ representing "good executions" such that $\mathcal{T}_{\mathsf{in}}[q_1, k]$ is nonempty and $v_k = 0$, say $G_1(q_1, h_1)$. For $k \in G_1(q_1, h_1)$, the challenges $(\epsilon_{k,j})_{j\in[\ell+1]}$ were sampled such that the multiplication check protocol presented in the Section (2.2) is passed in that repetition. By Lemma (1), since h_1 is sampled uniformly at random, this happens with probability at most $1/2^{\lambda}$.

Lemma 1. *If the secret-shared input $(x_j, y, z_j)_{j\in[C]}$ contains an incorrect multiplication triple, or if the shares of $((a_j, y)_{j\in[C]}, c)$ form an incorrect dot product, then the parties output* `Accept` *in the sub-protocol with probability at most $1/2^{\lambda}$.*

Proof. Let $\Delta_{z_j} = z_j - x_j \cdot y$ and $\Delta_c = -\sum_{j\in[C]} a_j \cdot y + c$. Then

$$\begin{aligned}
v &= \sum_{j\in[C]} \epsilon_j \cdot z_j - \alpha \cdot y + c \\
&= \sum_{j\in[C]} \epsilon_j \cdot z_j - \sum_{j\in[C]} \epsilon_j \cdot x_j \cdot y - \sum_{j\in[C]} a_j \cdot y + c \\
&= \sum_{j\in[C]} \epsilon_j \cdot (z_j - x_j \cdot y) - \sum_{j\in[C]} a_j \cdot y + c \\
&= \sum_{j\in[C]} \epsilon_j \cdot \Delta_{z_j} + \Delta_c
\end{aligned}$$

Define a multivariate polynomial

$$Q(X_1, \ldots, X_C) = X_1 \cdot \Delta_{z_1} + \cdots + X_C \cdot \Delta_{z_C} + \Delta_c$$

Algorithm 4: $H_1(q_1 = \sigma_1)$:

Parse σ_1 as $\left(\mathsf{salt}, ((\mathsf{com}_k^{(i)})_{i\in[N]}, \Delta\mathsf{pt}_k, \Delta c_k, (\Delta t_{k,j})_{j\in[\ell]})_{k\in[\tau]}\right)$.
for $k \in [\tau], i \in [N]$ **do**
    $\mathsf{com}_k^{(i)} \to \mathsf{Bad}$.
```
// If the committed seed is known for some k,i, then B records the shares of the
   secret key and the multiplication output values for that party, derived from
   that seed and the offsets in σ1
```
for $k \in [\tau], i \in [N] : \exists \mathsf{seed}_k^{(i)} : ((\mathsf{salt}, k, i, \mathsf{seed}_k^{(i)}), \mathsf{com}_k^{(i)}, \mathsf{pt}_k^{(i)}, a_k^{(i)}, c_k^{(i)}, (t_{k,j}^{(i)})_{j\in[\ell]}) \in \mathcal{Q}_\mathsf{c}$ **do**
    if $i = 1$ **then**
        $\mathsf{pt}_k^{(i)} \leftarrow \mathsf{pt}_k^{(i)} + \Delta\mathsf{pt}_k, c_k^{(i)} \leftarrow c_k^{(i)} + \Delta c_k$ and $(t_{k,j}^{(i)} \leftarrow t_{k,j}^{(i)} + \Delta t_{k,j})_{j\in[\ell]}$
    $(\mathsf{pt}_k^{(i)}, c_k^{(i)}, (t_{k,j}^{(i)}))_{j\in[\ell]} \to \mathcal{T}_\mathsf{sh}[q_1, k, i]$
```
// If the shares of the various elements are known for every party in that
   repetition, B records the resulting secret key, multiplication inputs and S-box
   outputs
```
for *each* $k : \forall i, \mathcal{T}_\mathsf{sh}[q_1, k, i] \neq \emptyset$ **do**
    $\mathsf{pt}_k \leftarrow \sum_i \mathsf{pt}_k^{(i)}, c_k \leftarrow \sum_i c_k^{(i)}, a_k \leftarrow \sum_i a_k^{(i)}, (t_{k,j} \leftarrow \sum_i t_{k,j}^{(i)})_{j\in[\ell]}$ and $t_{k,\ell+1}^{(i)} = A_\mathsf{iv} \cdot t_{k,*}^{(i)} + b_\mathsf{iv}$ where $t_{k,*}^{(i)} = [t_{k,1}^{(i)}| \dots |t_{k,\ell}^{(i)}]$ is the output shares of the first ℓ S-boxes.
    Derive the binary matrix $A_\mathsf{iv} \in (\mathbb{F}_2^{n\times n})^\ell$ and the vector $b_\mathsf{iv} \in \mathbb{F}_2^n$ from the initial vector iv.
    for $j \in [\ell]$ **do**
        Set $x_{k,j} = t_{k,j}$ and $z_{k,j} = (\mathsf{pt}_k)^{2^{e_j}}$.
    for $j = \ell + 1$ **do**
        Set $x_{k,j} = A_\mathsf{iv} \cdot t_{k,*} + b_\mathsf{iv}$ where $t_{k,*} = [t_{k,1}| \dots |t_{k,\ell}]$ is the output shares of the first ℓ S-boxes and $z_{k,j} = (x_{k,j})^{2^{e_*}} + \mathsf{ct} \cdot x_{k,j}$.
    $(\mathsf{pt}_k) \to \mathcal{T}_\mathsf{in}[q_1, k]$.
$r \xleftarrow{\$} \{0,1\}^{2\lambda}$.
if $r \in \mathsf{Bad}$ **then**
    abort.
$r \to \mathsf{Bad}$.
$(q_1, r) \to \mathcal{Q}_1$.
```
// Compute the multiplication check protocol values.
```
$(\epsilon_{k,j})_{j\in[\ell+1]} \leftarrow \mathsf{Expand}(r)$.
for *each* $k : \mathcal{T}_\mathsf{in}[q_1, k] \neq \emptyset$ **do**
    $\alpha_k = \sum_{j\in[\ell+1]} \epsilon_j \cdot x_j + a_k$.
    $v_k = \sum_{j\in[\ell+1]} \epsilon_j \cdot z_{k,j} - \alpha_k \cdot \mathsf{pt} + c_k$.
Return r.

Algorithm 5: $H_2(q_2 = (h_1, \sigma_2))$:

$h_1 \to \mathsf{Bad}$.
$r \xleftarrow{\$} \{0,1\}^{2\lambda}$.
if $r \in \mathsf{Bad}$ **then**
  abort.
$r \to \mathsf{Bad}$.
$(q_2, r) \to \mathcal{Q}_2$.
Return r.

over $\mathbb{F}_{2^n}$ and note that $v = 0$ if and only if $Q(\epsilon_1, \ldots, \epsilon_C) = 0$. In the case of a cheating prover, Q is nonzero, and by the multivariate version of the Schwartz-Zippel lemma, the probability that $Q(\epsilon_1, \ldots, \epsilon_C) = 0$ is at most $1/2^\lambda$, since Q has total degree 1 and $(\epsilon_1, \ldots, \epsilon_C)$ is chosen uniformly at random. □

Given $\mathcal{B}$ outputs $\perp$, the number of elements $\#G_1(q_1, h_1)|_\perp \sim X_{q_1}$ where $X_{q_1} = \mathcal{B}(\tau, p_1)$, where $\mathcal{B}(\tau, p_1)$ is the binomial distribution with τ events, each with success probability $p_1 = 1/2^\lambda$. We select the query-response pair $(q_{\mathsf{best}_1}, h_{\mathsf{best}_1})$ such that $\#G_1(q_1, h_1)$ is the maximum. Then, the following holds.

$$\#G_1(q_{\mathsf{best}_1}, h_{\mathsf{best}_1})|_\perp \sim X = \max_{q_1 \in \mathcal{Q}_1} \{X_{q_1}\}.$$

Cheating in the second round. Let $q_2 = (h_1, \sigma_2)$ be a query to H_2. Note that q_2 can only be used in the winning EUF-KO game when the corresponding $(q_1, h_1) \in \mathcal{Q}_1$ exists. For the bad repetition $k \in [\tau] \backslash G_1(q_1, h_1)$, either $\mathcal{T}_{\mathsf{in}}[q_1, k]$ is empty (which means verification fails so that $\mathcal{A}$ loses) or $v_k \neq 0$ but the verification passes. Hence, it should be the case that one of the N parties cheated. Since $h_2 = (\bar{i}_k)_{k \in [\tau]} \in [N]^\tau$ is distributed uniformly at random, the probability that one of the N parties have cheated for all bad executions k is

$$\left(\frac{1}{N}\right)^{\tau - \#G_1(q_1, h_1)} \leq \left(\frac{1}{N}\right)^{\tau - \#G_1(q_{\mathsf{best}_1}, h_{\mathsf{best}_1})}.$$

To sum up, we can analyze the probability that $\mathcal{A}$ wins conditioning on $\mathcal{B}$ outputting $\perp$ is

$$\Pr[\mathcal{A} \text{ wins} \mid \mathcal{B} \text{ outputs } \perp] \leq \Pr[X + Y = \tau], \tag{3}$$

where X is as before, and $Y = \max_{q_2 \in \mathcal{Q}_2}\{Y_{q_2}\}$ where Y_{q_2} variables are independently and identically distributed as $\mathcal{B}(\tau - X, 1/N)$.

Finally, combining (1), (2) and (3) all together, we obtain the following.

$$\Pr[\mathcal{A} \text{ wins}] \leq \frac{(\tau N + 1) \cdot Q^2}{2^{2\lambda}} + \Pr[X + Y = \tau] + \Pr[\mathcal{B} \text{ outputs } \mathsf{pt}],$$

where $Q = Q_{\mathsf{c}} + Q_1 + Q_2$, X and Y are defined as above. Setting KeyGen as an ϵ_{OWF}-secure OWF, we achieve (1) as desired.

□

Theorem 2 (EUF-CMA Security of AIMer)**.** *Assume that* $\mathsf{CommitAndExpand}, H_1, H_2$ *and* Expand *are modeled as random oracles, the seed tree construction is computationally hiding, the* (N, τ, λ) *parameters of* AIMer *are appropriately chosen, and that the key generation is a secure one-way function. Then, the* AIMer *signature scheme is EUF-CMA-secure.*

Proof. Let $\mathcal{A}$ be an EUF-CMA adversary for given $(\mathsf{iv}, \mathsf{ct})$. Let G_0 be the original EUF-CMA game and $\mathcal{B}$ be an EUF-KO adversary that simulates the EUF-CMA game to $\mathcal{A}$. When $\mathcal{A}$ queries one of the random oracles, $\mathcal{B}$ checks if the query has been recorded so that it sends back the recorded answer if so, and otherwise, it records a pair of query and result it retrieves and forwards the answer to $\mathcal{A}$.

- G_0: $\mathcal{B}$ knows the secret key pt for the forwarded public key $(\mathsf{iv}, \mathsf{ct})$.

- G_1: $\mathcal{B}$ replaces real signatures with simulated ones no longer using pt. $\mathcal{B}$ uses the EUF-KO challenge $\mathsf{pt}^*(\neq \mathsf{pt})$ in its simulation with $\mathcal{A}$.

We define G_0(resp. G_1) as the probability that $\mathcal{A}$ succeeds in Game G_0(resp. G_1). The advantage of $\mathcal{A}$ is $\epsilon_{\text{CMA}} = \mathsf{G}_0 = (\mathsf{G}_0 - \mathsf{G}_1) + \mathsf{G}_1$.

HYBRID ARGUMENTS. We bound $(\mathsf{G}_0 - \mathsf{G}_1)$ by defining a sequence of games to connect G_0 and G_1 and constructing hybrid arguments. Upon receiving a signing query from $\mathcal{A}$, $\mathcal{B}$ simulates a signature using randomly sampled pt^*, selects one of the party $\mathcal{P}_{i^*}$ for cheating in the verification and the broadcast of the output shares $v_k^{(i^*)}$ so that it passes multiplication checking protocols. We show that the signature values are sampled from a distribution that is computationally indistinguishable from that of real signatures while it is sampled independently of pt^*. $\mathcal{B}$ sets the random oracle H_1 and H_2 to output uniform random challenges $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]}$ and $(\bar{i}_k)_{k\in[\tau]}$, respectively. The definition of subgames and hybrid arguments are the same as in the EUF-CMA proof in [KZ22] (Theorem 7 in Appendix) except that we do not have to cheat on the broadcast of party $P_{\bar{i}_k}$'s output share $\mathsf{ct}_k^{(\bar{i}_k)}$, since the output broadcast is implicit in our protocol.

1. In G_0, $\mathcal{B}$ knows pt so that it computes signatures honestly. $\mathcal{B}$ aborts only if the salt that it samples in Phase 1 has already been queried.

2. $\mathcal{B}$ randomly chooses h_2 and programs the random oracle H_2 to output h_2 when queried in Phase 4. The unopened parties $(\bar{i}_k)_{k\in[\tau]}$ is derived by expanding h_2. Simulation is aborted if the queries to H_2 have been made previously.

3. $\mathcal{B}$ replaces the seed of the unopened parties $\mathsf{seed}_k^{(\bar{i}_k)}$ in the binary tree by a random element for each $k \in [\tau]$. It is indistinguishable from the previous subgame since the tree structure is computationally hiding.

4. $\mathcal{B}$ replaces the outputs of $\mathsf{CommitAndExpand}(\mathsf{salt}, k, \bar{i}_k, \mathsf{seed}_k^{(\bar{i}_k)})$ by random elements and programs the random oracle H_c to output the same values for the respective queries. $\mathcal{B}$ aborts if the replaced commitment value collides with that in $\mathsf{CommitAndExpand}(x)$ where x is queried by $\mathcal{A}$.

5. $\mathcal{B}$ randomly chooses h_1 and programs the random oracle H_1 to output h_1 in Phase 2. The checking values $((\epsilon_{k,j})_{j\in[\ell+1]})_{k\in[\tau]}$ is derived by expanding h_1. Simulation is aborted if the queries to H_1 have been made previously.

6. $\mathcal{B}$ replaces $\alpha_k^{(\bar{i}_k)}$ with a uniformly random value and sets $v_k^{(\bar{i}_k)} \leftarrow -\sum_{i\neq\bar{i}_k} v_k^{(i)}$. Note that if the multiplication triple is wrong, then $v_k^{(\bar{i}_k)} \leftarrow -\sum_{i\neq\bar{i}_k} v_k^{(i)}$ is different from an honest value derived from legitimate calculation. However $(\bar{i}_k)$ is unopened and the multiplication check is still passed.

7. $\mathcal{B}$ sets $(\Delta t_{k,j})_{j\in[\ell]}$ and Δc_k to random values in Phase 1.

8. $\mathcal{B}$ replaces the real pt by a random key pt^* as $\mathsf{pt}_k^{(\bar{i}_k)}$ is independent from the seeds $\mathcal{A}$ observes. The distribution of $\Delta\mathsf{pt}_k$ is not changed and $\mathcal{A}$ has no information about pt^*.

If the algorithm is not aborted, above games are all indistinguishable to each other, which results the simulated signatures in G_1 and the real signatures in G_0 are indistinguishable. The abort happens when:

- A_1 : The salt it sampled has been used before.

- A_2 : The committed value it replaces is queried.

- A_3 : Queries to H_1 and H_2 have been made previously.

Let Q_{salt} be the number of different salts queried during the game (by both $\mathcal{A}$ and $\mathcal{B}$), Q_{c} be the number of queries made to Commit by $\mathcal{A}$ including those made during signature queries and Q_1(resp. Q_2) be the number of queries made to H_1(resp. H_2) during the game. Then the probability of each event occurring is bounded by $\Pr[A_1] \leq Q_{\mathsf{salt}}/2^{2\lambda}$, $\Pr[A_2] \leq Q_{\mathsf{c}}/2^{\lambda}$, and $\Pr[A_3] \leq Q_1/2^{2\lambda} + Q_2/2^{2\lambda}$.

Therefore

$$\begin{aligned}\Pr[\mathcal{B} \text{ aborts}] &\leq Q_{\mathsf{salt}}/2^{2\lambda} + Q_{\mathsf{c}}/2^{\lambda} + Q_1/2^{2\lambda} + Q_2/2^{2\lambda} \\ &= (Q_{\mathsf{salt}} + Q_1 + Q_2)/2^{2\lambda} + Q_{\mathsf{c}}/2^{\lambda} \\ &\leq (Q_1 + Q_2)/2^{2\lambda-1} + Q_{\mathsf{c}}/2^{\lambda} \quad (\because Q_{\mathsf{salt}} \leq Q_1 + Q_2) \\ &\leq Q/2^{\lambda} \quad (\text{where } Q = Q_1 + Q_2 + Q_{\mathsf{c}})\end{aligned}$$

and

$$\begin{aligned}\mathsf{G}_0 - \mathsf{G}_1 \leq& Q_s \cdot (\epsilon_{\text{TREE}} + \Pr[\mathcal{B} \text{ aborts}]) \\ \leq& Q_s \cdot (\epsilon_{\text{TREE}} + Q/2^{\lambda}),\end{aligned}$$

where Q_s be the total number of signature queries.

