

AlMer Signature Scheme

KpqC Winter Camp 2026

Seongkwang Kim

Samsung SDS

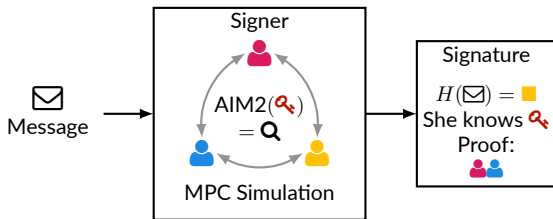
Introduction

What is AlMer?

- Post-quantum digital signature scheme
- Based on **MPC-in-the-Head (MPCitH)** paradigm
- Security relies on **symmetric-key primitives only**
 - No number-theoretic assumptions
- Designed by researchers in Samsung SDS, KAIST, and Sungshin Women's University

AIMer at a Glance

- **Key idea:** Prove knowledge of preimage of a one-way function using ZK proof
- **Two main components:**
 1. AIM2 one-way function
 2. Customized BN++ ZK proof system



- **Six variants:**
 - **aimer128f/s**, **aimer192f/s**, **aimer256f/s**
 - **f = fast** (faster performance, larger signature)
 - **s = short** (slower performance, smaller signature)

AImer Performance Summary

Variant	Security	KeyGen (ms)	Sign (ms)	Verify (ms)	Sig. Size (B)	PK Size (B)
aimer128f	128	0.03	0.42	0.41	5,888	32
aimer128s	128	0.03	3.18	3.13	4,160	32
aimer192f	192	0.05	1.04	1.03	13,056	48
aimer192s	192	0.05	7.94	7.86	9,120	48
aimer256f	256	0.10	2.07	2.03	25,120	64
aimer256s	256	0.10	15.26	14.81	17,056	64

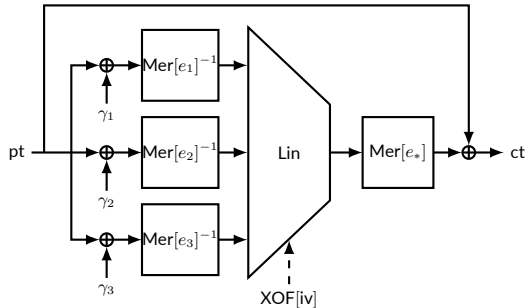
Benchmarked on Intel Xeon E5-1650 v3 @ 3.50GHz

The AIM2 One-Way Function

AIM2 Overview

- One-way function based on large S-boxes
- Operates over binary extension field \mathbb{F}_{2^n}
- **Secret key:** (pt, iv, ct)
- **Public key:** (iv, ct) pair where $ct = \text{AIM2}(pt, iv)$
- **IV** (initialization vector): randomly sampled by a user, determines the linear layer
- **Three security levels:**
 - Level I: $n = 128$
 - Level III: $n = 192$
 - Level V: $n = 256$

AIM2 Structure



Data flow:

1. $pt + \gamma_i$ into parallel S-boxes
2. S-box outputs through Lin layer
3. Final S-box $\text{Mer}[e_*]$
4. XOR with $pt \Rightarrow ct$

Lin is generated from iv via XOF.

AIM2 Components: S-boxes

- **Inverse Mersenne S-boxes** (power permutations): $x^{(2^{e_i}-1)^{-1}}$
 - Inversion done modulo $2^n - 1$
 - Exponent notation: $\text{Mer}[e]^{-1}$ means the inverse monomial map
- **First layer:** ℓ parallel S-box branches ($\ell \in \{2, 3\}$)
 - Each branch: $t_i = (\text{pt} + \gamma_i)^{(2^{e_i}-1)^{-1}}$
 - γ_i are IV-independent constants
- **Mersenne S-box:** $x^{2^{e^*}-1}$

AIM2 Components: Linear Layer

- **Random \mathbb{F}_2 -linear layer:** $\text{Lin} : (\mathbb{F}_{2^n})^\ell \rightarrow \mathbb{F}_{2^n}$
 - Generated pseudorandomly from IV using XOF (XOF)