BOUNDING G_1. If $\mathcal{A}$ outputs a valid signature in G_1, then $\mathcal{B}$ outputs a valid signature in the EUF-KO game. Finally we have

$$\mathsf{G}_1 \leq \epsilon_{\text{KO}} \leq \epsilon_{\text{OWF}} + \frac{(\tau N + 1)Q^2}{2^{2\lambda}} + \Pr[X + Y = \tau],$$

where the bound on the advantage ϵ_{KO} of a EUF-KO attacker follows from Theorem 1. We conclude that

$$\epsilon_{\text{CMA}} \leq \epsilon_{\text{OWF}} + \frac{(\tau N + 1)Q^2}{2^{2\lambda}} + \Pr[X + Y = \tau] + Q_s \cdot (\epsilon_{\text{TREE}} + Q/2^{\lambda}).$$

Assuming that the seed tree construction is hiding (so that ϵ_{TREE} is negligible), that key generation is a one-way function and that parameters (N, τ, λ) are appropriately chosen implies that ϵ_{CMA} is negligible in λ. □

5.2 Information-Theoretic Security of AIM in the Random Permutation Model

In this section, we consider the one-wayness of AIM. More precisely, we will prove the *everywhere preimage resistance* [RS04] of AIM when the underlying S-boxes are modeled as public random permutations and iv is (implicitly) fixed.[2]

For simplicity, we will assume that $\ell = 2$. The security of AIM with $\ell > 2$ is similarly proved. In the public permutation model and in the single-user setting, AIM is defined as

$$\mathsf{AIM}(\mathsf{pt}) = S_3(A_1 \cdot S_1(\mathsf{pt}) \oplus A_2 \cdot S_2(\mathsf{pt}) \oplus b) \oplus \mathsf{pt}$$

for $\mathsf{pt} \in \{0,1\}^n$, where S_1, S_2, S_3 are independent public random permutations, and A_1 and A_2 are fixed $n \times n$ invertible matrices, and b is a fixed $n \times 1$ vector over $\mathbb{F}_2$.

In the preimage resistance experiment, a computationally unbounded adversary $\mathcal{A}$ with oracle access to S_i, $i = 1, 2, 3$, selects and announces a point $\mathsf{ct} \in \{0,1\}^n$ before making queries to the underlying permutations. After making q forward and backward queries in total, $\mathcal{A}$ obtains a *query history*

$$\mathcal{Q} = \{(i_j, x_j, y_j)\}_{j=1}^{q}$$

[2] We do not claim that the algebraic S-boxes of AIM behave like random permutations. The point of the provable security of AIM is that one cannot break the one-wayness of AIM without exploiting any particular properties of the underlying S-boxes.

such that $S_{i_j}(x_j) = y_j$ and $\mathcal{A}$'s j-th query is either $S_{i_j}(x_j) = y_j$ or $S_{i_j}^{-1}(y_j) = x_j$ for $j = 1, \ldots q$. We say that *$\mathcal{A}$ succeeds in finding a preimage of* ct if its query history $\mathcal{Q}$ contains three queries $S_1(x_1) = y_1$, $S_2(x_2) = y_2$ and $S_3(x_3) = y_3$ such that $x_1 = x_2$, $x_3 = A_1 \cdot y_1 \oplus A_2 \cdot y_2 \oplus b$ and $\mathsf{ct} = y_3 \oplus \mathsf{pt}$. In this case, we say that $\mathcal{A}$ wins the preimage-finding game, breaking the one-wayness of AIM. Assuming that $\mathcal{A}$ is information-theoretic, we can prove that $\mathcal{A}$'s winning probability, denoted $\mathbf{Adv}_{\mathsf{AIM}}^{\mathsf{epre}}(q)$, is upper bounded as follows.

$$\mathbf{Adv}_{\mathsf{AIM}}^{\mathsf{epre}}(q) \leq \frac{2q}{2^n - q}. \tag{4}$$

PROOF OF (4). Since $\mathcal{A}$ is information-theoretic, we can assume that $\mathcal{A}$ is deterministic. Furthermore, we assume that $\mathcal{A}$ does not make any redundant query. We will also slightly modify $\mathcal{A}$ so that whenever $\mathcal{A}$ makes a (forward or backward) query to S_1 (resp. S_2) obtaining $S_1(x) = y$ (resp. $S_2(x) = y$), $\mathcal{A}$ makes an additional *forward* query to S_2 (resp. S_1) with x *for free*. This additional query will not degrade $\mathcal{A}$'s preimage-finding advantage since $\mathcal{A}$ is free to ignore it.

An evaluation $\mathsf{AIM}(\mathsf{pt}) = \mathsf{ct}$ consists of three S-box queries. Among the three S-box queries, the lastly asked one is called the *preimage-finding query*. We distinguish two cases.

Case 1. The preimage-finding query is made to either S_1 or S_2. Since $\mathcal{A}$ consecutively obtains a pair of queries of the form $S_1(x) = y_1$ and $S_2(x) = y_2$, any preimage-finding query to either S_1 or S_2 should be forward. If it is $S_1(x)$ (without loss of generality), then there should be queries $S_2(x) = y$ for some y and $S_3(z) = x \oplus \mathsf{ct}$ for some z that have already been made by $\mathcal{A}$. In order for $S_1(x)$ to be the preimage-finding query, it should be the case that

$$S_3(A_1 \cdot S_1(x) \oplus A_2 \cdot S_2(x) \oplus B) = x \oplus \mathsf{ct}$$

or equivalently,

$$S_1(x) = A_1^{-1} \cdot (z \oplus b \oplus A_2 \cdot y)$$

which happens with probability at most $\frac{1}{2^n - q}$. Therefore, the probability of this case is upper bounded by $\frac{q}{2^n - q}$.

Case 2. The preimage-finding query is made to S_3. In order to address this case, we use the notion of a *wish list*, which was first introduced in [AFK+11]. Namely, whenever $\mathcal{A}$ makes a pair of queries $S_1(x) = y_1$ and $S_2(x) = y_2$, the evaluation

$$S_3 : A_1 \cdot y_1 \oplus A_2 \cdot y_2 \oplus b \mapsto x \oplus \mathsf{ct}$$

is included in the wish list $\mathcal{W}$. In order for an S_3-query to complete an evaluation $\mathsf{AIM}(\mathsf{pt}) = \mathsf{ct}$ for any pt, at least one "wish" in $\mathcal{W}$ should be made come true. Each evaluation in $\mathcal{W}$ is obtained with probability at most $\frac{1}{2^n - q}$, and $|\mathcal{W}| \leq q$. Therefore, the probability of this case is upper bounded by $\frac{q}{2^n - q}$.

Overall, we can conclude that

$$\mathbf{Adv}_{\mathsf{AIM}}^{\mathsf{epre}}(q) \leq \frac{2q}{2^n - q}.$$

ONE-WAYNESS IN THE MULTI-USER SETTING. In the multi-user setting with u users, $\mathcal{A}$ is given u different target images, where the adversarial goal is to invert any of the target images. In this setting, the adversarial preimage finding advantage is upper bounded by

$$\frac{2uq}{2^n - q}. \tag{5}$$

The proof of (5) follows the same line of argument as the single-user security proof. The difference is that the probability that each query to either S_1 or S_2 becomes the preimage-finding one is upper bounded by $\frac{uq}{2^n - q}$ and the size of the wish list (in the second case) is upper bounded by uq.

We note that the above bound does not mean that AIM provides only the birthday-bound security in the multi-user setting. The straightforward birthday-bound attack is mitigated since AIM is based on a distinct linear layer for every user.

6 Security Evaluation

6.1 Summary of Expected Security Strength

The AIMer signature scheme provides three levels of security: L1 (AES-128), L3 (AES-192), and L5 (AES-256). Each security level corresponds to the security of AES in the parentheses, and it implies that we expect AIMer with L1, L3, and L5 parameters to be as secure as AES-128, AES-192, AES-256 respectively, against both classical and quantum attacks. In this section, we examine the concrete security of the three components of AIMer: the non-interactive zero-knowledge proof of knowledge (NIZKPoK), the one-way function, and the hash functions.

SECURITY OF THE NIZKPOK SYSTEM. The NIZKPoK system in AIMer is BN++ [KZ22] with slight modification on the hash functions in use. We will look into this modification later in this section. The security of BN++ is proved in the random oracle model, and the security of AIMer can be proved similarly in the random oracle model.

In the quantum-accessible random oracle model (QROM), an adversary is allowed to make superposition queries to the random oracle. The NIZKPoK system in AIMer (and BN++) follows the spirit of the Fiat-Shamir transform [FS87], and there has been a significant amount of research on the QROM security of the Fiat-Shamir transform [LZ19, DFMS19, DFM20, DFMS22a, DFMS22b]. The NIZKPoK system of AIMer should be seen as a variant of the original Fiat-Shamir transform, while its security is not immediate from the above results, and we will prove it as a future work.

The parameters N and τ are fixed based on the soundness analysis given in [KZ22]; we see that an attacker should make at least 2^λ guesses in order to produce a valid forgery without any knowledge of the secret key. Since a single guess involves at least $3N$ hash or XOF calls (where a single call of hash is more costly than AES), AIMer with our recommended sets of parameters would provide a sufficient level of security.

SECURITY OF AIM. AIM is a one-way function, which does not follow the traditional design rationale of symmetric primitives. It takes random strings iv and pt as input, and outputs $\mathsf{ct} = \mathsf{AIM}(\mathsf{iv}, \mathsf{pt})$. We expect that finding pt^* for a given pair $(\mathsf{iv}, \mathsf{ct})$ such that $\mathsf{ct} = \mathsf{AIM}(\mathsf{iv}, \mathsf{pt}^*)$ is as hard as key recovery of AES with the same security level. To support our claim, we not only prove the information-theoretic security of AIM but also investigate its security against brute-force attack, algebraic attacks, statistical attacks, and quantum attacks in Section 6.3.

We prove the everywhere preimage resistance [RS04] of AIM in the random permutation model. The one-wayness is proved assuming that the S-boxes are modeled as public random permutations. Although our choice of S-boxes is far from a random permutation, the proof itself exhibits that AIM is one-way unless any particular properties of the underlying S-boxes are exploited.

For the algebraic attacks, we analyzed the security of AIM against XL, Gröbner basis algorithm, and Dinur's equation solving algorithm [Din21]. We found out that AIM is secure even if the equations generated while running the XL algorithm are all independent. All the algebraic attacks on AIM requires more gate-count complexity than those on AES, or requires more than 2^λ memory bits. For the statistical attacks, we bounded the weights of differential/linear trails although statistical attacks are impossible with a single input-output pair. AIM has the minimum differential weight less than λ, while it does not link to any collision. For the quantum attacks, we looked into Grover's algorithm, quantum algebraic attacks, and quantum generic attacks. The most powerful attack among them turns out to be the Grover's algorithm while its complexity against AIM is not lower than applied to AES with the same security level.

In the multi-user setting, we expect that finding one of pt_i given multiple pairs $\{(\mathsf{iv}_i, \mathsf{ct}_i)\}$ such that $\mathsf{ct}_i = \mathsf{AIM}(\mathsf{iv}_i, \mathsf{pt}_i)$ for some i is hard assuming that iv's are randomly chosen. If iv's are arbitrarily chosen, collision of ct_i can be connected to a forgery. For example, if an IV value iv^* collides q times in a set of public keys, an attacker may compute the function $\mathsf{AIM}(\mathsf{iv}^*, \mathsf{pt})$ for c times with distinct pt's. Then, the probability of collision is approximately $qc/2^n$, which implies a security degradation.

Except the risk of collision, multiple pairs $\{(\mathsf{iv}_i, \mathsf{ct}_i)\}$ do not lead to a strengthened attack on AIM to the best of our knowledge. For algebraic attacks, any two sets of equations built for distinct pt's are not compatible. For statistical attacks, any two public-key pairs are not compatible to differential/linear cryptanalysis if corresponding pt's are distinct.

HASH FUNCTION SECURITY. The AIMer signature scheme requires a lot of calls to hash functions and extendable output functions (XOFs). All the hash functions and XOFs are based on NIST-standardized XOF SHAKE [NIS15]. SHAKE-128 is used for the L1 parameters, and SHAKE-256 is used for the L3 and L5 parameters. All the hash functions use 2λ-bit digests of the SHAKE output.

We expect the concrete security provided by SHAKE for collision and preimage resistance as claimed in [NIS15]. For $\lambda = 128, 256$, the preimage resistance of SHAKE-λ with k-bit digest is claimed to be $\min(2^k, 2^{2\lambda})$ in the classical setting, and a cryptographic hash function with k-bit digest is generally believed to have $O(2^{k/2})$ preimage resistance in the quantum setting [Gro96]. In both cases, hash functions with 2λ-bit digests provide λ-bit preimage resistance. For collision resistance, while a generic quantum algorithm of finding a hash collision is of complexity $O(2^{k/3})$ when the output size is k bits [BHT98], Bernstein pointed out that the quantum hash collision algorithm has worse performance compared to classical algorithms in practice [Ber09]. Since it is claimed that k-bit digests of SHAKE-λ has collision resistance of $\min(2^{k/2}, \lambda)$ against classical attacks, the 2λ-bit digest also allows λ-bit collision resistance against classical and quantum attacks.

6.2 Soundness Analysis

In this section, we analyze the soundness error of the AIMer signature scheme to determine the set of parameters (λ, N, τ). A more formal analysis is given in Section 5.1. Let τ_1 and τ_2 denote the number of repetitions for which the attacker need to make correct guesses on the first challenge $\epsilon_{k,j}$ in Phase 2 and the second challenge $\bar{i}_k$ in Phase 4 in Algorithm 1, respectively. Then, it should be the case that $\tau = \tau_1 + \tau_2$. For $i = 1, 2$, let P_i be the probability that the attacker makes correct guesses for τ_i challenges in the i-th challenge space.

The first challenge is sampled from the set of size 2^n, so the probability of correctly guessing τ_1 challenges in the first challenge space is given as

$$P_1 = \sum_{k=\tau_1}^{\tau} \binom{\tau}{k} p^k \cdot (1-p)^{\tau-k}$$

where $p = 2^{-\lambda}$. On the other hand, since the second challenge space is of size N, and the attacker needs to make correct guesses in the remaining repetitions, one has

$$P_2 = 1/N^{\tau_2} = 1/N^{\tau-\tau_1}.$$

Overall, the attack complexity is given as

$$C = \min_{0 \le \tau_1 \le \tau} (1/P_1 + 1/P_2).$$

Our parameters are set in a way such that $C \ge 2^\lambda$.

6.3 Known Attacks to AIM

6.3.1 Brute-force Attack

Although a brute-force attack on a symmetric primitive is rather trivial (compared to public key cryptosystems), we estimate its gate-count complexity to compare the concrete security of AIM and AES.

By using addition chain exponentiation technique [Knu97], the numbers of required finite field multiplications are 11, 14, and 17 for AIM-I, AIM-III, and AIM-V, respectively (see Table 6). Assuming that a single $\mathbb{F}_{2^n}$-multiplication requires n^2 AND gates and n^2 XOR gates, the gate-count complexity of a brute-force attack is given as $2^{146.4}$, $2^{211.9}$, and 2^{277} for AIM-I, AIM-III, and AIM-V, respectively. It implies that a brute-force attack on AIM is more costly than AES for each category of security strength.

6.3.2 Algebraic Attacks

Since our attack model does not allow multiple evaluations for a single instance of AIM, we do not consider interpolation, higher-order differential, and cube attacks. As discussed in [KHS+22], we focus on the Gröbner basis and the XL attacks using a single evaluation of AIM. We also consider algebraic attacks which have been recently studied for MPC/ZK-friendly ciphers such LowMC [ARS+15] and large S-box-based ones.

How to Build Boolean Systems of Equations from AIM. There are multiple ways of building a system of equations from an evaluation of AIM. We can categorize them according to the number of (Boolean) variables and find the optimal choice of variables to obtain a system of the lowest degree. Since $\ell \in \{2, 3\}$ is recommended, we consider 4 types of systems of equations as follows.

1. Systems in n variables.
2. Systems in $2n$ variables.
3. Systems in $3n$ variables.
4. Systems in $4n$ variables (only for AIM-V).

Using the quadratic relation between an input and the output of each Mersenne S-box, we can establish a system of *quadratic* equations in $(\ell + 1)n$ variables. With fewer variables, the resulting systems would have higher degrees. The goal of this section is to find a system of equations of the lowest degree for each type, where such systems of equations are denoted $S_1, S_2, \ldots, S_{\mathsf{quad}}$, respectively. The optimal systems of equations will be defined using the following variables.

- x: the input of AIM, i.e., pt
- y_i: the output of $\mathsf{Mer}[e_i]$ for $i = 1, \ldots, \ell$
- z: the output of Lin

The underlying $\ell + 1$ Mersenne S-boxes determine explicit and implicit relations between these variables. For example, $\mathsf{Mer}[e_i]$ implicitly determines $3n$ quadratic equations in x and y_i, while y_i (resp. x) can be explicitly represented by a polynomial in x (resp. y_i). We can also explicitly represent y_i using y_j for $j \neq i$ or z as follows.

$$\mathsf{Mer}[e_i] \circ \mathsf{Mer}[e_j]^{-1}(y_j) = y_i = \mathsf{Mer}[e_i]\big(\mathsf{Mer}[e_*](z) \oplus \mathsf{ct}\big).$$

The degree of y_i with respect to z might be greater than the degree of $\mathsf{Mer}[e_i] \circ \mathsf{Mer}[e_*]$ due to the constant addition, while we will ignore the effect by ct for simplicity. Table 4 shows the degrees of all the possible explicit relations from AIM, and this table can be used to find the optimal systems of equations.

After exhaustive search, we found the optimal systems S_1, S_2, S_3 and S_{quad}. First, in order to obtain the S_1 systems, choose z as an n-bit variable. Then x and y_i can be represented as polynomials in z; x is of degree e_*, y_1 is of degree $\deg(\mathsf{Mer}[e_1] \circ \mathsf{Mer}[e_*])$, and y_3 (only for AIM-V) is of degree $\deg(\mathsf{Mer}[e_3] \circ \mathsf{Mer}[e_*])$ with respect to z. Let Lin' denote a linear function such that $y_2 = \mathsf{Lin}'(y_1, y_3, z)$ (which is uniquely determined by Lin). Then we have the following equation.

$$\big(\mathsf{Mer}[e_*](z) \oplus \mathsf{ct}\big)^{2^{e_2}} = \big(\mathsf{Mer}[e_*](z) \oplus \mathsf{ct}\big) \cdot \mathsf{Lin}'\Big(\mathsf{Mer}[e_1]\big(\mathsf{Mer}[e_*](z) \oplus \mathsf{ct}\big), \mathsf{Mer}[e_3]\big(\mathsf{Mer}[e_*](z) \oplus \mathsf{ct}\big), z\Big).$$

Relation	AIM-I	AIM-III	AIM-V
$\mathsf{Mer}[e_1]$	3	5	3
$\mathsf{Mer}[e_1]^{-1}$	43	77	171
$\mathsf{Mer}[e_2]$	27	29	53
$\mathsf{Mer}[e_2]^{-1}$	19	53	29
$\mathsf{Mer}[e_2] \circ \mathsf{Mer}[e_1]^{-1}$	9	121	103
$\mathsf{Mer}[e_1] \circ \mathsf{Mer}[e_2]^{-1}$	57	73	87
$\mathsf{Mer}[e_*]$	5	7	5
$\mathsf{Mer}[e_*]^{-1}$	77	55	205
$\mathsf{Mer}[e_1] \circ \mathsf{Mer}[e_*]$	5	7	5
$\mathsf{Mer}[e_2] \circ \mathsf{Mer}[e_*]$	27	29	53
$\mathsf{Mer}[e_*]^{-1} \circ \mathsf{Mer}[e_1]^{-1}$	67	78	171
$\mathsf{Mer}[e_*]^{-1} \circ \mathsf{Mer}[e_2]^{-1}$	64	100	110
$\mathsf{Mer}[e_3]$	-	-	7
$\mathsf{Mer}[e_3]^{-1}$	-	-	183
$\mathsf{Mer}[e_3] \circ \mathsf{Mer}[e_1]^{-1}$	-	-	173
$\mathsf{Mer}[e_3] \circ \mathsf{Mer}[e_2]^{-1}$	-	-	203
$\mathsf{Mer}[e_1] \circ \mathsf{Mer}[e_3]^{-1}$	-	-	37
$\mathsf{Mer}[e_2] \circ \mathsf{Mer}[e_3]^{-1}$	-	-	227
$\mathsf{Mer}[e_3] \circ \mathsf{Mer}[e_*]$	-	-	7
$\mathsf{Mer}[e_*]^{-1} \circ \mathsf{Mer}[e_3]^{-1}$	-	-	125

Table 4: Degrees of the compositions and the inverses of the Mersenne S-boxes of AIM.

Since every Mersenne S-box used in AIM is represented by $3n$ quadratic equations, the above system of equations can be seen as a system of $3n$ (Boolean) equations of degree

$$e_* + \max\big(\deg(\mathsf{Mer}[e_1] \circ \mathsf{Mer}[e_*]), \deg(\mathsf{Mer}[e_3] \circ \mathsf{Mer}[e_*])\big).$$

Second, in order to obtain the S_2 systems, we begin with x and y_2, and using $y_1 = \mathsf{Mer}[e_1](x)$ and $y_3 = \mathsf{Mer}[e_3](x)$ (only for AIM-V), we establish the following system of equations.

$$x \cdot y_2 = x^{2^{e_2}},$$
$$\mathsf{Lin}\big(\mathsf{Mer}[e_1](x), y_2, \mathsf{Mer}[e_3](x)\big) \cdot (x \oplus \mathsf{ct}) = \mathsf{Lin}\big(\mathsf{Mer}[e_1](x), y_2, \mathsf{Mer}[e_3](x)\big)^{2^{e_*}}$$

We note that $3n$ quadratic equations are obtained from the first equation, and $3n$ equations of degree $\max(e_1, e_3) + 1$ from the second one.

Third, in order to obtain the S_3 system for AIM-V, we begin with x, y_2 and y_3, and using $y_1 = \mathsf{Mer}[e_1](x)$, we establish the following system of equations.

$$x \cdot y_2 = x^{2^{e_2}}$$
$$x \cdot y_3 = x^{2^{e_3}}$$
$$\mathsf{Lin}\big(\mathsf{Mer}[e_1](x), y_2, y_3\big) \cdot (x \oplus \mathsf{ct}) = \mathsf{Lin}\big(\mathsf{Mer}[e_1](x), y_2, y_3\big)^{2^{e_*}}.$$

We note that $6n$ quadratic equations are obtained from the first and the second equations, and $3n$ equations of degree $e_1 + 1$ are from the third one.

Finally, the S_{quad} systems are quadratic with x and all the y_i's being variables. Using the implicit relations for all $\ell + 1$ S-boxes, we establish the following system of equations.

$$\begin{aligned}
x \cdot y_1 &= x^{2^{e_1}} \\
x \cdot y_2 &= x^{2^{e_2}} \\
&\vdots \\
x \cdot y_\ell &= x^{2^{e_\ell}} \\
\mathsf{Lin}(y_1, y_2, \ldots, y_\ell) \cdot (x \oplus \mathsf{ct}) &= \mathsf{Lin}(y_1, y_2, \ldots, y_\ell)^{2^{e_*}},
\end{aligned}$$

which can be extended to a system of $3(\ell+1)n$ quadratic equations in $(\ell+1)n$ variables.

Scheme	Name	#Var	Variables	(#Eq, Deg)	Gröbner Basis		XL		Dinur [Din21]	
					d_{reg}	Time	D	Time	Time	Memory
AIM-I	S_1	n	z	$(3n, 10)$	51	300.8	52	244.8	**137.3**	138.3
	S_2	$2n$	x, y_2	$(3n, 2) + (3n, 4)$	22	**214.9**	14	150.4	248.3	253.7
	S_{quad}	$3n$	x, y_1, y_2	$(9n, 2)$	20	222.8	12	**148.0**	330.1	346.3
AIM-III	S_1	n	z	$(3n, 14)$	82	474.0	84	375.3	**202.1**	203.3
	S_2	$2n$	x, y_2	$(3n, 2) + (3n, 6)$	31	**310.6**	18	203.0	377.5	382.9
	S_{quad}	$3n$	x, y_1, y_2	$(9n, 2)$	27	310.8	15	**194.1**	487.7	512.1
AIM-V	S_1	n	z	$(3n, 12)$	100	601.1	101	489.7	**264.1**	265.9
	S_2	$2n$	x, y_2	$(3n, 2) + (3n, 8)$	40	**406.2**	26	289.5	506.3	511.7
	S_3	$3n$	x, y_2, y_3	$(6n, 2) + (3n, 4)$	47	510.4	20	**260.6**	716.1	732.3
	S_{quad}	$4n$	x, y_1, y_2, y_3	$(12n, 2)$	45	530.3	19	266.1	854.4	897.7

Table 5: Optimal systems of equations and their security against algebraic attacks. $(\#\text{Eq}, \text{Deg}) = (a, b)$ means that the system contains a equations of degree b. The degree of regularity (resp. the target degree) of the system is denoted d_{reg} (resp. D). The time and the memory complexities are estimated in bits.

Table 5 summarizes the number of variables, the number of equations, and their degrees for the optimal systems of equations, and their security against the Gröbner basis attack, the XL attack, and Dinur's algorithm based on the polynomial method [Din21].

GRÖBNER BASIS ATTACK. The Gröbner basis attack is to solve a system of equations by computing its Gröbner basis. The complexity of Gröbner basis computation can be estimated using the *degree of regularity* of the system of equations [BFS04]. Consider a system of m equations in n variables $\{f_i\}_{i=1}^m$. Let d_i denote the degree of f_i for $i = 1, 2, \ldots, m$. If the system of equations is over-defined, i.e., $m > n$, then the degree of regularity can be estimated by the smallest of the degrees of the terms with non-positive coefficients for the following Hilbert series under the semi-regular assumption [Frö85].

$$\mathrm{HS}(z) = \frac{1}{(1-z)^n} \prod_{i=1}^{m} (1 - z^{d_i}).$$

Given the degree of regularity d_{reg}, the complexity of computing a Gröbner basis of the system is known to be

$$O\left(\binom{n + d_{reg}}{d_{reg}}^{\omega}\right)$$

where ω is the linear algebra constant.[3] See [KHS+22] for the details.

[3]It means that the complexity matrix multiplication of two $n \times n$ matrices is $O(n^\omega)$. We will conservatively set this constant to be 2 in this document.

For AIM, the system S_2 turns out to permit the most efficient computation of a Gröbner basis; the corresponding Hilbert series is given as

$$\frac{(1-z^2)^{5n}(1-z^d)^{3n}}{(1-z)^{2n}}$$

including the field equations of degree 2 in all variables in S_2, where $d = \max(e_1, e_3) + 1$. The estimated degrees of regularity and the corresponding time complexities of computing Gröbner bases are given in Table 5 for AIM-I, III, V.

XL ATTACK. The XL algorithm, proposed by Courtois et al. [CKPS00], can be viewed as a generalization of the relinearization attack [KS99]. For a system of m quadratic equations in n variables over $\mathbb{F}_2$, the XL algorithm extends the system of equations by multiplying all the monomials of degree at most $D-2$ for some $D > 2$ to obtain a larger number of linearly independent equations than the number of monomials appearing in the system. Since the number of monomials of degree at most $D-2$ is $\sum_{i=1}^{D-2}\binom{n}{i}$, the resulting system consists of $(\sum_{i=0}^{D-2}\binom{n}{i})m$ equations of degree at most D with at most $\sum_{i=1}^{D}\binom{n}{i}$ monomials of degree at most D. When the number of equations equals the number of monomials as D grows, one can solve the extended system of equations by linearization.

The complexity of the XL attack depends on the number of linearly independent equations obtained from the XL algorithm, while we can loosely upper bound the number of linearly independent equations by $(\sum_{i=0}^{D-2}\binom{n}{i})m$. Under the assumption that all the equations obtained from the XL algorithm are linearly independent, which is in favor of the attacker, we can search for the (smallest) degree D such that

$$\left(\sum_{i=0}^{D-2}\binom{n}{i}\right)m \geq T_D \tag{6}$$

where T_D denotes the exact number of monomials appearing in the extended system of equations, which is upper bounded by $\sum_{i=1}^{D}\binom{n}{i}$. Once D is fixed, the extended system of equations can be solved by trivial linearization whose time complexity is given as $O\left(T_D^{\omega}\right)$.

For AIM-I and III, the system S_{quad} permits the most efficient XL attack. For the system S_{quad}, the target degree D is determined as the smallest one satisfying

$$\left(\sum_{i=0}^{D-2}\binom{3n}{i}\right)9n \geq T_D$$

where the number of monomials T_D is assumed to be $\sum_{i=1}^{D}\binom{3n}{i}$.

On the other hand, the system S_3 is the most efficient system for AIM-V. We note that more careful analysis is required for the other systems of equations of different degrees with a particular structure. For example, the S_2 system of AIM-V consists of two types of equations of different degrees: $3n$ equations of degree 2, and $3n$ equations of degree d, all in x and y_2, where $d = \max(e_1, e_3) + 1$. We observe that each type of equations have 2^n solutions since y_2 is uniquely determined for each x, and this property makes one to compute the target degree in a different way.

With target degree D, the extended system of equations for S_2 is represented as

$$M\mathbf{v} = \begin{bmatrix} M_2 \\ \hdashline M_* \end{bmatrix}\mathbf{v} = \mathbf{c}$$

where $\mathbf{v}$ is a vector of monomials of degree at most D in x and y_2, M_2 (resp. M_*) is the matrix whose rows are the coefficients of the extended system from $\mathsf{Mer}[e_2]$ (resp. $\mathsf{Mer}[e_*]$), and $\mathbf{c}$ is the corresponding constant vector. The number of rows of M_2 is greater than that of M_* since the original system from $\mathsf{Mer}[e_2]$ has a lower degree than $\mathsf{Mer}[e_*]$. In order for the XL attack to work with the target degree D, the matrix M

should have full rank and the number of rows should not be smaller than the number of columns, so that $\mathbf{v}$ is uniquely determined.

On the other hand, the submatrix M_2 itself cannot have full rank since $M_2\mathbf{v} = \mathbf{c}$ should have 2^n solutions (one for each x) as its original system from $\mathsf{Mer}[e_2]$ does. More precisely, the nullity of M_2 should not be smaller than $\sum_{j=1}^{D}\binom{n}{j}$. Otherwise, it implies that there is a linear relation on the monomials consisting of only x variables, for example,

$$\sum_{\mathbf{a}} c_{\mathbf{a}}\mathbf{x}^{\mathbf{a}} = 0$$

where $\mathbf{x}^{\mathbf{a}} = \prod_{i=1}^{n} x_i^{a_i}$ for $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{a} = (a_1, \ldots, a_n)$ such that $\sum_{i=1}^{n} a_i \le D$, and $c_{\mathbf{a}}$ is a Boolean constant. This relation cannot hold for all $\mathbf{x}$, which is a contradiction. Then, for M to have full rank, the rank of M_* should be at least the nullity of M_2, yielding a necessary condition that the number of rows of M_* should be at least $\sum_{j=1}^{D}\binom{n}{j}$ provided that M has no nonzero column.[4] As the number of rows of M_* is the number of equations in the extended system from $\mathsf{Mer}[e_*]$, the target degree D should satisfy the following.

$$3n \cdot \sum_{j=0}^{D-d}\binom{2n}{j} \ge \sum_{j=1}^{D}\binom{n}{j}. \tag{7}$$

For the S_2 system of AIM-III and AIM-V, the target degree is determined by the minimum D satisfying (7), whereas it is not for AIM-I. The difference comes from the small value of $d = 4$ of AIM-I compared to $d = 6$ and $d = 8$ of AIM-III and AIM-V, respectively. A similar argument also holds for the S_3 system of AIM-V, but it does not determine the target degree either due to the small value of $d = 4$ in S_3.

Table 5 shows the target degree and corresponding attack complexity for each system of AIM. We note that the time complexity of the XL attack has been estimated under the strong assumption that all the equations obtained by the XL algorithm are linearly independent, which might not be the case in general. Even with this strong assumption, we see that AIM is secure against the XL attack for all the parameter sets.

Algebraic Attacks on Symmetric Primitives with Large S-box. Several symmetric primitives based on large fields have been proposed with applications to zero-knowledge proof systems such as MiMC [AGR+16], Starkad/Poseidon [GKR+21], and Jarvis [AD18]. Some of them have been analyzed with algebraic attacks exploiting the property that their linear layers are represented as polynomials of low degrees over large fields [ACG+19, EGL+20]. However, AIM uses a randomized linear layer which is expected to have degree 2^{n-1} over $\mathbb{F}_{2^n}$. For this reason, the above attacks would not apply to AIM.

Applicability of Algebraic Attacks on LowMC. LowMC [ARS+15] is the first FHE/MPC-friendly block cipher, and one of its applications is to the Picnic signature scheme. LowMC has been analyzed in the context of the signature scheme, where an adversary is given only a single plaintext-ciphertext pair. In this setting, a number of algebraic attacks have been proposed [BBDV20, BBVY21, LIM21b, Din21, LMSI22, BBCV22], mainly based on two algebraic techniques: linearization by guessing, and the polynomial method [Bei93].

The main idea of linearization-based algebraic attacks on LowMC, first proposed in [BBDV20], is to linearize the underlying S-boxes by guessing a single output bit for each S-box evaluation. In this way, one obtains a system of low-degree polynomial equations at the cost of guessing a small number of bits, and it can be solved efficiently. This linearization technique has been further extended [BBVY21, LIM21b]. However, this type of attacks work only when the underlying S-boxes are of small size. When it comes to AIM, its large S-boxes yield dense implicit equations over $\mathbb{F}_2$, which makes the *guess-and-linearization* infeasible.

The polynomial method [Bei93] has been studied in complexity theory, and later found its application to the design of algorithms for certain problems [Wil14], one of which is to solve a system of polynomial equations over a finite field. The resulting algorithm is known as the first algorithm that achieves exponential speedup over the exhaustive search even in the worst case [LPT+17]. Recently, Dinur [Din21] proposed a generic equation-solving algorithm based on the polynomial method with time complexity $O(n^2 \cdot$

[4]This condition is satisfied by the assumption that all monomials of degrees up to D appear in the extended system, which can be assumed in the case of AIM.

$2^{(1-1/(2.7d))n}$) where n is the number of variables and d is the degree of the system. One arguable issue of this algorithm is its high memory complexity of $O(n^2 \cdot 2^{(1-1/(1.35d))n})$, making it infeasible in practice. For AIM, the memory complexity required by Dinur's algorithm exceeds the security level, i.e., more than 2^λ bits of memory is required for each level of security λ. Table 5 shows the time and the memory complexity of the Dinur's method for each system of AIM. Subsequent works [LMSI22, BBCV22] proposed to reduce the memory complexity of the algorithm at the cost of slightly increased time complexity, while these variants do not apply to AIM since they all follow the guess-and-linearization strategy.

6.3.3 Differential and Linear Cryptanalysis

An adversary is allowed to evaluate AIM with an arbitrary input pair $(\mathsf{pt}, \mathsf{iv})$ in an offline manner. However, such an evaluation is independent of the actual secret key pt^*, so the adversary is not able to collect a sufficient amount of statistical data which are related to pt^*. Furthermore, the linear layer of AIM is generated independently at random for every user. For this reason, we believe that our construction is secure against any type of statistical attacks including (impossible) differential, boomerang, and integral attacks.

In the multi-target scenario, an adversary has no information on which users have the same secret. Even for multiple users with the same iv, statistical attacks would not be feasible since all the inputs and their differences are unknown to the adversary. That said, to prevent any unexpected variant of differential and linear cryptanalysis, we summarize a lower bound of the weight of differential and correlation trails in this section.