- **Structure:**

$$\text{Lin}(t_1, \dots, t_\ell) = L_1(t_1) + \dots + L_\ell(t_\ell) + b$$

- Each L_i : bijective \mathbb{F}_2 -linear component
- **Properties:**
 - Provides diffusion and randomization depending on IV

- **Full computation:**

$$c = (\text{Lin}(t_1, \dots, t_\ell))^{2^{e^*} - 1} \oplus \text{pt}$$

AIM2 Parameters

Model	n	ℓ	e_1	e_2	e_3	e_*	Field
I	128	2	49	91	–	3	$\mathbb{F}_{2^{128}}$
III	192	2	17	47	–	5	$\mathbb{F}_{2^{192}}$
V	256	3	11	141	7	3	$\mathbb{F}_{2^{256}}$

Key Generation & OWF Evaluation

Key Generation Algorithm:

1. Sample random secret key $pt \xleftarrow{\$} \mathbb{F}_{2^n}$
2. Choose initialization vector iv
3. Compute ciphertext $ct = \text{AIM2}(pt, iv)$
4. Output $sk = (pt, iv, ct)$, $pk = (iv, ct)$

OWF Security:

- Given (iv, ct) , finding pt is computationally hard
- Security reduces to hardness of inverting AIM2

Zero-Knowledge Protocol

MPC-in-the-Head Paradigm

Core Idea: Simulate an N -party MPC protocol “in the head” of the prover

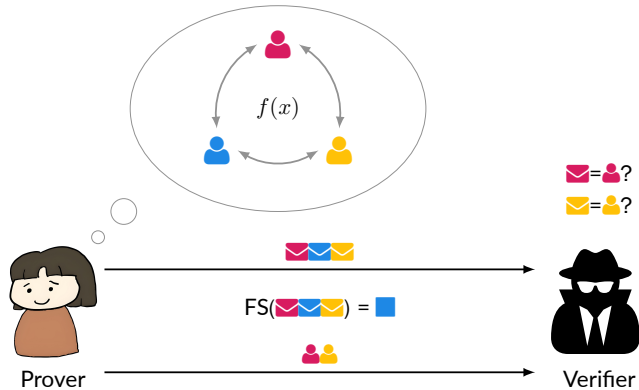
Protocol Flow:

- Prover splits secret key into N additive shares
- Simulates MPC computation of the one-way function
- Verifier challenges: open all parties except one
- If computation is consistent \rightarrow prover knows the secret key

Security Guarantees:

- One hidden party reveals nothing about the secret
- Fiat-Shamir heuristic: hash-based challenges \rightarrow non-interactive

MPCitH Signature Overview



Key Steps:

- Prover commits to N simulated parties
- Fiat-Shamir challenge determines which party to hide
- Verifier checks consistency of opened parties

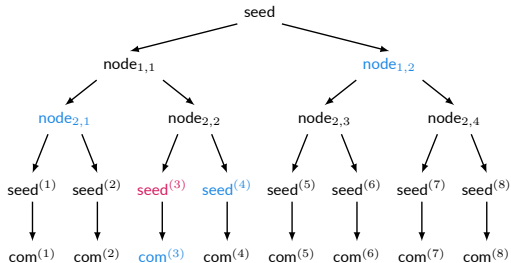
AIMer Protocol: Steps 1 & 2

Step 1: Party Simulation

- Expand master seed via GGM tree into N party seeds
- Each party seed \rightarrow PRG \rightarrow random shares $(pt^{(i)}, t_j^{(i)}, a_j^{(i)}, c^{(i)})$
- Commit to each party:
 $com^{(i)} = H(seed^{(i)})$

Step 2: Mult. Triple Generation

- From shares, compute correction values for multiplication triples
- Ensure $pt \cdot a_0 + \sum_j t_j \cdot a_j = c$ (with corrections)
- These triples enable verifiable computation of AIM2



AlMer Protocol: Steps 3 & 4

Step 3: Multiplication Check via Fiat-Shamir

- Hash commitments \rightarrow **Challenge 1:** $\epsilon_0, \dots, \epsilon_\ell \in \mathbb{F}_{2^n}$
- Each party i computes:

$$\begin{aligned}\alpha^{(i)} &= a^{(i)} + \sum_{j=0}^{\ell} \epsilon_j \cdot x_j^{(i)} \\ v^{(i)} &= c^{(i)} + \text{pt}^{(i)} \cdot \alpha + \sum_{j=0}^{\ell} \epsilon_j \cdot z_j^{(i)}\end{aligned}$$

- Open $\alpha = \sum_i \alpha^{(i)}$, then publish all $\{(\alpha^{(i)}, v^{(i)})\}_i$
- Verification: $\sum_i v^{(i)} = 0$ confirms multiplication triples are valid

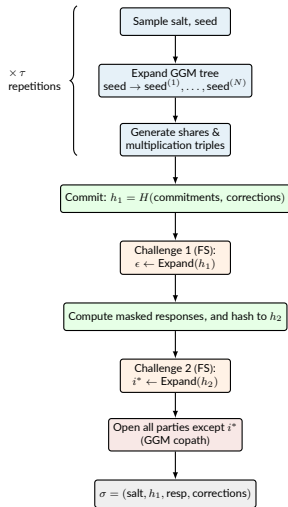
Step 4: Party Opening

- Hash $(\alpha^{(i)}, v^{(i)}) \rightarrow$ **Challenge 2:** party index i^* to hide
- Open all parties except i^* : reveal GGM copath
- Verifier reconstructs all opened parties, checks consistency

AlMer Signing Algorithm

Sign(sk, μ):

1. Sample salt, master seed
2. For each repetition $1, \dots, \tau$:
 - Expand GGM tree $\rightarrow N$ party seeds
 - Generate shares, compute corrections
3. Commit:
 $h_1 = H(\text{salt}, \mu, \text{com}^{(1)}, \dots, \text{com}^{(N)}, \dots)$
4. **Challenge 1** (FS): $\epsilon \leftarrow \text{Expand}(h_1)$
5. Compute masked responses, and hash
 $h_2 \leftarrow H(h_1, \text{resp})$
6. **Challenge 2** (FS): $i^* \leftarrow \text{Expand}(h_2)$
7. Open: reveal GGM copath (except i^*)
8. Output: $\sigma = (\text{salt}, h_1, \text{resp}, \text{corrections})$



AlMer Verification Algorithm

Verify(pk, μ , σ):

1. Parse signature $\sigma = (\text{salt}, h_1, \text{responses}, \text{corrections})$
2. Recompute party seeds from GGM copaths
3. Regenerate shares $(\text{pt}^{(i)}, t_j^{(i)}, a_j^{(i)}, c^{(i)})$ for opened parties
4. Recompute **Challenge 1**: $\epsilon \leftarrow \text{Expand}(h_1)$
5. Verify masked responses are consistent with opened shares
6. Recompute **Challenge 2**: $i^* \leftarrow \text{Expand}(h_2)$
7. Verify party index i^* matches hidden party
8. Check commitment consistency for hidden party
9. Accept if and only if all checks pass

Signature Size Analysis

Signature Components:

- $\left(\text{salt}, h_1, h_2, \left(\text{path}_k, \text{com}_k^{(\bar{i}_k)}, \Delta \text{pt}_k, (\Delta t_{k,j})_{j \in [\ell]}, \Delta c_k, \alpha_k^{(\bar{i}_k)} \right)_{k \in [\tau]} \right)$

Trade-off: Controlled by repetitions τ and parties N

- f-variants:** Fewer repetitions, more parties \rightarrow fast
- s-variants:** More repetitions, less parties \rightarrow short

Parameter Set	Repetitions τ	Parties N	Signature Size
aimer128f	33	16	5,888 B
aimer128s	17	256	4,160 B
aimer192f	49	16	13,056 B
aimer192s	25	256	9,120 B
aimer256f	65	16	25,120 B
aimer256s	33	256	17,056 B