DIFFERENTIAL CRYPTANALYSIS. Since AIM is a key-less primitive, we will estimate the security of AIM against differential cryptanalysis by lower bounding the weight of a differential trail (for example, as in [DVA12]).

Given a function $f : \{0,1\}^n \to \{0,1\}^m$, the *weight* of a differential $(\Delta x, \Delta y) \in \{0,1\}^n \times \{0,1\}^m$ is defined by

$$w_d(\Delta x \xrightarrow{f} \Delta y) \stackrel{\text{def}}{=} n - \log_2 |\{x \in \{0,1\}^n : f(x \oplus \Delta x) \oplus f(x) = \Delta y\}| \,.$$

The weight is not defined if there is no x such that $f(x \oplus \Delta x) \oplus f(x) = \Delta y$. Otherwise, we say that Δx and Δy are *compatible*.

A differential trail is the composition of compatible differentials. For AIM, a differential trail from an input to the output (ignoring the feed-forward) can be represented as follows.

$$Q = \Delta_0 \xrightarrow{\mathsf{Mer}[e_1,\ldots,e_\ell]} \Delta_1 \xrightarrow{\mathsf{Lin}} \Delta_2 \xrightarrow{\mathsf{Mer}[e_*]} \Delta_3.$$

Then the weight of the differential trail Q is defined as

$$w_d(Q) \stackrel{\text{def}}{=} \sum_{i=0}^{2} w_d(\Delta_i \to \Delta_{i+1}).$$

The weight of a Mersenne S-box is determined by the number of solutions to $\mathsf{Mer}[e](x \oplus \Delta x) \oplus \mathsf{Mer}[e](x) = \Delta y$, which is a polynomial equation of degree $2^e - 2$. Therefore, there are at most $2^e - 2$ solutions to this equation, which implies for $\Delta x \neq 0$,

$$w_d(\Delta x \xrightarrow{\mathsf{Mer}[e]} \Delta y) \geq n - \log_2(2^e - 2) \geq n - e.$$

Then we have

$$\begin{aligned} w_d(Q) &= \sum_i w_d(\Delta_i \to \Delta_{i+1}) \\ &\geq \max_{1 \leq i \leq \ell} (n - e_i) = n - e_1. \end{aligned}$$

So, for any differential trail Q, $w_d(Q)$ is close to λ with $\lambda = n$. We note that a trail Q such that $w_d(Q) < \lambda$ never incur a collision, and the existence of such trail does not imply the feasibility of differential cryptanalysis since an adversary is not given a large enough number of plaintext-ciphertext pairs to mount the analysis.

DIFFERENCE ENUMERATION ATTACK. Recently, difference enumeration attacks to LowMC have been proposed [RST18, LIM21a, LSW+22], which require only a couple of chosen plaintext-ciphertext pairs. In such attacks, an adversary enumerates all possible input and output differences and tries to find a collision and recover the unknown key. This type of attacks work for LowMC since it is based on small S-boxes. So one can easily find all possible differentials in LowMC. On the other hand, AIM is based on n-bit S-boxes, making it infeasible to enumerate all possible differences of each S-box.

LINEAR CRYPTANALYSIS. In contrast to differential cryptanalysis, security against linear cryptanalysis has been rarely evaluated for key-less primitives since its goal is to retrieve the secret key, not finding a collision or a second-preimage. That said, we lower bound the weight of a correlation trail for completeness in a similar way to differential cryptanalysis.

Given a function $f : \{0,1\}^n \to \{0,1\}^m$, the *weight* of a correlation $(\alpha, \beta) \in \{0,1\}^n \times \{0,1\}^m$ is defined by

$$w_l(\alpha \xrightarrow{f} \beta) \stackrel{\text{def}}{=} n - \log_2 \left| 2 \left| \{x \in \{0,1\}^n : \alpha^\top x = \beta^\top f(x)\} \right| - 2^n \right|.$$

The weight is not defined if there are exactly 2^{n-1} values for x such that $\alpha^\top x = \beta^\top f(x)$. Otherwise, we say that α and β are *compatible*.

A correlation trail is the composition of compatible correlations. For AIM, a correlation trail from an input to the output (ignoring the feed-forward) can be represented as follows.

$$Q = \alpha_0 \xrightarrow{\mathsf{Mer}[e_1,\ldots,e_\ell]} \alpha_1 \xrightarrow{\mathsf{Lin}} \alpha_2 \xrightarrow{\mathsf{Mer}[e_*]} \alpha_3.$$

Then the weight of the correlation trail Q is defined as

$$w_l(Q) \stackrel{\text{def}}{=} \sum_{i=0}^{2} w_l(\alpha_i \to \alpha_{i+1}).$$

When d is not a power-of-2 and $f(x) = x^d$ is invertible over $\mathbb{F}_{2^n}$, one has the following generic bound [KSW19].

$$\left| 2 \left| \{x : \alpha^\top x = \beta^\top f(x)\} \right| - 2^n \right| \le (d-1)2^{n/2}$$

for any compatible correlation (α, β). Therefore the weight of a correlation trail of a Mersenne S-box is lower bounded by $w_l(Q) \ge \frac{n}{2} - e$. Then we have

$$\begin{aligned} w_l(Q) &= \sum_i w_l(\alpha_i \to \alpha_{i+1}) \\ &\ge \max_{1 \le i \le \ell} (n/2 - e_i) + w_l(\alpha_2 \to \alpha_3) \\ &\ge \max_{1 \le i \le \ell} (n/2 - e_i) + (n/2 - e_*) \\ &= n - e_1 - e_*. \end{aligned}$$

As Lin is a (full-rank) compression function, α_2 cannot be the zero mask. Since linear cryptanalysis requires $2^{2w_l(Q)}$ plaintext-ciphertext pairs, AIM would be secure against linear cryptanalysis if

$$2(n - e_1 - e_*) \ge \lambda$$

which is the case for AIM. We emphasize again that linear cryptanalysis is not practically relevant in our setting since AIM does not use any secret key, while all the inputs are kept secret and every user is assigned a distinct linear layer.

6.3.4 Quantum Attacks

Quantum attacks are classified into two types according to the attack model. In the Q1 model, an adversary is allowed to use quantum computation without making any quantum query, while in the Q2 model, both quantum computation and quantum queries are allowed [Zha12].

As a generic algorithm for exhaustive key search, Grover's algorithm has been known to give quadratic speedup compared to the classical brute-force attack [Gro96]. In this section, we investigate if any specialized quantum algorithm targeted at AIM might possibly achieve better efficiency than Grover's algorithm in the Q1 model.

COST OF GROVER'S ALGORITHM. We consider the cost metric of NIST [NIS22], which is defined as the product of the quantum circuit size and the quantum circuit depth with respect to Clifford and T gates.

Given a one-way function f taking n bits as input, the circuit size and the depth of the preimage-finding attack on f using Grover's algorithm is estimated as follows [JBK+22].

$$\text{(Grover's circuit size/depth)} = \text{(size/depth of } f) \times 2 \times \left\lfloor \frac{\pi}{4}\sqrt{2^n} \right\rfloor .$$

The quantum circuit size and the depth of AIM can be computed in a modular manner. AIM is based on three types of operations: finite field multiplication, finite field squaring, and random matrix multiplication. The cost of finite field multiplication is estimated based on the state-of-the-art result of Toffoli-depth one implementation of finite field multiplication [JKL+22], while we ignore the cost of modular reduction in finite field multiplication and finite field squaring since they are far more efficient than other operations [MCT17]. For random matrix multiplication and Toffoli gate decomposition, we refer to the recent implementation of LowMC [JBK+22] and the implementation of [AMMR13] (using 8 Clifford gates and 7 T gates with depth 8), respectively.

Table 6 summarizes the total number of operations and the number of operations executed in serial (depth) for each type of operation where all the S-boxes are implemented with addition chain exponentiation by each shortest chain (see Section 8.1 for the detail). Based on these numbers and the above references, the total cost of Grover's algorithm on AIM is also estimated (in log) for each level of security. We see that AIM-I, AIM-III and AIM-V satisfy the security level I, III and V, respectively.[5] Recently, Jang et al. [JKO+23] analyzed the cost of the Grover's algorithm on AIM-I, and the cost is given as $2^{160.11}$ which implies the security level L1.

Scheme	#Operations, Depth			Total Cost	Level of Security
	FF Mul	FF Square	Mat Mul		
AIM-I	11, 9	32, 30	1, 1	159.79	I (≥157)
AIM-III	14, 11	38, 34	1, 1	225.22	III (≥221)
AIM-V	17, 11	64, 56	1, 1	291.74	V (≥285)

Table 6: The number of operations and the depth for each type of operation used in AIM, and the total cost of Grover's algorithm on AIM for each level of security.

QUANTUM ALGEBRAIC ATTACK. When an algebraic root-finding algorithm works over a small field, the guess-and-determine strategy might be effectively combined with Grover's algorithm, reducing the overall time complexity.

The GroverXL algorithm [BY18] is a quantum version of the FXL algorithm [CKPS00], which solves a system of multivariate quadratic equations over a finite field. A single evaluation of AIM can be represented by Boolean quadratic equations using intermediate variables. Precisely, we have a system of $4(\ell+1)n$ quadratic equations (including field equations) in $(\ell+1)n$ variables. For this system of equations, the time complexity of GroverXL is given as $2^{(0.3496+o(1))(\ell+1)n}$ when using $\omega = 2$, which is worse than Grover's algorithm.

The `QuantumBooleanSolve` algorithm [FHK+17] is a quantum version of the `BooleanSolve` algorithm [BFSS13], which solves a system of Boolean quadratic equations. In [FHK+17], its time complexity has been analyzed only for a system of equations with the same number of variables and equations. A single

[5] In the call for proposals by NIST [NIS22], the security level I, III, V are defined as the strength of AES-128, AES-192, AES-256, respectively, against Grover's algorithm.

evaluation of AIM can be represented by $4(\ell+1)n$ quadratic equations in $(\ell+1)n$ variables. In this case, the complexity of `QuantumBooleanSolve` is given as $O(2^{0.462(\ell+1)n})$, which is worse than Grover's algorithm.

In contrast to the algorithms discussed above, Chen and Gao [CG22] proposed a quantum algorithm to solve a system of multivariate equations using the Harrow-Hassidim-Lloyd (HHL) algorithm [HHL09] that solves a sparse system of linear equations with exponential speedup. In brief, Chen and Gao's algorithm solves a system of linear equations from the Macaulay matrix by the HHL algorithm. It has been claimed that this algorithm enjoys exponential speedup for a certain set of parameters. When applied to AIM, the hamming weight of the secret key should be smaller than $O(\log n)$ to achieve exponential speedup [DGG$^+$21]. Otherwise, this algorithm is slower than Grover's algorithm [DGG$^+$21].

QUANTUM GENERIC ATTACK. A generic attack does not use any particular property of the underlying components (e.g., S-boxes for AIM). The underlying smaller primitives are typically modeled as public random permutations or functions. The Even-Mansour cipher [EM97], the FX-construction [KR01] and a Feistel cipher [LR86] have been analyzed in the classic and generic attack model. As their quantum analogues, the Even-Mansour cipher [KM12, BHNP$^+$19], the FX-construction [LM17, HS18] and a Feistel cipher [KM10] have been analyzed in the Q1 or Q2 model. Most of these attacks can be seen as a combination of Simon's period finding algorithm [Sim97] (in the Q2 model), and Grover's/offline Simon's algorithms [BHNP$^+$19] (in the Q1 model). Since Simon's period finding algorithm requires multiple queries to a *keyed* construction (which is not the case for AIM), we believe that the above attacks do not apply to AIM in a straightforward manner.

6.4 Attacks in the Multi-User Setting

The analysis of the multi-user security of a cryptographic scheme is crucial, as most cryptographic schemes are used by multiple users in practice. In this setting, an adversary is given multiple users' instances (e.g., public keys and corresponding signatures), and it aims to attack one of them.

MULTI-USER EUF-CMA SECURITY. Since the EUF-CMA security is a fundamental requirement for digital signatures, it is natural to consider Multi-User EUF-CMA (MU-EUF-CMA) security in the multi-user setting. Here, the adversary is given multiple signing oracles (corresponding to distinct public keys), and tries to generate a valid forgery under one of given public keys through a chosen message attack. Thanks to the generic reduction from EUF-CMA security to MU-EUF-CMA security [GMLS02], AIMer provides MU-EUF-CMA security with losses that are (at most) linear in the number of users. In addition, the concrete design of AIMer takes into account multi-user attacks, or more generally, *multi-target attacks*.

MULTI-TARGET ATTACKS. In the multi-target attack, an adversary is given a multiple number of targets, for example, the outputs of a cryptosystem computed with different secret keys. This is inherently possible in the multi-user setting, and even in a single-user setting, when multiple targets are available to the adversary.

There are many examples of successful multi-target attacks. In [DN19], Dinur and Nadler proposed an effective multi-target attack on Picnic version 1.0. The main idea is that an attacker collects multiple outputs generated from unknown seeds of the unopened party in the MPCitH protocol, compares them to the outputs from guessed ones, trying to find a collision using a certain efficient algorithm such as hash tables to recover the seed of the unopened party. Once the seed is revealed, the secret key is also recovered from its additive shares. The above attack is mitigated in the next version of the Picnic signature by using a random salt and domain seperation prefixes as an additional input of underlying hash functions and XOFs.

Multi-target attacks have also been proposed on hash-based signature schemes [BXKSN21, YAG21]. As many hash outputs are used as secret keys of the underlying one-time signature (OTS), the seed guessing technique also works in hash-based signatures, and the recovered seed reveals the corresponding secret keys. It can be mitigated by domain separation of the hash functions according to the position of the OTS instances. Another multi-target attack on SPHINCS$^+$ of the L5 parameter set exploits the small state size of SHA-256 [PKC22], but it is not applicable when SHAKE256 is used as the underlying hash function.

When it comes to AIMer, the use of iv mitigates multi-target attacks. AIM generates its linear layer from a random iv, so not only each user has a different secret key (i.e., the input of AIM), but also the functions themselves are all different. Moreover, similarly to the mitigation techniques described above, a random 2λ-bit salt is used, and domain separation is applied to each hash function and the XOF used in the signature. It would prevent any type of efficient multi-target preimage search attack, such as time/memory/data trade-off attacks [BS00] and parallel quantum multi-target preimage attacks [BB18]. We refer to Section 7.2 for detailed specifications of the hash functions.

KEY SUBSTITUTION ATTACKS. In a key substitution attack (KSA), an adversary is given a signature σ_A under a public key pk_A. Then the adversary tries to produce a fake public key pk_E such that σ_A is also a valid signature under pk_E. This type of attacks were first considered in [BWM99], under the name *unknown key-share attacks*, and later formalized in [MS04]. Although the possibility of KSA does not violate the MU-EUF-CMA security, it may need to be considered in practical applications of digital signatures, in particular, when non-repudation property is required [KM13]. Fortunately, the security against KSAs can be achieved in the generic way using the following theorem.

Theorem 3 (Theorem 6 in [MS04]). *Let* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *be an EUF-CMA secure digital signature scheme. Then,* $\Pi' = (\mathsf{Gen}, \mathsf{Sign}', \mathsf{Verify})$ *is a secure digital signature scheme against KSAs with*

$$\mathsf{Sign}' = \mathsf{Sign}(\mathsf{sk}, \mathsf{Encode}(\mathsf{pk}, m))$$

where Encode *is an unambiguous encoding scheme of public keys and messages.*

In AIMer, a (fixed length) public key is always appended to the message before hashing, so we believe that AIMer is secure against KSAs.

6.5 Side-Channel Attacks

The key generation of AIMer is executed in constant time. The signing algorithm is not executed in constant time while the timing difference originates only from public information. When N is not a power-of-two, the time that it takes to construct seeds_k in Algorithm 1 (Line 33) depends on the undisclosed index $\bar{i}_k$ which is public information. Therefore, we conclude that the secret information of AIMer does not affect the running time of AIMer.

Many masking techniques to thwart side-channel attacks follow the form of secret-sharing [ISW03, BBP+17, KR19]. As AIMer generates a signature by simulating secret-shared computation of an one-way function, it seemingly provides a natural mitigation to some side-channel attacks. Nevertheless, we expect that AIMer will be vulnerable to power [KJJ99] attacks, electromagnetic radiation (EM) attacks [QS01] and fault-injection attacks [BDL97] without any protection in its implementation. Recently, machine learning has also been combined with a number of existing side-channel attacks on conventional/post-quantum encryption schemes [DGD+19, WD20, DNG22]. We will prepare appropriate countermeasures against these attacks in the future.

7 Specification of the AIMer Signature Scheme

7.1 Field Representation

In AIM, fields $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$ are used for AIM-I, AIM-III, and AIM-V, respectively. Each field is defined by $\mathbb{F}_2[X]/(f(X))$ with a primitive polynomial $f(X)$. The primitive polynomials of low weights have been chosen for efficient implementation as follows.

- AIM-I: $f(X) = X^{128} + X^7 + X^2 + X + 1$,
- AIM-III: $f(X) = X^{192} + X^7 + X^2 + X + 1$,
- AIM-V: $f(X) = X^{256} + X^{10} + X^5 + X^2 + 1$.

Prefix	Description	Functions
0x01	Computing challenge hash in phase 2.	h_1_commitment
0x02	Computing challenge hash in phase 4.	h_2_commitment
0x03	Computing parties' seeds as binary tree leaves.	expand_seed, expand_seed_x4
0x04	Committing to parties' seeds and generating tapes.	commit_to_seed_and_expand_tape, commit_to_seed_and_expand_tape_x4
-	Expanding hash in phase 2.	h_1_expand
-	Expanding hash in phase 4.	h_2_expand
-	Generating affine layer.	generate_matrices_L_and_U

Table 7: The prefix of each types of input in SHAKE.