Reference Implementation

Code Repository Overview

GitHub: [samsungsds-opensource/AIMER](https://github.com/samsungsds-opensource/AIMER) **Language:** C (portable)

Source Files:

```
Reference_Implementation/
+-- aim2.c / aim2.h           -- AIM2 one-way function
+-- sign.c / sign.h           -- Sign & Verify (core protocol)
+-- tree.c / tree.h           -- GGM seed tree
+-- field128.c / field192.c    -- Field arithmetic
|   field256.c / field.h
+-- hash.c / hash.h           -- Hash / XOF wrappers
+-- params.h                   -- Parameter definitions
```

Key Data Structures (from `sign.h`):

```
typedef struct tape_t {           // per-party random tape
    gf pt_share;                 // secret key share
    gf t_shares[AIMER_L];       // S-box output shares
    gf a_share;                  // mult check randomness
    gf c_share;                  // mult check correction
} tape_t;
```

Code: Field Arithmetic (gf)

```
1 // field.h -- element of GF(2^n)
2 #if SECURITY_BITS == 128
3 typedef uint64_t gf[2]; // 128 bits
4 #elif SECURITY_BITS == 192
5 typedef uint64_t gf[3]; // 192 bits
6 #else
7 typedef uint64_t gf[4]; // 256 bits
8 #endif

1 // field128.c -- squaring in GF(2^128)
2 void gf_sqr(gf c, const gf a)
3 {
4     uint64_t t = 0;
5     uint64_t temp[4] = {0,};
6     poly64_sqr(&temp[1], &temp[0], a[0]);
7     poly64_sqr(&temp[3], &temp[2], a[1]);
8     // reduction mod x^128 + x^7 + x^2 + x + 1
9     t = temp[2]
10        ^ ((temp[3]>>57)^(temp[3]>>62)
11           ^ (temp[3]>>63));
12     c[1] = temp[1] ^ temp[3];
13     c[1] ^= (temp[3]<<7) | (t>>57);
14     c[1] ^= (temp[3]<<2) | (t>>62);
15     c[1] ^= (temp[3]<<1) | (t>>63);
16     c[0] = temp[0] ^ t;
17     c[0] ^= (t<<7);
18     c[0] ^= (t<<2);
19     c[0] ^= (t<<1);
20 }
```

Representation:

- Element of \mathbb{F}_{2^n} stored as array of `uint64_t`
- 128-bit: 2 words, 192-bit: 3 words, 256-bit: 4 words

Key operations (from `field.h`):

- `gf_add`: XOR (addition in \mathbb{F}_{2^n})
- `gf_mul`: Schoolbook + reduction
- `gf_sqr`: Schoolbook + reduction
- `gf_exp`: square-and-multiply
- `gf_mat_vec_mul`: matrix-vector over \mathbb{F}_2

Reduction polynomial ($n = 128$):

$$f(x) = x^{128} + x^7 + x^2 + x + 1$$

Squaring:

- `poly64_sqr`: squaring of a 64-bit word without reduction
- Produces 256-bit result in `temp[0..3]`
- Reduce using shifts matching $f(x)$

Code: Key Generation

```
1 int crypto_sign_keypair(  
2     uint8_t *pk, uint8_t *sk)  
3 {  
4     uint8_t pt[AIM2_NUM_BYTES_FIELD];  
5     uint8_t iv[AIM2_IV_SIZE];  
6  
7     randombytes(pt, AIM2_NUM_BYTES_FIELD);  
8     randombytes(iv, AIM2_IV_SIZE);  
9     crypto_sign_keypair_internal(  
10        pk, sk, pt, iv);  
11     return 0;  
12 }
```

```
1 void crypto_sign_keypair_internal(  
2     uint8_t *pk, uint8_t *sk,  
3     const uint8_t *pt, const uint8_t *iv)  
4 {  
5     aim2(pk + AIM2_IV_SIZE, pt, iv);  
6  
7     memcpy(pk, iv, AIM2_IV_SIZE);  
8     memcpy(sk, pt, AIM2_NUM_BYTES_FIELD);  
9     memcpy(sk + AIM2_NUM_BYTES_FIELD, pk,  
10        AIM2_IV_SIZE + AIM2_NUM_BYTES_FIELD);  
11 }
```