Let $\mathtt{x}[\mathtt{i}]$ denote the i-th coefficient bit of $x \in \mathbb{F}_{2^n}$ for $i = 0, \dots, n-1$, where the most (resp. least) significant bit is $\mathtt{x}[0]$ (resp. $\mathtt{x}[\mathtt{n}-1]$). A field element is also written in hexadecimal format. For example, we will write a hexadecimal number $\mathtt{0xA0}\underbrace{\mathtt{0}\dots\mathtt{0}}_{28}\mathtt{01}$ in $\mathbb{F}_{2^{128}}$ to denote $x \in \mathbb{F}_{2^{128}}$ such that $\mathtt{x}[127] = \mathtt{x}[125] = \mathtt{x}[0] = 1$ and $\mathtt{x}[\mathtt{i}] = 0$ for the other indices i, which corresponds to $X^{127} + X^{125} + 1$ as a polynomial.

7.2 Hash Functions and Extendable-Output Functions

All the hash functions in AIMer are based on SHAKE128 or SHAKE256 [NIS15]. We use SHAKE128 with 256-bit outputs for $n = 128$ and SHAKE256 with 384 and 512-bit outputs for $n = 192, 256$, respectively. As SHAKE supports arbitrary length of outputs, the extendable-output functions (XOFs) are also based on SHAKE. We use SHAKE128 for $n = 128$ and SHAKE256 for $n = 192, 256$ in a similar manner to hash functions. We summarize all the hash functions and XOFs in Table 7.

As the SHAKE implementation supports parallel execution of four instances, we computed them in batches of four if it is possible to compute the hashes in parallel, such as expanding parties' seeds as binary tree leaves (expand_seed_x4) and committing to parties' seeds and generating tapes (commit_to_seed_and_expand_tape_x4).

When the SHAKE hash functions are used for different types of input, we separated the hash functions by adding 1-byte prefix in each input to prevent hash collisions, except functions h_1_expand, h_2_expand, and generate_matrices_L_and_U.

The function h_1_expand (resp. h_2_expand) is a function expanding hash values in Phase 2 (resp. Phase 4). The domain separation is not applied to h_1_expand (resp. h_2_expand) because the input of the hash function is a hash digest derived from h_1_commitment (resp. h_2_commitment), which is already hashed by a domain-separated function.

In the function generate_matrices_L_and_U, the input to the hash function is an iv of size n. On the other hand, the input to the other hash functions is at least $3n$ bits or longer. Therefore the input to the hash function in the generate_matrices_L_and_U function would not collide with any input to the hash functions used in the other functions. For this reason, domain separation is not applied to the generate_matrices_L_and_U function.

The functions hash_init and hash_init_x4 are used for hash functions with no prefix, and the functions hash_init_prefix and hash_init_prefix_x4 are used for hash functions with 1-byte prefix.

7.3 Key Generation

Key generation is executed via a function aimer_keygen. First, the input of AIM pt, and the initial vector iv are randomly chosen. Then the outputs of AIM ct is determined by $\mathsf{ct} = \mathsf{AIM}(\mathsf{iv}, \mathsf{pt})$. The input pt is the secret key of AIMer and $(\mathsf{iv}, \mathsf{ct})$ is the corresponding public key.

The affine layer in AIM consists of an $n \times \ell n$ binary matrix $\mathtt{A}$ and a vector $\mathtt{b}$ of size n, derived from iv. The matrix $\mathtt{A} = [\mathtt{A}_1 | \dots | \mathtt{A}_\ell]$ is composed of ℓ invertible matrices $\mathtt{A}_i$. Each invertible matrix $\mathtt{A}_i = \mathtt{L}_i \times \mathtt{U}_i$ is obtained by multiplying an $n \times n$ lower triangular matrix $\mathtt{L}_i$ and an $n \times n$ upper triangular matrix $\mathtt{U}_i$ where an lower triangular matrix L (resp. upper triangular matrix U) is a square matrix in which all the entries above (resp. below) the main diagonal are zero. In matrix L (resp. U), we say that $\mathtt{L}[\mathtt{i}][\mathtt{j}]$ (resp. $\mathtt{U}[\mathtt{i}][\mathtt{j}]$) is a *fixed bit* if $\mathtt{i} \leq \mathtt{j}$ (resp. $\mathtt{i} \geq \mathtt{j}$) and $\mathtt{L}[\mathtt{i}][\mathtt{j}]$ (resp. $\mathtt{U}[\mathtt{i}][\mathtt{j}]$) is a *free bit* if $\mathtt{i} > \mathtt{j}$ (resp. $\mathtt{i} < \mathtt{j}$). In summary, L is of the form

$$\mathtt{L}[\mathtt{i}][\mathtt{j}] = \begin{cases} 1 & \text{if } \mathtt{i} = \mathtt{j}, \\ 0 & \text{if } \mathtt{i} < \mathtt{j}, \\ 0 \text{ or } 1 & \text{if } \mathtt{i} > \mathtt{j}, \end{cases}$$

and U is of the form

$$\mathtt{U}[\mathtt{i}][\mathtt{j}] = \begin{cases} 1 & \text{if } \mathtt{i} = \mathtt{j}, \\ 0 \text{ or } 1 & \text{if } \mathtt{i} < \mathtt{j}, \\ 0 & \text{if } \mathtt{i} > \mathtt{j}. \end{cases}$$

XOF is initialized with the initial vector iv, and the *free bits* in each matrices and the vector b are determined by the outputs of the XOF. The procedure to generate the lower triangular matrix L and the upper triangular matrix U is described in Algorithm 6 and 7, respectively. In the pseudocodes, $\mathtt{XOF.squeeze}(t)$ denotes a t-byte sequence squeezed from the XOF. After generating $\mathtt{L}_1, \mathtt{U}_1, \ldots, \mathtt{L}_\ell, \mathtt{U}_\ell$ in the order as presented, b is generated by $\mathtt{b} \leftarrow \mathtt{XOF.squeeze}(n/8)$ in little endian order. Note that XOF is stateful, as it is initialized by iv and maintains its state throughout the generation of the matrices and the vector.

The matrices and the vector are generated from the function generate_matrices_L_and_U. In this function, matrix_A corresponds to $\mathtt{L}_1, \mathtt{U}_1, \ldots, \mathtt{L}_\ell, \mathtt{U}_\ell$. Since each matrix $\mathtt{A_i}$ is multiplied by the vector $\mathsf{Mer}[e_i](\mathsf{pt})$ only once during key generation, $\mathtt{A_i} \cdot \mathsf{Mer}[e_i](\mathsf{pt})$ is computed in the order of $\mathtt{L_i} \cdot (\mathtt{U_i} \cdot \mathsf{Mer}[e_i](\mathsf{pt}))$.

Algorithm 6: Algorithm to generate the lower triangular matrix L:

---

for *each* $\mathtt{i} \in [n]$ **do**
    for *each* $\mathtt{j} \in [n]$ **do**
        if $\mathtt{i} < \mathtt{j}$ **then**
            $\mathtt{L}[\mathtt{i}][\mathtt{j}] = 0$.
        if $\mathtt{i} = \mathtt{j}$ **then**
            $\mathtt{L}[\mathtt{i}][\mathtt{j}] = 1$.
for *each* $\mathtt{j} \in [n]$ **do**
    for *each* $\mathtt{i} \in [n/8]$ **do**
        if *There is a free bit in* $\mathtt{L}[8\mathtt{i} : 8\mathtt{i} + 7][\mathtt{j}]$ **then**
            $\mathtt{x} \leftarrow \mathtt{XOF.squeeze}(1)$.
            for *each* $\mathtt{t} \in [8]$ **do**
                if $\mathtt{L}[8\mathtt{i} + \mathtt{t}][\mathtt{j}]$ *is a free bit* **then**
                    $\mathtt{L}[8\mathtt{i} + \mathtt{t}][\mathtt{j}] \leftarrow (\mathtt{t} + 1)$-th least significant bit in x.

7.4 Signature Generation

Input: Signer's key pair $(pk = (\mathtt{iv}, \mathtt{ct}), sk = \mathtt{pt})$, msg as a byte array to be signed.

Output: Signature σ on msg as a byte array.

// Phase 1

Algorithm 7: Algorithm to generate the upper triangular matrix U:

```
for each i ∈ [n] do
    for each j ∈ [n] do
        if i > j then
            U[i][j] = 0.
        if i = j then
            U[i][j] = 1.
for each j ∈ [n] do
    for each i ∈ [n/8] do
        if There is a free bit in U[8i : 8i + 7][j] then
            x ← XOF.squeeze(1).
            for each t ∈ [8] do
                if U[8i + t][j] is a free bit then
                    U[8i + t][j] ← (t + 1)-th least significant bit in x.
```

1. Declare a list of commitments $\texttt{party\_seed\_commitments}[\tau][N]$ (byte arrays, each of length $2n$ bits), a list of random tapes $\texttt{random\_tapes}[\tau][N]$ (each of length $n + \ell n + n + n$ bits), and a $2n$-bit value salt.

2. Generate the affine layer of AIM; $(\texttt{matrix\_A}, \texttt{vector\_b}) \leftarrow \texttt{generate\_matrix\_LU}(\texttt{iv})$.

3. Compute outputs of the first ℓ S-boxes of AIM; $\texttt{sbox\_outputs}[\ell] \leftarrow \texttt{compute\_sbox\_outputs}(\texttt{pt})$.

4. Sample a $2n$-bit random salt $\texttt{salt}$.

5. For each parallel repetition $\texttt{k}$ from 0 to $\tau - 1$:

   (a) Sample a n-bit random $\texttt{master\_seed}$.

   (b) Generate seeds of the parties from the master seed;

   $$\texttt{seed\_trees}[\texttt{k}] \leftarrow \texttt{make\_seed\_tree}(\texttt{master\_seed}, \texttt{salt}, N, \texttt{k})$$

   (c) For each party $\texttt{i}$ from 0 to $N - 1$, commit to the party's seed and expand tape after committing;

   $$\begin{aligned}&(\texttt{party\_seed\_commitments}[\texttt{k}][\texttt{i}], \texttt{random\_tapes}[\texttt{k}][\texttt{i}])\\ &\quad \leftarrow \texttt{commit\_to\_seed\_and\_expand\_tape}(\texttt{get\_leaf}(\texttt{seed\_trees}[\texttt{k}], \texttt{salt}, \texttt{k}, \texttt{i})).\end{aligned}$$

6. Declare lists of finite field elements $\texttt{shared\_x}[\tau][N][\ell + 1]$, $\texttt{shared\_z}[\tau][N][\ell + 1]$, $\texttt{shared\_t}[\tau][N][\ell]$, $\texttt{shared\_dot\_a}[\tau][N]$, $\texttt{shared\_dot\_c}[\tau][N]$, and a list of byte arrays $\texttt{shared\_pt}[\tau][N]$ (each of length n bits).

7. For each parallel repetition $\texttt{k}$ from 0 to $\tau - 1$:

   (a) Zero-initialize the adjusting value $\Delta\texttt{pt}$; $\texttt{proof}[\texttt{k}].\texttt{pt\_delta} \leftarrow 0$.

   (b) For each party $\texttt{i}$ from 0 to $N - 1$:

      i. Sample the tape; $\texttt{shared\_pt}[\texttt{k}][\texttt{i}] \leftarrow \texttt{random\_tapes}[\texttt{k}][\texttt{i}]_{[0:n-1]}$.

      ii. $\texttt{proof}[\texttt{k}].\texttt{pt\_delta} \leftarrow \texttt{proof}[\texttt{k}].\texttt{pt\_delta} \oplus \texttt{shared\_pt}[\texttt{k}][\texttt{i}]$.

   (c) Compute the difference; $\texttt{proof}[\texttt{k}].\texttt{pt\_delta} \leftarrow \texttt{proof}[\texttt{k}].\texttt{pt\_delta} \oplus \texttt{pt}$.

   (d) Adjust the first share; $\texttt{first\_pt\_share} \leftarrow \texttt{shared\_pt}[\texttt{k}][0] \oplus \texttt{proof}[\texttt{k}].\texttt{pt\_delta}$.

(e) Zero-initialize the adjusting values Δz, which are $\texttt{proof}[\texttt{k}].\texttt{z\_delta}[0], \ldots, \texttt{proof}[\texttt{k}].\texttt{z\_delta}[\ell-1]$.

(f) For each party i from 0 to $N-1$:

i. For each AIM S-Box index j from 0 to $\ell-1$:

A. Sample the tape; $\texttt{shared\_t}[\texttt{k}][\texttt{i}][\texttt{j}] \leftarrow \texttt{random\_tapes}[\texttt{k}][\texttt{i}]_{[\texttt{jn}:(\texttt{j}+1)\texttt{n}-1]}$.

B. $\texttt{proof}[\texttt{k}].\texttt{z\_delta}[\texttt{j}] \leftarrow \texttt{proof}[\texttt{k}].\texttt{z\_delta}[\texttt{j}] \oplus \texttt{shared\_t}[\texttt{k}][\texttt{i}][\texttt{j}]$.

(g) For each AIM S-Box index j from 0 to $\ell-1$:

i. $\texttt{proof}[\texttt{k}].\texttt{z\_delta}[\texttt{j}] \leftarrow \texttt{proof}[\texttt{k}].\texttt{z\_delta}[\texttt{j}] \oplus \texttt{sbox\_outputs}[\texttt{j}]$.

ii. Adjust the first shares; $\texttt{shared\_t}[\texttt{k}][0][\texttt{j}] \leftarrow \texttt{shared\_t}[\texttt{k}][0][\texttt{j}] \oplus \texttt{proof}[\texttt{k}].\texttt{z\_delta}[\texttt{j}]$.

(h) Compute MPC multiplication triples described in Section 4.2, Step 3 of Phase 1; $(\texttt{shared\_z}[\texttt{k}], \texttt{shared\_x}[\texttt{k}]) \leftarrow \texttt{aim\_mpc}(\texttt{shared\_pt}[\texttt{k}], \texttt{ct}, \texttt{matrix\_A}, \texttt{vector\_b}, \texttt{shared\_t}[\texttt{k}])$. The details is in Section 7.6.5.

8. For each parallel repetition k from 0 to $\tau-1$:

(a) Initialize a field element a as zero.

(b) Zero-initialize the adjusting value Δc; $\texttt{proof}[\texttt{k}].\texttt{c\_delta} \leftarrow 0$.

(c) For each party i from 0 to $N-1$:

i. $\texttt{a\_share} \leftarrow \texttt{a\_share} \oplus \texttt{random\_tapes}[\texttt{k}][\texttt{i}]_{[(\ell+1)\texttt{n}:(\ell+2)\texttt{n}-1]}$.

ii. $\texttt{shared\_dot\_c}[\texttt{k}][\texttt{i}] \leftarrow \texttt{random\_tapes}[\texttt{k}][\texttt{i}]_{[(\ell+2)\texttt{n}:(\ell+3)\texttt{n}-1]}$.

iii. $\texttt{proof}[\texttt{k}].\texttt{c\_delta} \leftarrow \texttt{proof}[\texttt{k}].\texttt{c\_delta} \oplus \texttt{shared\_dot\_c}[\texttt{k}][\texttt{i}]$.

(d) $\texttt{a} \leftarrow \texttt{a} \times \texttt{pt}$.

(e) $\texttt{proof}[\texttt{k}].\texttt{c\_delta} \leftarrow \texttt{a} \oplus \texttt{proof}[\texttt{k}].\texttt{c\_delta}$.

(f) $\texttt{shared\_dot\_c}[\texttt{k}][0] \leftarrow \texttt{shared\_dot\_c}[\texttt{k}][0] \oplus \texttt{proof}[\texttt{k}].\texttt{c\_delta}$.

// Phase 2

9. Declare a list of challenge values $\texttt{epsilons}[\tau][\ell+1]$.

10. Compute the first challenge hash; $\texttt{h\_1} \leftarrow \texttt{h\_1\_commitment}()$ as described in Section 7.6.3 with input

$$\texttt{0x01} \parallel \texttt{msg} \parallel pk \parallel \texttt{salt} \parallel \texttt{party\_seed\_commitments} \parallel \texttt{proof.pt\_delta} \parallel \texttt{proof.z\_delta} \parallel \texttt{proof.c\_delta}.$$

11. Expand the challenge from the hash; $\texttt{epsilons} \leftarrow \texttt{h\_1\_expand}(\texttt{h\_1})$.

// Phase 3

12. Declare lists of field elements $\texttt{alpha\_shares}[\tau][N]$ and $\texttt{v\_shares}[\tau][N]$.

13. For each parallel repetition k from 0 to $\tau-1$:

(a) Initialize a field element alpha as zero.

(b) For each party i from 0 to $N-1$:

i. $\texttt{alpha\_shares}[\texttt{k}][\texttt{i}] \leftarrow \texttt{shared\_x}[\texttt{k}][\texttt{i}][0] \times \texttt{epsilons}[\texttt{k}][0] \oplus \texttt{shared\_dot\_a}[\texttt{k}][\texttt{i}]$ using the multiplication over $\mathbb{F}_{2^\lambda}$.

ii. For each AIM S-Box index j from 1 to ℓ, construct the shares $\alpha_k^{(i)}$:

A. $\texttt{alpha\_shares}[\texttt{k}][\texttt{i}] \leftarrow \texttt{shared\_x}[\texttt{k}][\texttt{i}][\texttt{j}] \times \texttt{epsilons}[\texttt{k}][\texttt{j}] \oplus \texttt{alpha\_shares}[\texttt{k}][\texttt{i}]$.

iii. $\texttt{alpha} \leftarrow \texttt{alpha} \oplus \texttt{alpha\_shares}[\texttt{k}][\texttt{i}]$.