Simplest entry point — 3 steps:

1. Sample randomness

- $pt \xleftarrow{\$} \mathbb{F}_{2^n}$ (secret key / plaintext)
- $iv \xleftarrow{\$} \{0, 1\}^{128}$

2. Evaluate OWF

$$ct = \text{AIM2}(pt, iv)$$

3. Pack keys

- $pk = iv || ct$
- $sk = pt || iv || ct$

Note: sk contains pk as a suffix.
This avoids recomputing ct during signing.

Code: Sign & Verify (Wrapper Functions)

```
1 int crypto_sign_signature(  
2     uint8_t *sig, size_t *siglen,  
3     const uint8_t *m, size_t mlen,  
4     const uint8_t *ctx, size_t ctxlen,  
5     const uint8_t *sk)  
6 {  
7     uint8_t prefix[256];  
8     uint8_t randomness[SECURITY_BYTES];  
9     prefix[0] = ctxlen;  
10    memcpy(prefix + 1, ctx, ctxlen);  
11    #ifdef RANDOMIZED_SIGNING  
12    randombytes(randomness, SECURITY_BYTES);  
13    #else  
14    memset(randomness, 0, SECURITY_BYTES);  
15    #endif  
16    crypto_sign_signature_internal(  
17        sig, siglen, m, mlen,  
18        prefix, 1 + ctxlen, randomness, sk);  
19    return 0;  
20 }  
  
1 int crypto_sign_verify(  
2     const uint8_t *sig, size_t siglen,  
3     const uint8_t *m, size_t mlen,  
4     const uint8_t *ctx, size_t ctxlen,  
5     const uint8_t *pk)  
6 {  
7     uint8_t prefix[256];  
8     prefix[0] = ctxlen;  
9     memcpy(prefix + 1, ctx, ctxlen);  
10    return crypto_sign_verify_internal(  
11        sig, siglen, m, mlen,  
12        prefix, 1 + ctxlen, pk);  
13 }
```

Wrappers handle context string encoding before calling internal functions.

crypto_sign_signature:

- Encode context: $\text{prefix} = \text{len} || \text{ctx}$
- Generate signing randomness ρ
- RANDOMIZED_SIGNING: fresh ρ each time (default) vs. deterministic ($\rho = 0$)
- Delegates to `_internal` \Rightarrow next slide

crypto_sign_verify:

- Same context encoding as signing
- Delegates to `_internal`
- Returns 0 on success, -1 on failure

Design pattern: Thin wrappers \rightarrow internal functions.
Separates wrappers from core protocol logic to do a deterministic test.

Code: Signing Overview (5 Phases)

```
1 void crypto_sign_signature_internal(
2     uint8_t *sig, size_t *siglen,
3     const uint8_t *m, size_t mlen,
4     const uint8_t *pre, size_t prelen,
5     const uint8_t *rnd, const uint8_t *sk)
6 {
7     signature_t *sign = (signature_t *)sig;
8
9     // Phase 1: Commit to seeds & views
10    run_phase_1(sign, commits, nodes,
11        mult_chk, alpha_v_shares,
12        sk, rnd, m, mlen, pre, prelen);
13
14    // Phase 2-3: Multiplication checking
15    run_phase_2_and_3(sign, alpha_v_shares,
16        mult_chk);
17
18    // Phase 4: Challenge party indices
19    hash_init(&ctx);
20    hash_update(&ctx, sign->h_2, ...);
21    hash_squeeze(&ctx, indices, AIMER_T);
22
23    // Phase 5: Open all but hidden party
24    for (rep = 0; rep < AIMER_T; rep++) {
25        i_bar = indices[rep];
26        reveal_all_but(sign->proofs[rep]
27            .reveal_path, nodes[rep], i_bar);
28        memcpy(sign->proofs[rep]
29            .missing_commitment,
30            commits[rep][i_bar], ...);
31    }
32    *siglen = CRYPTO_BYTES;
33 }
```

Top-level signing flow:

Phase 1: Commit to party views

- Expand GGM tree $\rightarrow N$ party seeds
- Generate tapes, compute aim2_mpc
- Output: h_1 , correction values δ

Phase 2-3: Multiplication check

- $\epsilon \leftarrow \text{Expand}(h_1)$ (Challenge 1)
- Aggregate α, v shares, output h_2

Phase 4: Select hidden party

- $i^* \leftarrow \text{Expand}(h_2)$ (Challenge 2)

Phase 5: Open views

- Reveal GGM copath (all except i^*)
- Include i^* 's commitment and α share

Signature: $\sigma = (\text{salt}, h_1, h_2, \{\text{proofs}\}_{\tau=1}^T)$

Code: run_phase_1() – Commit & Simulate

```
1 static void run_phase_1(...) {
2   gf_from_bytes(pt_gf, sk);
3   aim2_sbox_outputs(sbox_outputs, pt_gf);
4   aim2_generate_linear(matrix_A, vector_b,
5     sk + AIM2_NUM_BYTES_FIELD);
6   // generate salt and root seeds
7   hash(sk, mu, rnd) -> salt, root_seeds
8
9   for (rep = 0; rep < AIMER_T; rep++) {
10    expand_tree(nodes[rep], salt, rep,
11      root_seeds[rep]);
12    memset(&delta, 0, sizeof(tape_t));
13
14    for (party = 0; party < AIMER_N;
15      party++) {
16      commit_and_expand_tape(&tape,
17        commits[rep][party], salt,
18        rep, party, nodes[rep][...]);
19      // accumulate offsets
20      gf_add(delta.pt_share, ..., ...);
21      gf_add(delta.t_shares[i], ..., ...);
22
23      // last party: adjust shares
24      if (party == AIMER_N - 1) {
25        delta.pt = delta.pt + pt_gf;
26        delta.t[i] += sbox_outputs[i];
27      }
28      // run MPC on each party's view
29      aim2_mpc(&mult_chk[rep][party],
30        matrix_A, ct_gf);
31    }
32    // hash all commits -> h_1
33  }
34 }
```

Heart of the signing algorithm.

Lines 2–5: Precompute witness

- S-box outputs $t_i = (\text{pt} + \gamma_i)^{(2^{e_i} - 1) - 1}$
- Linear layer matrix A from IV

Lines 10–12: For each repetition τ :

- Expand GGM tree: root seed $\rightarrow N$ party seeds

Lines 16–23: For each party:

- CommitAndExpand: seed \rightarrow (commitment, tape)
- Accumulate Δ = sum of all random shares

Lines 26–30: Last party correction

- $\Delta \text{pt} = \bigoplus_j \text{pt}^{(j)} \oplus \text{pt}$, $\Delta t_i = \bigoplus_j t_i^{(j)} \oplus t_i$
- Ensures shares reconstruct to real witness