(c) For each party i from 0 to $N-1$, compute the multiplication-checking protocol:

i. $\texttt{v\_shares}[\texttt{k}][\texttt{i}] \leftarrow \texttt{alpha} \times \texttt{shared\_pt}[\texttt{k}][\texttt{i}] \oplus \texttt{shared\_dot\_c}[\texttt{k}][\texttt{i}]$.

ii. For each AIM S-Box index j from 0 to ℓ:
   A. $\texttt{v\_shares}[\texttt{k}][\texttt{i}] \leftarrow \texttt{epsilons}[\texttt{k}][\texttt{j}] \times \texttt{shared\_z}[\texttt{k}][\texttt{i}][\texttt{j}] \oplus \texttt{v\_shares}[\texttt{k}][\texttt{i}]$.

// Phase 4

14. Compute the second challenge hash; $\texttt{h\_2} \leftarrow \texttt{h\_2\_commitment}()$ as described in Section 7.6.3 with input

$$\texttt{0x02} \parallel \texttt{salt} \parallel \texttt{h\_1} \parallel (\texttt{alpha\_shares, v\_shares}).$$

15. Expand the challenge from the hash; $\texttt{missing\_parties}[\tau] \leftarrow \texttt{h\_2\_expand}(\texttt{h\_2})$.

// Phase 5

16. For each parallel repetition k from 0 to $\tau - 1$, reveal the view of the disclosed parties and the commitment of the undisclosed party:
    (a) $\texttt{proof}[\texttt{k}].\texttt{reveal\_list} \leftarrow \texttt{reveal\_all\_but}(\texttt{seed\_trees}[\texttt{k}], \texttt{missing\_parties}[\texttt{k}])$.
    (b) $\texttt{proof}[\texttt{k}].\texttt{missing\_commitment} \leftarrow \texttt{party\_seed\_commitments}[\texttt{missing\_parties}[\texttt{k}]]$.
    (c) $\texttt{proof}[\texttt{k}].\texttt{missing\_alpha\_share} \leftarrow \texttt{alpha\_shares}[\texttt{k}][\texttt{missing\_parties}[\texttt{k}]]$.
17. Serialize $(\texttt{salt}, \texttt{h\_1}, \texttt{h\_2}, \texttt{proof})$ as described in Section 7.6.7 and output it as the signature.

7.5 Signature Verification

Input: Signer's public key $pk = (\texttt{iv}, \texttt{ct})$, a message msg as a byte array, a signature σ as a byte array.

Output: Accept if σ is a valid signature of msg with respect to $(\texttt{iv}, \texttt{ct})$ or Reject otherwise.

1. Deserialize the signature σ to $(\texttt{salt}, \texttt{h\_1}, \texttt{h\_2}, \texttt{proof}[\tau])$ and derive the challenge indices $\texttt{missing\_parties}[\tau]$ as described in Section 7.6.6, where proof consists of $(\texttt{reveal\_list}, \texttt{missing\_commitment}, \texttt{pt\_delta}, \texttt{c\_delta}, \texttt{z\_delta}[0 : \ell - 1], \texttt{missing\_alpha\_share})$. If deserialization fails, reject the signature and output Reject.
2. Generate the affine layer of AIM; $(\texttt{matrix\_A}, \texttt{vector\_b}) \leftarrow \texttt{generate\_matrix\_LU}(\texttt{iv})$.
3. Expand the first challenge $\texttt{epsilons} \leftarrow \texttt{h\_1\_expand}(\texttt{h\_1})$ as described in Section 7.6.4.
4. For each parallel repetition k from 0 to $\tau - 1$:
    (a) Reconstruct the seed tree as described in Section 7.6.1;
        $\texttt{seed\_trees}[\texttt{k}] \leftarrow \texttt{reconstruct\_seed\_tree}(\texttt{reveal\_list}, \texttt{salt}, N, \texttt{k})$,
        where reveal_list is included in the $\texttt{proof}[\texttt{k}]$.
    (b) For each party i from 0 to $N - 1$:
        i. If $\texttt{i} \neq \texttt{missing\_parties}[\texttt{k}]$, recompute the commitment and the tapes;
           $(\texttt{party\_seed\_commitments}[\texttt{k}][\texttt{i}], \texttt{random\_tapes}[\texttt{k}][\texttt{i}]) \leftarrow$
           $\texttt{commit\_to\_seed\_and\_expand\_tape}(\texttt{get\_leaf}(\texttt{seed\_trees}[\texttt{k}], \texttt{salt}, \texttt{k}, \texttt{i}))$.
        ii. If $\texttt{i} = \texttt{missing\_parties}[\texttt{k}]$, move the missing commitment to party_seed_commitments;
           $\texttt{party\_seed\_commitments}[\texttt{k}][\texttt{i}] \leftarrow \texttt{missing\_commitment}$,
           where missing_commitment is included in the $\texttt{proof}[\texttt{k}]$.
5. Declare lists of field elements $\texttt{shared\_x}[\tau][N][\ell + 1]$, $\texttt{shared\_z}[\tau][N][\ell + 1]$, $\texttt{shared\_t}[\tau][N][\ell]$, $\texttt{shared\_dot\_a}[\tau][N]$, $\texttt{shared\_dot\_c}[\tau][N]$, a list of byte array $\texttt{shared\_pt}[\tau][N]$.
6. For each parallel repetition k from 0 to $\tau - 1$:

(a) For each party $\mathtt{i}$ from 0 to $N-1$:

   i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$, sample the tapes; $\mathtt{shared\_pt}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{random\_tapes}[\mathtt{k}][\mathtt{i}]_{[0:\mathtt{n}-1]}$.

(b) Adjust the first share of pt; $\mathtt{shared\_pt}[\mathtt{k}][0] \leftarrow \mathtt{shared\_pt}[\mathtt{k}][0] \oplus \mathtt{pt\_delta}$, where pt_delta is included in the $\mathtt{proof}[\mathtt{k}]$.

(c) For each party $\mathtt{i}$ from 0 to $N-1$:

   i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$:

      A. For each AIM S-Box index $\mathtt{j}$ from 0 to $\ell-1$, Sample the tapes; $\mathtt{shared\_t}[\mathtt{k}][\mathtt{i}][\mathtt{j}] \leftarrow \mathtt{random\_tapes}[\mathtt{k}][\mathtt{i}]_{[(\mathtt{j}+1)\mathtt{n}:(\mathtt{j}+2)\mathtt{n}-1]}$.

(d) For each AIM S-Box index $\mathtt{j}$ from 0 to $\ell-1$:

   i. Adjust the first share; $\mathtt{shared\_t}[\mathtt{k}][0][\mathtt{j}] \leftarrow \mathtt{shared\_t}[\mathtt{k}][0][\mathtt{j}] \oplus \mathtt{z\_delta}[\mathtt{j}]$, where $\mathtt{z\_delta}[\mathtt{j}]$ is included in the $\mathtt{proof}[\mathtt{k}]$.

(e) Recompute the multiplication triples; $(\mathtt{shared\_z}[\mathtt{k}], \mathtt{shared\_x}[\mathtt{k}]) \leftarrow \mathtt{aim\_mpc}(\mathtt{shared\_pt}[\mathtt{k}], \mathtt{ct}, \mathtt{matrix\_A}, \mathtt{vector\_b}, \mathtt{shared\_t}[\mathtt{k}])$.

7. For each parallel repetition $\mathtt{k}$ from 0 to $\tau-1$:

   (a) For each party $\mathtt{i}$ from 0 to $N-1$:

      i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$:

         A. Sample the tapes; $\mathtt{shared\_dot\_a}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{random\_tapes}[\mathtt{k}][\mathtt{i}]_{[(\ell+1)\mathtt{n}:(\ell+2)\mathtt{n}-1]}$.

         B. Sample the tapes; $\mathtt{shared\_dot\_c}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{random\_tapes}[\mathtt{k}][\mathtt{i}]_{[(\ell+2)\mathtt{n}:(\ell+3)\mathtt{n}-1]}$.

   (b) If $\mathtt{missing\_parties}[\mathtt{k}] \neq 0$, adjust the first share; $\mathtt{shared\_dot\_c}[\mathtt{k}][0] \leftarrow \mathtt{shared\_dot\_c}[\mathtt{k}][0] \oplus \mathtt{c\_delta}$, where c_delta is provided in the $\mathtt{proof}[\mathtt{k}]$.

8. Declare lists of field elements $\mathtt{alpha\_shares}[\tau][N]$ and $\mathtt{v\_shares}[\tau][N]$.

9. For each parallel repetition $\mathtt{k}$ from 0 to $\tau-1$:

   (a) For each party $\mathtt{i}$ from 0 to $N-1$:

      i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$:

         A. Initialize a field element alpha as zero.

         B. Recompute $\mathtt{alpha\_shares}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{shared\_x}[\mathtt{k}][\mathtt{i}][0] \times \mathtt{epsilons}[\mathtt{k}][0] \oplus \mathtt{shared\_dot\_a}[\mathtt{k}][\mathtt{i}]$.

         C. For each AIM S-Box index $\mathtt{j}$ from 1 to ℓ, recompute $\mathtt{alpha\_shares}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{shared\_x}[\mathtt{k}][\mathtt{i}][\mathtt{j}] \times \mathtt{epsilons}[\mathtt{k}][\mathtt{j}] \oplus \mathtt{alpha\_shares}[\mathtt{k}][\mathtt{i}]$.

         D. $\mathtt{alpha} \leftarrow \mathtt{alpha} \oplus \mathtt{alpha\_shares}[\mathtt{k}][\mathtt{i}]$.

   (b) Compute $\mathtt{alpha} \leftarrow \mathtt{alpha} \oplus \mathtt{missing\_alpha\_share}$, where missing_alpha_share is included in the $\mathtt{proof}[\mathtt{k}]$.

   (c) For each party $\mathtt{i}$ from 0 to $N-1$, recompute the multiplication-checking protocol:

      i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$:

         A. Recompute $\mathtt{v\_shares}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{alpha} \times \mathtt{shared\_pt}[\mathtt{k}][\mathtt{i}] \oplus \mathtt{shared\_dot\_c}[\mathtt{k}][\mathtt{i}]$.

         B. For each AIM S-Box index $\mathtt{j}$ from 0 to ℓ, recompute $\mathtt{v\_shares}[\mathtt{k}][\mathtt{i}] \leftarrow \mathtt{epsilons}[\mathtt{k}][\mathtt{j}] \times \mathtt{shared\_z}[\mathtt{k}][\mathtt{i}][\mathtt{j}] \oplus \mathtt{v\_shares}[\mathtt{k}][\mathtt{i}]$.

   (d) For each party $\mathtt{i}$ from 0 to $N-1$:

      i. If $\mathtt{i} \neq \mathtt{missing\_parties}[\mathtt{k}]$:

         A. Compute the v-share of the missing party; $\mathtt{v\_shares}[\mathtt{k}][\mathtt{missing\_parties}[\mathtt{k}]] \leftarrow \mathtt{v\_shares}[\mathtt{k}][\mathtt{missing\_parties}[\mathtt{k}]] \oplus \mathtt{v\_shares}[\mathtt{k}][\mathtt{i}]$.

10. Recompute the first challenge hash; $\texttt{h\_1\_prime} \leftarrow \texttt{h\_1\_commitment}()$ as described in Section 7.6.3 with input

$$\texttt{0x01} \parallel \texttt{msg} \parallel pk \parallel \texttt{salt} \parallel \texttt{party\_seed\_commitments} \parallel \texttt{proof.pt\_delta} \parallel \texttt{proof.z\_delta} \parallel \texttt{proof.c\_delta}.$$

11. Recompute the second challenge hash; $\texttt{h\_2\_prime} \leftarrow \texttt{h\_2\_commitment}()$ as described in Section 7.6.3 with input

$$\texttt{0x02} \parallel \texttt{salt} \parallel \texttt{h\_1} \parallel (\texttt{alpha\_shares, v\_shares}).$$

12. Compare `h_1` to `h_1_prime` and `h_2` to `h_2_prime`, respectively. If they match, σ is a valid signature; return `Accept`, otherwise return `Reject`.

7.6 Supporting Functions

7.6.1 Seed Trees: make_seed_tree, reveal_all_but, reconstruct_seed_tree

In this section, we describe some functions to compute or reconstruct the seed tree. The total number of nodes `num_nodes` is $2^d - 1 + N$, where $d = \lceil \log(N) \rceil$. The seed tree consists of three arrays of `num_nodes` elements: `data` with n-bit data, `have_value` and `exists` with integers, i.e.,

$$\texttt{seed\_tree} = (\texttt{data}, \texttt{have\_value}, \texttt{exists}).$$

Integer arrays `have_value` and `exists` are all initially set to zero. The root is the node of index 0 (`data[0]`). When the index of a parent node is `i`, the index of the left (resp. right) child node is $2\texttt{i} + 1$ (resp. $2\texttt{i} + 2$).

PROCESS OF `make_seed_tree`. The function `make_seed_tree` generates the seed of each party using binary tree structure. Hash function `expand_seed` outputting a $2n$-bit digest is used to make two children nodes from a parent node. The leaf nodes are only used for seeds. The details are described as follows.

Input: `master_seed, salt, repetition_index`.

Output: a (almost) complete binary tree `seed_tree` having N leftmost leaves with `master_seed` as the root node.

1. Set elements in `exists` corresponding to leaf nodes to be 1.
2. For each of non-leaf nodes, set elements in `exists` to be 1 if the corresponding nodes have at least one child.
3. Set $\texttt{data}[0] \leftarrow \texttt{master\_seed}$, and initialize $\texttt{parent\_node} \leftarrow \texttt{data[0]}$.
4. For the index `parent_node_index` from 0 to $(\texttt{num\_nodes} - N)$, if $\texttt{exists}[\texttt{parent\_node\_index}] = 1$, the data of children nodes are computed by `expand_seed` with input

$$\texttt{0x03} \parallel \texttt{data[parent\_node\_index]} \parallel \texttt{salt} \parallel \texttt{repetition\_index} \parallel \texttt{parent\_node\_index}$$

   and the left child gets the first n-bit output then the right child gets the rest of n-bit if it exists (otherwise, the corresponding output is discarded).

PROCESS OF `has_sibling`.

Input: A tree structure and the index of a node `index`.

Output: Return 1 if the node of the input index has sibling, otherwise return 0.

1. If `exists[index] = 0`, return 0.

2. If $(\texttt{index}\%2 = 1) \wedge (\texttt{exists}[\texttt{index}+1] \neq 1)$, return 0.
3. Return 1.

PROCESS OF `reveal_all_but`. The function `reveal_all_but` outputs `reveal_list`, which is the set of intermediate nodes required to rebuild the (punctured) seed tree. Any node on the path from the missing leaf to the root are excluded, and the other siblings of those nodes are recorded in the `reveal_list`. In our implementation, `reveal_list` always is an array of the same length $\lceil \log(N) \rceil$ with n-bit elements.

Input: A tree structure `tree` and the index of the missing leaf `missing_index`.

Output: An array of n-bit data `reveal_list` of length $\lceil \log(N) \rceil$.

1. Zero-initialize `reveal_list`.
2. `path_index` $\leftarrow 0$.
3. Set the first node index `node` $\leftarrow$ `num_nodes` $- N +$ `missing_index`. This index will be updated as the index of the parent nodes in the next loop.
4. While `node` $\neq 0$:
   (a) If $\texttt{has\_sibling}(\texttt{tree}, \texttt{node}) = 0$:
      i. increase `path_index` by 1,
      ii. set `node` $\leftarrow \left\lfloor \frac{\texttt{node}-1}{2} \right\rfloor$.
   (b) Else:
      i. record the data of the sibling node to the `reveal_list`[`path_index`],
      ii. increase `path_index` by 1,
      iii. set `node` $\leftarrow \left\lfloor \frac{\texttt{node}-1}{2} \right\rfloor$.
5. Return `reveal_list`.

PROCESS OF `reconstruct_seed_tree`. The function `reconstruct_seed_tree` rebuilds the punctured seed tree from `reveal_list`.

Input: `reveal_list, salt, repetition_index, missing_index`.

Output: A recovered seed tree structure `tree`.

1. Allocate the tree structure to `tree`.
2. Set `exists`[`node`] $\leftarrow 1$ if `node` indicates leaf nodes or the root node.
3. Set `exists`[`node`] $\leftarrow 1$ if `node` indicates a non-leaf node with at least one child.
4. Set the first node index `node` $\leftarrow$ `num_nodes` $- N +$ `missing_index`. This index will be updated as the index of the parent nodes in the next loop.
5. While `node` $\neq 0$:
   (a) If $\texttt{has\_sibling}(\texttt{tree}, \texttt{node}) = 0$:
      i. increase `path_index` by 1,
      ii. set `node` $\leftarrow \left\lfloor \frac{\texttt{node}-1}{2} \right\rfloor$.
   (b) Else:

i. record reveal_list[path_index] to data[sibling_node] where sibling_node is the sibling's index of node,
ii. set have_value[sibling_node] $\leftarrow 1$,
iii. increase path_index by 1,
iv. set node $\leftarrow \left\lfloor \frac{\texttt{node}-1}{2} \right\rfloor$.

6. For an index node from 0 to (num_nodes $- N$):
   (a) If have_value[node] = exists[node] = 1 and node $\leq$ num_nodes, compute the data of children nodes, similarly as Step 4 of function make_seed_tree.
7. Return tree.

7.6.2 Committing to the party's seed and expanding tape: commit_to_seed_and_expand_tape

Input: The salt salt, repetition_index, party_index, the input party's seed seed.

Output: The commitment party_seed_commitments and the random tape random_tapes.

1. Absorb the hash prefix 0x04, salt, repetition_index, party_index, and seed to XOF in this order.
2. Squeeze $2n/8$ bytes from XOF to party_seed_commitments[repetition_index][party_index].
3. Squeeze $(\ell + 3)n/8$ bytes from XOF to random_tapes[repetition_index][party_index].
4. Output party_seed_commitments and random_tapes.