Line 32: Run aim2_mpc per party \Rightarrow next slide

Code: aim2_mpc() – MPC Circuit Evaluation

```
1 void aim2_mpc(mult_chk_t *mult_chk,  
2   const gf matrix_A[AIMER_L][NUM_BITS],  
3   const gf ct_gf)  
4 {  
5   // First AIMER_L S-boxes:  
6   // t^{2^e} + t*ct = x*pt --> z = x*pt  
7   for (size_t ell = 0; ell < AIMER_L; ell++)  
8   {  
9     gf_sqr(mult_chk->z_shares[ell],  
10      mult_chk->x_shares[ell]);  
11     for (size_t i = 1;  
12          i < aim2_exponents[ell]; i++)  
13       gf_sqr(mult_chk->z_shares[ell],  
14        mult_chk->z_shares[ell]);  
15     gf_mul_add(mult_chk->z_shares[ell],  
16      mult_chk->x_shares[ell],  
17      aim2_constants[ell]);  
18     gf_mat_vec_mul_add(  
19      mult_chk->x_shares[AIMER_L],  
20      mult_chk->x_shares[ell],  
21      matrix_A[ell]);  
22   }  
23   // Final S-box:  
24   // x^{2^e} + x*ct = x*pt --> z = x*pt  
25   gf_sqr(mult_chk->z_shares[AIMER_L],  
26    mult_chk->x_shares[AIMER_L]);  
27   for (size_t i = 1;  
28        i < aim2_exponents[AIMER_L]; i++)  
29     gf_sqr(mult_chk->z_shares[AIMER_L],  
30      mult_chk->z_shares[AIMER_L]);  
31   gf_mul_add(mult_chk->z_shares[AIMER_L],  
32    mult_chk->x_shares[AIMER_L], ct_gf);  
33 }
```

Core of the ZK proof: evaluates AIM2 circuit on secret-shared inputs.

Reformulated equations:

Instead of $t_i = (pt + \gamma_i)^{(2^{e_i}-1)^{-1}}$, use:

$$t_i \cdot pt = t_i^{2^{e_i}} + \gamma_i \cdot t_i$$

This is a **multiplication check**: $x \cdot pt = z$

Lines 7–22: For each S-box branch:

1. Compute $t_i^{2^{e_i}}$ via repeated squaring
2. Add $t_i \cdot \gamma_i$ term
3. Accumulate into linear layer input

Lines 24–33: Final S-box: same pattern

$$y \cdot pt = y^{2^{e^*}} + y \cdot ct$$

Key insight: Squarings are \mathbb{F}_2 -linear \Rightarrow **free** in MPC (no interaction needed). Only the multiplication $x \cdot pt$ requires a multiplication triple.

Code: Verification Overview

```
1 int crypto_sign_verify_internal(  
2     const uint8_t *sig, ...,  
3     const uint8_t *pk)  
4 {  
5     const signature_t *sign = ...;  
6     aim2_generate_linear(matrix_A,  
7         vector_b, pk);  
8  
9     // Recompute challenges  
10    indices = Expand(sign->h_2);  
11    epsilons = Expand(sign->h_1);  
12  
13    for (rep = 0; rep < AIMER_T; rep++) {  
14        i_bar = indices[rep];  
15        reconstruct_tree(nodes, salt,  
16            sign->proofs[rep].reveal_path,  
17            rep, i_bar);  
18  
19        for (party = 0; party < AIMER_N;  
20            party++) {  
21            if (party == i_bar) {  
22                // use missing_commitment  
23                continue;  
24            }  
25            commit_and_expand_tape(...);  
26            if (party == AIMER_N - 1) { ... }  
27            aim2_mpc(&mult_chk, matrix_A, ct);  
28            // accumulate alpha, v  
29        }  
30    }  
31    return memcmp(h_1', h_1) ||  
32        memcmp(h_2', h_2) ? -1 : 0;  
33 }
```

Mirrors signing, but reconstructs from signature.

Lines 6–11: Setup

- Regenerate Lin from pk
- Recompute challenges from h_1, h_2

Lines 14–17: For each repetition:

- Reconstruct GGM tree from copath
- Recover $N - 1$ party seeds (except i^*)

Lines 19–29: For each party:

- Party i^* : use commitment & α -share from σ
- Others: regenerate tape, re-run aim2_mpc

Lines 32–33: Final check

- Recompute h'_1, h'_2 from reconstructed views
- Accept iff $h'_1 = h_1 \wedge h'_2 = h_2$