7.6.3 Computing the Challenge: h_1_commitment, h_2_commitment

PROCESS OF h_1_commitment.

Input: The message msg, the public key $pk = (\texttt{iv}, \texttt{ct})$, the salt salt,
the commitments party_seed_commitments[τ][N],
the share-adjusting values pt_delta[τ], c_delta[τ], and z_delta[τ][ℓ].

Output: The challenge hash h_1.

1. Absorb the hash prefix 0x01, msg, pk, and salt to XOF in this order.
2. For each parallel repetition k from 0 to $\tau - 1$:
   (a) For each party i from 0 to $N - 1$, absorb party_seed_commitments[k][i] to XOF.
   (b) Absorb pt_delta[k] to XOF.
   (c) For each AIM S-Box index j from 0 to $\ell - 1$, absorb z_delta[k][j] to XOF.
   (d) Absorb c_delta[k] to XOF.
3. Squeeze $2n/8$ bytes from XOF to h_1.
4. Output the challenge hash h_1.

PROCESS OF h_2_commitment.

Input: The challenge hash h_1, the salt salt,
the broadcast values alpha_shares[τ][N], and v_shares[τ][N].

Output: The challenge hash h_2.

1. Absorb the hash prefix `0x02`, `salt`, and `h_1` to XOF in this order.
2. For each parallel repetition `k` from 0 to $\tau - 1$:
   (a) For each party `i` from 0 to $N - 1$:
      i. Absorb `alpha_shares[k][i]` to XOF.
      ii. Absorb `v_shares[k][i]` to XOF.
3. Squeeze $2n/8$ bytes from XOF to `h_2`.
4. Output the challenge hash `h_2`.

7.6.4 Expanding the Challenge Hash: `h_1_expand`, `h_2_expand`

PROCESS OF `h_1_expand`.

Input: The challenge hash `h_1`.

Output: The challenge value `epsilons`$[\tau][\ell + 1]$.

1. Absorb `h_1` to XOF.
2. Declare the challenge value `epsilons`.
   For each parallel repetition `k` from 0 to $\tau - 1$:
   (a) For each AIM S-Box index `j` from 0 to ℓ:
      i. Squeeze $n/8$ bytes from XOF, and convert to a field element `epsilons[k][j]`.
3. Output the challenge value `epsilons`$[\tau][\ell + 1]$.

PROCESS OF `h_2_expand`.

Input: The challenge hash `h_2`.

Output: The challenge index `missing_parties`$[\tau]$.

1. Absorb `h_2` to XOF.
2. Initialize `squeeze_bytes` $\leftarrow N > 256\ ?\ 2 : 1$.
3. Initialize `mask` $\leftarrow (1 \ll \lceil \log(N) \rceil) - 1$.
4. Declare list of challenge indices `missing_parties`$[\tau]$.
   For each parallel repetition `k` from 0 to $\tau - 1$:
   (a) Squeeze `squeeze_bytes` bytes from XOF to `party`.
   (b) Compute `party` $\leftarrow$ `party` & `mask`.
   (c) If `party` $\geq N$, continue at Step 4.(a),
   else `missing_parties[k]` $\leftarrow$ `party`.
5. Output the challenge index `missing_parties`$[\tau]$.

7.6.5 MPC Simulation: aim_mpc

It computes MPC multiplication triples $\texttt{shared\_x}[N][\ell+1]$ and $\texttt{shared\_z}[N][\ell+1]$ as described in Section 4.2, Step 3 of Phase 1.

Input: The shares of the input pt of the parties $\texttt{shared\_pt}[N]$, the output of AIM ct, the linear components of AIM $(\texttt{matrix\_A}, \texttt{vector\_b})$, the number of parties N, and the shares of S-box outputs $\texttt{shared\_t}[N][\ell]$.

Output: The shares of multiplication triples $\texttt{shared\_z}[N][\ell+1]$ and $\texttt{shared\_x}[N][\ell+1]$.

1. Convert the output ct to a field element.
2. For each party i from 0 to $N-1$:
   (a) Convert $\texttt{shared\_pt}[\texttt{i}]$ to a field element.
   (b) For each AIM S-Box index j from 0 to $\ell-1$:
      i. Compute $\texttt{shared\_x}[\texttt{i}][\texttt{j}] \leftarrow \texttt{transposed\_matmul}(\texttt{shared\_t}[\texttt{i}][\texttt{j}], \texttt{matrix\_A}[\texttt{j}])$.
   (c) Compute $\texttt{shared\_x}[\texttt{i}][\ell] \leftarrow \texttt{shared\_x}[\texttt{i}][0] \oplus \cdots \oplus \texttt{shared\_x}[\texttt{i}][\ell-1]$.
   (d) If $\texttt{i} = 0$, compute $\texttt{shared\_x}[\texttt{i}][\ell] \leftarrow \texttt{shared\_x}[\texttt{i}][\ell] \oplus \texttt{vector\_b}$.
   (e) For each AIM S-Box index j from 0 to $\ell-1$:
      i. Compute $\texttt{shared\_x}[\texttt{i}][\texttt{j}] \leftarrow \texttt{shared\_t}[\texttt{i}][\texttt{j}]$.
      ii. Compute $\texttt{shared\_z}[\texttt{i}][\texttt{j}] \leftarrow \texttt{power\_of\_2\_exponentiation\_with\_e}_{\texttt{j}+1}(\texttt{shared\_pt}[\texttt{i}])$.
   (f) Compute
   $\texttt{shared\_z}[\texttt{i}][\ell] \leftarrow \texttt{ct} \times \texttt{shared\_x}[\texttt{i}][\ell] \oplus \texttt{power\_of\_2\_exponentiation\_with\_e}_*(\texttt{shared\_x}[\texttt{i}][\ell])$.
3. Output $\texttt{shared\_z}[N][\ell+1]$ and $\texttt{shared\_x}[N][\ell+1]$.

7.6.6 Serialization of Signatures

Input: The signature $\sigma = (\texttt{salt}, \texttt{h\_1}, \texttt{h\_2}, \texttt{proof}[\tau])$, where proof consists of $(\texttt{reveal\_list}, \texttt{missing\_commitment}, \texttt{pt\_delta}, \texttt{c\_delta}, \texttt{z\_delta}[\ell], \texttt{missing\_alpha\_share})$.

Output: A byte array sig, encoding the signature σ.

1. Write salt to sig, using $2n/8$ bytes.
2. Write h_1 to sig, using $2n/8$ bytes.
3. Write h_2 to sig, using $2n/8$ bytes.
4. Append tuples of proof of each repetition k from 0 to $\tau-1$,
   (a) Append reveal_list to sig, which is $\lceil \log(N) \rceil n/8$ bytes.
   (b) Append missing_commitment to sig, which is $2n/8$ bytes.
   (c) Append pt_delta to sig, which is $n/8$ bytes.
   (d) Append c_delta to sig, which is $n/8$ bytes.
   (e) Append $\texttt{z\_delta}[\ell]$ to sig, which is $\ell n/8$ bytes.
   (f) Append missing_alpha_share to sig, which is $n/8$ bytes.
5. Output sig.

7.6.7 Deserialization of Signatures

Input: A byte array sig, encoding the signature σ.

Output: The signature $\sigma = (\texttt{salt}, \texttt{h\_1}, \texttt{h\_2}, \texttt{proof}[\tau])$, where proof consists of (reveal_list, missing_commitment, pt_delta, c_delta, z_delta$[\ell]$, missing_alpha_share), challenge indices missing_parties$[\tau]$.

1. Read the first $2n/8$ bytes from sig, and assign them to salt.
2. Read the next $2n/8$ bytes from sig, and assign them to h_1.
3. Read the next $2n/8$ bytes from sig, and assign them to h_2.
4. Expand missing_parties$[\tau]$ = h_2_expand(h_2) as described in Section 7.6.4.
5. Read tuples from sig, and append them to proof[k] of each repetition k from 0 to $\tau - 1$,
   (a) Read the next $\lceil \log_2(N) \rceil n/8$ bytes from sig, and assign them to reveal_list.
   (b) Read the next $2n/8$ bytes from sig, and assign them to missing_commitment.
   (c) Read the next $n/8$ bytes from sig, and assign them to pt_delta.
   (d) Read the next $n/8$ bytes from sig, and assign them to c_delta.
   (e) Read the next $\ell n/8$ bytes from sig, and assign them to z_delta$[\ell]$.
   (f) Read the next $n/8$ bytes from sig, and assign them to missing_alpha_share.
6. Output (σ, missing_parties$[\tau]$).

8 Implementation and Performance

The implementation is available at https://aimer-signature.org. Our source codes are implemented with the BN++ repository[6] as a reference.

8.1 Implementation Details

TRANSPOSED MATRIX. In general, if the matrix is given in transposed form, matrix multiplication can be done more efficiently. Therefore, algorithms in AIMer for generating matrices from iv (Algorithm 6 and 7) are already well designed to generate transposed matrices directly, and we recommend performing all matrix multiplications in transposed form. In our implementation, the matrices e2_power_matrix defined in aim.h, and matrix_A generated in the functions generated_matrices_L_and_U and generated_matrices_LU are stored in the transposed form, and the multiplication of transposed matrices is done in GF_transposed_matmul.

MATRIX-BASED POWER-OF-2 EXPONENTIATION. During the MPC process in aim_mpc, for the k-th repetition and the i-th party, $(\mathsf{pt}_k^{(i)})^{2^{e_j}}$ is evaluated for $j \in \{1, \ldots, \ell\}$. As exponentiation of 2^{e_j} in $\mathbb{F}_{2^n}$ is linear over $\mathbb{F}_2$, $(\mathsf{pt}_k^{(i)})^{2^{e_j}}$ can be also derived from power_matrix $\cdot$ $(\mathsf{pt}_k^{(i)})$, where power_matrix is an $n \times n$ binary matrix corresponding to the 2^{e_j}-th power. The matrix multiplication can be faster than direct squaring for large exponents. Therefore, matrix-based power-of-2 exponentiation is applied for e_2 since e_2 has been chosen as a large number. The binary matrix corresponding to the 2^{e_2} exponentiation has been named as e2_power_matrix.

ADDITION CHAIN EXPONENTIATION. As described in Section 3.1, the S-boxes in AIM are defined as exponentiation by Mersenne numbers over a large field such as $x^{2^{e_j}-1}$ where $j \in \{1, \ldots, \ell\}$ and $x^{2^{e_*}-1}$ for $x \in \mathbb{F}_{2^n}$.

[6] https://github.com/IAIK/bnpp_helium_signatures/tree/main/bnpp_rain

Addition chain exponentiation [Knu97] by the shortest addition chain requires fewer field multiplications than binary exponentiation. For example, in the case of $x^{2^{27}-1}$ for AIM-I, binary exponentiation requires 26 field squarings and 26 field multiplications but addition chain exponentiation requires 26 field squarings and only 6 field multiplications by an addition chain $x \to x^{2^2-1} \to x^{2^3-1} \to x^{2^6-1} \to x^{2^{12}-1} \to x^{2^{24}-1} \to x^{2^{27}-1}$. All the S-boxes have been implemented using addition chain exponentiation by each shortest chain.

8.2 Performance

8.2.1 Description of the Benchmarking Environments

We describe our two implementations of AIMer signature scheme:

Reference. Our reference implementation was optimized using only C.

Optimized. We also provide an optimized implementation using AVX2 vector instructions.

We measured our reference and optimized implementations in Intel Xeon E5-1650 v3 @ 3.50 GHz with 128 GB of RAM on the Ubuntu 18.04 operating system. We also disabled TurboBoost and Hyper-threading features, and used the `taskset` command. All implementations used in the benchmarks were compiled using gcc 7.5.0 compiler with the optimization level `-O3`.

8.2.2 Key and Signature Sizes

In Table 8, we provide the size of AIMer public key, secret key, and signature for various parameter sets. These numbers are the same for both reference and optimized implementations.

Parameters	Public key size (bytes)	Secret key size (bytes)	Signature size (bytes)
AIMER_L1_PARAM1	32	16	5,904
AIMER_L1_PARAM2	32	16	4,880
AIMER_L1_PARAM3	32	16	4,176
AIMER_L1_PARAM4	32	16	3,840
AIMER_L3_PARAM1	48	24	13,080
AIMER_L3_PARAM2	48	24	10,440
AIMER_L3_PARAM3	48	24	9,144
AIMER_L3_PARAM4	48	24	8,352
AIMER_L5_PARAM1	64	32	25,152
AIMER_L5_PARAM2	64	32	19,904
AIMER_L5_PARAM3	64	32	17,088
AIMER_L5_PARAM4	64	32	15,392

Table 8: Key and signature sizes for various parameter sets.

8.2.3 Timing Results

In Tables 9 and 10, we provide the timing results as milliseconds and CPU clock cycles of reference and optimized implementations on the benchmark platform. The timing results were measured by the average clock cycles executed 10^4 times.

Parameters	Keygen		Sign		Verify	
	(ms)	(cycles)	(ms)	(cycles)	(ms)	(cycles)
AIMER_L1_PARAM1	0.02	59,483	1.23	4,294,114	1.15	4,011,553
AIMER_L1_PARAM2	0.02	59,654	2.94	10,284,335	2.88	10,077,658
AIMER_L1_PARAM3	0.02	59,593	9.66	33,819,763	9.59	33,555,727
AIMER_L1_PARAM4	0.02	59,582	48.16	168,559,507	47.55	166,436,892
AIMER_L3_PARAM1	0.04	131,234	3.08	10,767,276	2.92	10,222,797
AIMER_L3_PARAM2	0.04	130,656	8.07	28,254,891	7.93	27,738,451
AIMER_L3_PARAM3	0.04	131,852	23.63	82,706,117	23.93	83,765,726
AIMER_L3_PARAM4	0.04	131,911	120.14	420,497,831	114.99	402,461,878
AIMER_L5_PARAM1	0.09	311,887	6.06	21,217,778	5.83	20,395,571
AIMER_L5_PARAM2	0.09	312,090	15.56	54,457,539	15.29	53,516,330
AIMER_L5_PARAM3	0.09	313,543	47.85	167,472,963	46.66	163,325,301
AIMER_L5_PARAM4	0.09	314,257	231.94	811,789,935	227.73	797,067,009

Table 9: Performance of **reference** implementation for various parameter sets.

Parameters	Keygen		Sign		Verify	
	(ms)	(cycles)	(ms)	(cycles)	(ms)	(cycles)
AIMER_L1_PARAM1	0.02	54,552	0.59	2,079,167	0.53	1,840,810
AIMER_L1_PARAM2	0.02	54,143	1.36	4,747,229	1.28	4,474,325
AIMER_L1_PARAM3	0.02	54,178	4.42	15,476,644	4.31	15,075,635
AIMER_L1_PARAM4	0.02	54,435	22.29	78,022,625	21.09	73,813,256
AIMER_L3_PARAM1	0.03	118,533	1.38	4,838,748	1.28	4,494,766
AIMER_L3_PARAM2	0.03	119,231	3.59	12,562,067	3.44	12,048,460
AIMER_L3_PARAM3	0.03	118,816	9.77	34,202,905	9.62	33,670,751
AIMER_L3_PARAM4	0.03	118,691	53.38	186,813,161	50.73	177,567,471
AIMER_L5_PARAM1	0.08	284,746	2.45	8,573,223	2.34	8,181,552
AIMER_L5_PARAM2	0.08	283,908	6.26	21,925,850	6.07	21,245,240
AIMER_L5_PARAM3	0.08	283,931	18.66	65,302,783	17.75	62,135,635
AIMER_L5_PARAM4	0.08	285,114	91.76	321,174,411	88.83	310,906,616

Table 10: Performance of AVX2 **optimized** implementation for various parameter sets.

8.2.4 Memory Usage

In this section, we list the memory usage for our implementations. Since our implementations focus on the timing and signature size, optimization for memory usage are not considered. Memory usage was measured by the Valgrind[7]-3.13.0 with the subtool Massif. We utilized Massif using the following command:

```
valgrind --tool=massif --stacks=yes ./tests/test_sign
```

Then, for profiling output file, we utilized the tool `ms_print` using the following command:

```
ms_print massif.out.pid
```

The peak memory usage of reference and optimized implementations was described in Table 11.

Parameters	**Reference**		**Optimized**	
	Sign (KB)	Verify (KB)	Sign (KB)	Verify (KB)
AIMER_L1_PARAM1	195.6	193.1	195.8	192.9
AIMER_L1_PARAM2	439.7	440.6	439.7	440.6
AIMER_L1_PARAM3	1,395.5	1,397.6	1,395.5	1,397.6
AIMER_L1_PARAM4	6,759.1	6,761.3	6,759.1	6,761.2
AIMER_L3_PARAM1	426.6	421.3	492.7	487.8
AIMER_L3_PARAM2	1,039.8	1,042.1	1,212.7	1,211.6
AIMER_L3_PARAM3	3,060.8	3,065.4	3,564.9	3,569.6
AIMER_L3_PARAM4	14,797.0	14,802.4	17,206.9	17,211.8
AIMER_L5_PARAM1	854.9	846.7	855.0	846.7
AIMER_L5_PARAM2	2,066.1	2,058.4	2,068.4	2,058.3
AIMER_L5_PARAM3	6,174.7	6,183.0	6,174.6	6,182.9
AIMER_L5_PARAM4	29.740.7	29,749.2	29,740.6	29,749.2

Table 11: Peak memory usage of **reference** and **optimized** implementations

9 Advantages and Limitations

9.1 General

AIMer shares similar advantages with other MPCitH-based signature schemes as follows.