Code: aim2() — OWF Evaluation

```
1 void aim2(uint8_t ct[], const uint8_t pt[],
2           const uint8_t iv[])
3 {
4     gf matrix_L[AIMER_L][NUM_BITS_FIELD];
5     gf matrix_U[AIMER_L][NUM_BITS_FIELD];
6     gf vector_b = {0,};
7     gf state[AIMER_L];
8     gf pt_gf = {0,}, ct_gf = {0,};
9     gf_from_bytes(pt_gf, pt);
10
11     generate_matrices_L_and_U(
12         matrix_L, matrix_U, vector_b, iv);
13
14     for (size_t i = 0; i < AIMER_L; i++) {
15         gf_add(state[i], pt_gf,
16               aim2_constants[i]);
17         gf_exp(state[i], state[i],
18               aim2_sbox_exponents[i]);
19         gf_mat_vec_mul(state[i], state[i],
20               matrix_U[i]);
21         gf_mat_vec_mul(state[i], state[i],
22               matrix_L[i]);
23     }
24     for (size_t i = 1; i < AIMER_L; i++)
25         gf_add(state[0], state[0], state[i]);
26     gf_add(state[0], state[0], vector_b);
27
28     gf_exp(state[0], state[0],
29           aim2_sbox_exponents[AIMER_L]);
30     gf_add(ct_gf, state[0], pt_gf);
31     gf_to_bytes(ct, ct_gf);
32 }
```

AIM2 One-Way Function

Lines 4–12: Setup

- Parse plaintext pt from bytes
- Generate Lin matrices from IV via XOF
- L, U : LU-decomposed form of Lin

Lines 14–23: First S-box layer ($\times \ell$)

- gf_add: $\text{state}_i = \text{pt} + \gamma_i$
- gf_exp: $\text{state}_i^{(2^{e_i} - 1)^{-1}}$
- gf_mat_vec_mul: apply Lin

Lines 24–26: Sum branches + bias b

$$y = \sum_i \text{Lin}_i(t_i) + b$$

Lines 28–31: Final S-box + feedforward

$$\text{ct} = y^{2^{e^*} - 1} \oplus \text{pt}$$

Used in:

- Key generation (compute ct from pt, iv)
- Not called during signing/verifying (MPC version used instead)

Code: Linear Layer Generation

```
1 static void generate_matrices_L_and_U(  
2     gf matrix_L[][NUM_BITS_FIELD],  
3     gf matrix_U[][NUM_BITS_FIELD],  
4     gf vector_b,  
5     const uint8_t iv[])  
6 {  
7     hash_instance ctx;  
8     hash_init(&ctx);  
9     hash_update(&ctx, iv, AIM2_IV_SIZE);  
10    hash_final(&ctx);  
11  
12    for (size_t ell = 0;  
13         ell < AIM2_NUM_INPUT_SBOX; ell++)  
14    {  
15        for (size_t row = 0;  
16             row < AIM2_NUM_BITS_FIELD; row++)  
17        {  
18            hash_squeeze(&ctx, buf,  
19                          AIM2_NUM_BYTES_FIELD);  
20            gf_from_bytes(temp, buf);  
21            // split into L (lower) and U (upper)  
22            // with 1 on diagonal  
23            size_t inter = row / 64;  
24            matrix_L[ell][row][inter] =  
25                (temp[inter] & lmask) | ormask;  
26            matrix_U[ell][row][inter] =  
27                (temp[inter] & umask) | ormask;  
28        }  
29    }  
30    hash_squeeze(&ctx, vector_b,  
31                AIM2_NUM_BYTES_FIELD);  
32 }
```

Generates the random linear layer from IV.

Lines 7–10: Initialize XOF with IV

- Uses SHAKE (hash-based XOF)
- IV is the **only** input \Rightarrow deterministic

Lines 12–29: For each S-box branch \times each row:

- Squeeze n bits from XOF
- Split into lower-triangular L and upper-triangular U matrices
- Diagonal forced to 1 (invertibility)

Lines 30–31: Squeeze bias vector b

LU decomposition:

The affine layer is stored as $A = L \cdot U$:

$$\text{Lin}(t_i) = L_i \cdot U_i \cdot t_i$$

This enables efficient matrix-vector multiplication during both OWF evaluation and MPC simulation.

Q&A

Attribution

- Illustrations was created using fontawesome latex package (<https://github.com/xdanaux/fontawesome-latex>).