- The security of AIMer depends only on the security of the underlying symmetric primitives. In particular, the security of AIMer is reduced to the one-wayness of AIM in the random oracle model.
- Among the signature schemes whose security depends only on symmetric primitives, AIMer enjoys the smallest signature size.
- AIMer enjoys the small secret and public key size; the small key size makes it easier to apply to many PKI applications based on multi-chain certificates or frequent certificate transmission.
- Key generation is simple and fast.
- AIMer provides a granular trade-off between the execution time and the signature size. This feature makes it possible to adjust the performance based on the user's requirements.

[7] https://valgrind.org/docs/manual/ms-manual.html

- AIMer is resistant to the reuse of the public randomnesses such as iv and salt. To the best of our knowledge, multiple uses of an identical value of iv or salt linearly increase the probability of a pk-collision or a multi-target hash collision, respectively.

AIMer also has similar limitations to other MPCitH-based signature schemes as follows.

- The signature size is relatively large compared to standardized lattice-based schemes.
- Signing and verification is slower compared to standardized lattice-based schemes.

9.2 Compatibility with Existing Protocols

The signature size of AIMer is larger than NIST selected algorithms such as CRYSTALS-Dilithium [LDK+22] and Falcon [PFH+22] except SPHINCS+ [HBD+22], while the bandwidth of AIMer is sufficiently small so that it is still compatible with many existing protocols. We experimentally checked the compatibility of the optimized implementation of AIMer at all security levels with the Open Quantum Safe (OQS) project.[8] After creating X.509 certificates signed with AIMer, we were able to establish TLS 1.3 connections without message fragmentation, where the key exchange algorithm was the hybrid protocol with ECDH (p256/p384/p521) [BCR+18] and CRYSTALS-Kyber [SAB+22] (512/768/1024) algorithms in OQS.

References

[ACG+19] Martin R Albrecht, Carlos Cid, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic cryptanalysis of STARK-friendly designs: application to MARVELlous and MiMC. In *ASIACRYPT 2019*, pages 371–397. Springer, 2019.

[AD18] Tomer Ashur and Siemen Dhooghe. MARVELlous: a STARK-Friendly Family of Cryptographic Primitives. Cryptology ePrint Archive, Paper 2018/1098, 2018. https://eprint.iacr.org/2018/1098.

[AFK+11] Frederik Armknecht, Ewan Fleischmann, Matthias Krause, Jooyoung Lee, Martijn Stam, and John Steinberger. The preimage security of double-block-length compression functions. In *ASIACRYPT 2011*, pages 233–251. Springer, 2011.

[AFK22] Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-Shamir Transformation of Multi-round Interactive Proofs. In Eike Kiltz and Vinod Vaikuntanathan, editors, *Theory of Cryptography*, pages 113–142, Cham, 2022. Springer Nature Switzerland.

[AGR+16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT 2016*, pages 191–219. Springer, 2016.

[AIK+01] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design and Analysis. In *SAC 2001*, pages 39–56. Springer, 2001.

[AMMR13] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013.

[ARS+15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT 2015*, pages 430–454. Springer, 2015.

[8] http://github.com/open-quantum-safe/liboqs

[BB18] Gustavo Banegas and Daniel J. Bernstein. Low-Communication Parallel Quantum Multi-Target Preimage Search. In *Selected Areas in Cryptography – SAC 2017*, pages 325–335. Springer, 2018.

[BBCV22] Subhadeep Banik, Khashayar Barooti, Andrea Caforio, and Serge Vaudenay. Memory-Efficient Single Data-Complexity Attacks on LowMC Using Partial Sets. Cryptology ePrint Archive, Paper 2022/688, 2022. https://eprint.iacr.org/2022/688.

[BBDV20] Subhadeep Banik, Khashayar Barooti, F. Betül Durak, and Serge Vaudenay. Cryptanalysis of LowMC instances using single plaintext/ciphertext pair. *IACR Transactions on Symmetric Cryptology*, 2020(4):130–146, Dec. 2020.

[BBP+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private Multiplication over Finite Fields. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 397–426, Cham, 2017. Springer International Publishing.

[BBVY21] Subhadeep Banik, Khashayar Barooti, Serge Vaudenay, and Hailun Yan. New Attacks on LowMC Instances with a Single Plaintext/Ciphertext Pair. In *ASIACRYPT 2021*, pages 303–331. Springer, 2021.

[BCR+18] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, 2018. NIST SP 800-56A Rev.3.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Bei93] R. Beigel. The polynomial method in circuit complexity. In *[1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference*, pages 82–95, 1993.

[Ber09] Daniel J Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete, 2009.

[BFS04] Magali Bardet, Jean-Charles Faugere, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004.

[BFSS13] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the complexity of solving quadratic Boolean systems. *Journal of Complexity*, 29(1):53–75, 2013.

[BHNP+19] Xavier Bonnetain, Akinori Hosoyamada, María Naya-Plasencia, Yu Sasaki, and André Schrottenloher. Quantum Attacks Without Superposition Queries: The Offline Simon's Algorithm. In *ASIACRYPT 2019*, pages 552–583. Springer, 2019.

[BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN'98: Theoretical Informatics: Third Latin American Symposium Campinas, Brazil, April 20–24, 1998 Proceedings 3*, pages 163–169. Springer, 1998.

[BN20] Carsten Baum and Ariel Nof. Concretely-Efficient Zero-Knowledge Arguments for Arithmetic Circuits and Their Application to Lattice-Based Cryptography. In *PKC 2020*, pages 495–526. Springer, 2020.

[BS00] Alex Biryukov and Adi Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In *Advances in Cryptology — ASIACRYPT 2000*, pages 1–13. Springer, 2000.

[BSGK+21] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In *PKC 2021*, pages 266–297. Springer, 2021.

[BWM99] Simon Blake-Wilson and Alfred Menezes. Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol. In *Public Key Cryptography*, pages 154–170. Springer, 1999.

[BXKSN21] Roland Booth, Yanhong Xu, Sabyasachi Karati, and Reihaneh Safavi-Naini. An Intermediate Secret-Guessing Attack on Hash-Based Signatures. In Toru Nakanishi and Ryo Nojima, editors, *Advances in Information and Computer Security*, pages 195–215. Springer, 2021.

[BY18] Daniel J. Bernstein and Bo-Yin Yang. Asymptotically Faster Quantum Algorithms to Solve Multivariate Quadratic Equations. In *PQCrypto 2018*, pages 487–506. Springer, 2018.

[CDG06] Nicolas T. Courtois, Blandine Debraize, and Eric Garrido. On Exact Algebraic [Non-]Immunity of S-Boxes Based on Power Functions. In *ACISP 2006*, pages 76–86. Springer, 2006.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *ACM CCS 2017*, pages 1825–1842, 2017.

[CG22] Yu-Ao Chen and Xiao-Shan Gao. Quantum Algorithm for Boolean Equation Solving and Quantum Algebraic Attack on Cryptosystems. *Journal of Systems Science and Complexity*, 35(1):373–412, Feb 2022.

[CKPS00] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *EUROCRYPT 2000*, pages 392–407. Springer, 2000.

[DFM20] Jelle Don, Serge Fehr, and Christian Majenz. The Measure-and-Reprogram Technique 2.0: Multi-Round Fiat-Shamir and More. In *CRYPTO 2020*, page 602–631. Springer, 2020.

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model. In *Advances in Cryptology – CRYPTO 2019*, volume 11693 of *Lecture Notes in Computer Science*, pages 356–383. Springer, 2019.

[DFMS22a] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Efficient NIZKs and Signatures from Commit-and-Open Protocols in the QROM. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 729–757. Springer Nature Switzerland, 2022.

[DFMS22b] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Online-Extractability in the Quantum Random-Oracle Model. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 677–706. Springer International Publishing, 2022.

[DGD+19] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. X-DeepSCA: Cross-Device Deep Learning Side Channel Attack. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[DGG+21] Jintai Ding, Vlad Gheorghiu, András Gilyén, Sean Hallgren, and Jianqiang Li. Limitations of the Macaulay matrix approach for using the HHL algorithm to solve multivariate polynomial systems. arXiv 2111.00405, 2021. https://arxiv.org/abs/2111.00405.

[Din21] Itai Dinur. Cryptanalytic Applications of the Polynomial Method for Solving Multivariate Equation Systems over GF(2). In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 374–403, Cham, 2021. Springer.

[DKR+22] Christoph Dobraunig, Daniel Kales, Christian Rechberger, Markus Schofnegger, and Greg Zaverucha. Shorter Signatures Based on Tailor-Made Minimalist Symmetric-Key Crypto. In *ACM CCS 2022*, pages 843–857. Association of Computing Machinery, November 2022.

[DN19] Itai Dinur and Niv Nadler. Multi-target Attacks on the Picnic Signature Scheme and Related Protocols. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 699–727, Cham, 2019. Springer.

[DNG22] Elena Dubrova, Kalle Ngo, and Joel Gärtner. Breaking a Fifth-Order Masked Implementation of CRYSTALS-Kyber by Copy-Paste. Cryptology ePrint Archive, Paper 2022/1713, 2022. https://eprint.iacr.org/2022/1713.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*, volume 2. Springer, 2002.

[dSGMOS19] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P Smart. BBQ: Using AES in picnic signatures. In *SAC 2019*, pages 669–692. Springer, 2019.

[DVA12] Joan Daemen and Gilles Van Assche. Differential Propagation Analysis of Keccak. In Anne Canteaut, editor, *Fast Software Encryption*, pages 422–441. Springer, 2012.

[EGL+20] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden, Christian Rechberger, Markus Schofnegger, and Qingju Wang. An algebraic attack on ciphers with low-degree round functions: application to full MiMC. In *ASIACRYPT 2020*, pages 477–506. Springer, 2020.

[EM97] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3):151–161, Jun 1997.

[FHK+17] Jean-Charles Faugère, Kelsey Horan, Delaram Kahrobaei, Marc Kaplan, Elham Kashefi, and Ludovic Perret. Fast Quantum Algorithm for Solving Multivariate Quadratic Equations. Cryptology ePrint Archive, Paper 2017/1236, 2017. https://eprint.iacr.org/2017/1236.

[Frö85] Ralf Fröberg. An Inequality for Hilbert Series of Graded Algebras. *MATHEMATICA SCANDINAVICA*, 56, Dec. 1985.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Security 2021*, pages 519–535. USENIX Association, 2021.

[GLR+20] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy. In *EUROCRYPT 2020*, pages 674–704. Springer, 2020.

[GMLS02] Steven D Galbraith, John Malone-Lee, and Nigel Paul Smart. Public key signatures in the multi-user setting. *Information Processing Letters*, 83(5):263–266, 2002.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security 2016*, pages 1069–1083. USENIX Association, 2016.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure against Adaptive Chosen-Message Attacks. *SIAM J. Comput.*, 17(2):281–308, apr 1988.

[Gro96] Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *ACM STOC '96*, page 212–219. Association for Computing Machinery, 1996.

[HBD+22] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2022, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[HHL09] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum Algorithm for Linear Systems of Equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009.

[HS18] Akinori Hosoyamada and Yu Sasaki. Cryptanalysis Against Symmetric-Key Schemes with Online Classical Queries and Offline Quantum Computations. In *CT-RSA 2018*, pages 198–218. Springer, 2018.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from Secure Multiparty Computation. In *ACM STOC 2007*, pages 21–30, 2007.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[JBK+22] Kyungbae Jang, Anubhab Baksi, Hyunji Kim, Hwajeong Seo, and Anupam Chattopadhyay. Improved Quantum Analysis of SPECK and LowMC (Full Version). Cryptology ePrint Archive, Paper 2022/1427, 2022. https://eprint.iacr.org/2022/1427.

[JKL+22] Kyungbae Jang, Wonwoong Kim, Sejin Lim, Yeajun Kang, and Hwajeong Seo. Optimized Implementation of Quantum Binary Field Multiplication with Toffoli Depth One. In *Information Security Applications*. Springer, 2022. To appear.

[JKO+23] Kyungbae Jang, Dukyoung Kim, Yujin Oh, Sejin Lim, Yujin Yang, Hyunji Kim, and Hwajeong Seo. Quantum Implementation of AIM: Aiming for Low-Depth. Cryptology ePrint Archive, Paper 2023/337, 2023. https://eprint.iacr.org/2023/337.

[KHS+22] Seongkwang Kim, Jincheol Ha, Mincheol Son, Byeonghak Lee, Dukjae Moon, Joohee Lee, Sangyub Lee, Jihoon Kwon, Jihoon Cho, Hyojin Yoon, and Jooyoung Lee. AIM: Symmetric Primitive for Shorter Signatures with Stronger Security (Full Version). Cryptology ePrint Archive, Paper 2022/1387, 2022. To appear at ACM CCS 2023.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved Non-Interactive Zero Knowledge with Applications to Post-Quantum Signatures. In *ACM CCS 2018*, pages 525–537. ACM, 2018.

[KM10] Hidenori Kuwakado and Masakatu Morii. Quantum distinguisher between the 3-round Feistel cipher and the random permutation. In *2010 IEEE International Symposium on Information Theory*, pages 2682–2685, 2010.

[KM12] Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type Even-Mansour cipher. In *2012 International Symposium on Information Theory and its Applications*, pages 312–316, 2012.

[KM13] Neal Koblitz and Alfred Menezes. Another look at security definitions, 2013.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[KR01] Joe Kilian and Phillip Rogaway. How to Protect DES Against Exhaustive Key Search (an Analysis of DESX). *Journal of Cryptology*, 14(1):17–35, Jan 2001.

[KR19] Yael Tauman Kalai and Leonid Reyzin. *A Survey of Leakage-Resilient Cryptography*, page 727–794. Association for Computing Machinery, New York, NY, USA, 2019.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE Public Key Cryptosystem by Relinearization. In *CRYPTO '99*, pages 19–30. Springer, 1999.

[KSW19] Daniel J Katz, KU Schmidt, and A Winterhof. Weil sums of binomials: Properties applications and open problems. In *Combinatorics and Finite Fields: Difference Sets, Polynomials, Pseudorandomness and Applications*, volume 23, pages 109–134. De Gruyter, 2019.

[KZ22] Daniel Kales and Greg Zaverucha. Efficient Lifting for Shorter Zero-Knowledge Proofs and Post-Quantum Signatures. Cryptology ePrint Archive, Paper 2022/588, 2022. https://eprint.iacr.org/2022/588.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[LIM21a] Fukang Liu, Takanori Isobe, and Willi Meier. Cryptanalysis of full LowMC and LowMC-M with algebraic techniques. In *CRYPTO 2021*, pages 368–401. Springer, 2021.

[LIM21b] Fukang Liu, Takanori Isobe, and Willi Meier. Low-Memory Algebraic Attacks on Round-Reduced LowMC. *Cryptology ePrint Archive*, 2021.

[LM17] Gregor Leander and Alexander May. Grover Meets Simon – Quantumly Attacking the FX-construction. In *ASIACRYPT 2017*, pages 161–178. Springer, 2017.

[LMSI22] Fukang Liu, Willi Meier, Santanu Sarkar, and Takanori Isobe. New Low-Memory Algebraic Attacks on LowMC in the Picnic Setting. *IACR Transactions on Symmetric Cryptology*, 2022(3):102–122, Sep. 2022.

[LPT+17] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, Ryan Williams, and Huacheng Yu. Beating Brute Force for Systems of Polynomial Equations over Finite Fields. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2190–2202. SIAM, 2017.

[LR86] Michael Luby and Charles Rackoff. How to Construct Pseudo-random Permutations from Pseudo-random Functions. In *CRYPTO '85*, pages 447–447. Springer, 1986.

[LSW+22] Fukang Liu, Santanu Sarkar, Gaoli Wang, Willi Meier, and Takanori Isobe. Algebraic Meet-in-the-Middle Attack on LowMC. Cryptology ePrint Archive, Paper 2022/019, 2022. https://eprint.iacr.org/2022/019, to appear Asiacrypt 2022.

[LZ19] Qipeng Liu and Mark Zhandry. Revisiting Post-quantum Fiat-Shamir. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 326–355, Cham, 2019. Springer International Publishing.

[MCT17] Edgard Muñoz-Coreas and Himanshu Thapliyal. Design of Quantum Circuits for Galois Field Squaring and Exponentiation. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 68–73, 2017.

[MS04] Alfred Menezes and Nigel Smart. Security of Signature Schemes in a Multi-User Setting. *Designs, Codes and Cryptography*, 33(3):261–274, Nov 2004.

[NIS15] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. FIPS PUB 202.

[NIS22] NIST. Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, 2022, 2022. available at https://csrc.nist.gov/projects/pqc-dig-sig.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[PKC22] Ray Perlner, John Kelsey, and David Cooper. Breaking Category Five SPHINCS$^+$ with SHA-256. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography*, pages 501–522. Springer, 2022.

[QS01] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *FSE 2004*, pages 371–388. Springer, 2004.

[RST18] Christian Rechberger, Hadi Soleimany, and Tyge Tiessen. Cryptanalysis of Low-Data Instances of Full LowMCv2. *IACR Transactions on Symmetric Cryptology*, 2018(3):163–181, 2018.

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

[Sim97] Daniel R. Simon. On the Power of Quantum Computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In *FSE 2007*, pages 181–195. Springer, 2007.

[WD20] Huanyu Wang and Elena Dubrova. Tandem Deep Learning Side-Channel Attack Against FPGA Implementation of AES. In *2020 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, pages 147–150, 2020.

[Wil14] Richard Ryan Williams. The Polynomial Method in Circuit Complexity Applied to Algorithm Design (Invited Talk). In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47–60, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[YAG21] Mahmoud Yehia, Riham AlTawy, and T. Aaron Gulliver. Security Analysis of DGM and GM Group Signature Schemes Instantiated with XMSS-T. In Yu Yu and Moti Yung, editors, *Information Security and Cryptology*, pages 61–81. Springer, 2021.

[Zha12] Mark Zhandry. How to Construct Quantum Random Functions. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 679–687, 2012.