

Hardware-Adapted SIMD Optimization of AIMer v3 on x86 and Cortex-M55

Seung-Won Lee¹[0009–0009–9583–1387], Si-Woo Eum¹[0000–0002–9583–5427],
Min-Seo Kim¹[0009–0008–1979–5778], Su-Been Cho¹[0009–0004–4352–8021], and
Hwa-Jeong Seo¹[0000–0003–0069–9061]

Department of Information Computer Engineering,
Hansung University, Seoul, 02876 Korea
dkajdfhd1@gmail.com, shuraatum@gmail.com, msqoo0212@gmail.com,
chosubin1208@gmail.com, hwajeong84@gmail.com

Abstract. AIMer v3 replaces AIM2 with AIM3. We test whether SIMD methods developed for AIM2 can accelerate AIM3 while preserving its results. We port the field-arithmetic, affine, party-parallel, and SHAKE methods of the earlier AVX-512 implementation to AIMer v3 and implement the same operations for AVX2. For Cortex-M55 MVE, we use 16-bit polynomial partial products for field multiplication, group corresponding 32-bit words from four parties, and vectorize both affine layers, including the input layer introduced in AIM3. The implementations preserve AIMer v3’s mathematical structure and proof procedures. For all six parameter sets, their outputs match the official known-answer tests and the reference results in differential tests. Relative to the reference on each platform, AVX2 and AVX-512 accelerate signing and verification by $3.14\text{--}4.41\times$ and $4.59\text{--}6.55\times$, respectively. Cortex-M55 MVE accelerates the same operations by $1.28\text{--}1.91\times$.

Keywords: AIMer v3 · AIM3 · AVX-512 · Cortex-M55 · MVE · Binary-Field Arithmetic

1 Introduction

Recently, cryptanalysis has shown that AIM2, the symmetric primitive underlying AIMer v2.1, does not achieve its claimed security level [3,13]. In response, AIMer replaces AIM2 with AIM3 [5]. Changes to the input processing and S-box structure alter the computation targeted by earlier optimized implementations[4].

AIMer constructs post-quantum signatures by proving knowledge of a preimage of a symmetric one-way function. Its MPC-in-the-Head proof simulates many virtual parties, repeatedly applying binary-field arithmetic and hashing[9]. This repetition creates both computational cost and opportunities for parallel execution. Therefore, binary-field arithmetic and independent-party parallelism remain central under AIM3. To exploit this parallelism, prior work optimized AIM2-based AIMer by processing multiple parties together with vector instructions[19]. An AVX-512 implementation exemplifies this approach [15]. However, AIM3

changes the computational structure, so the correctness and performance gains of implementations using these methods must be reassessed.

In addition, efficient implementations are needed beyond desktop processors. Long-lived embedded devices require post-quantum protection under tight processing and memory constraints; desktop speedups alone cannot establish efficiency in this setting. Cortex-M55 offers SIMD acceleration through MVE, but its narrower vectors, smaller register file, and smaller polynomial products prevent a direct translation of AVX-512 code[16]. Despite these constraints, adapting field arithmetic and party parallelization to the target hardware allows these methods to be extended to embedded devices. We preserve the mathematical structure and proof procedures of AIMer v3 while applying AIM2’s SIMD optimization techniques to AVX2 and AVX-512 implementations of AIM3 and adapting their field-arithmetic, affine, and party-parallel principles to Cortex-M55 MVE’s hardware characteristics.

This paper makes three contributions.

1. We adapt the earlier AVX-512 field, matrix, and Keccak methods to all six AIMer v3 parameter sets and implement the same scope with AVX2. This establishes their applicability after the AIM2-to-AIM3 transition.
2. We implement field arithmetic and party parallelization using 16-bit polynomial partial products and a four-party layout of 32-bit words tailored to the operation widths of Cortex-M55 MVE. In particular, we vectorize matrix-vector operations in AIM3’s newly introduced input affine layer as well as its output affine layer, extending existing x86 optimization principles to embedded implementations of AIM3.
3. Our ported implementations preserve AIMer v3’s mathematical structure and proof procedures, and we verify that their outputs match the official known-answer tests across all six parameter sets. They accelerate signing and verification by $3.14\text{--}4.41\times$ with AVX2, $4.59\text{--}6.55\times$ with AVX-512, and $1.28\text{--}1.91\times$ with Cortex-M55 MVE over their platform-local references, demonstrating that hardware-adapted implementations improve performance without changing the algorithm.

The remainder of this paper is organized as follows. Sections 2 and 3 introduce the necessary background and prior work. Section 4 explains the two implementations, Section 5 evaluates them, and Section 6 concludes.

2 Background

AIMer combines a symmetric one-way function with an MPC-in-the-Head proof. In AIMer v3, N virtual parties evaluate AIM3 over $\text{GF}(2^n)$ in each of τ repetitions. Table 1 gives the six parameter sets. The f sets prioritize speed, while the s sets use more parties and fewer repetitions to reduce signature size.

To address algebraic attacks on AIM2, AIM3 modifies its input processing and S-box structure as follows.

Table 1. AIMer v3 parameters: field-element bit length n , first-layer S-box count ℓ , S-box exponent parameters $(e_0, \dots, e_\ell)$, hash function, party count N , repetitions τ , and signature size.

Set	n (bits)	ℓ	$(e_0, \dots, e_\ell)$	Hash	N	τ	Signature (B)
128f	128	2	(3, 7, 11)	SHAKE128	16	33	6,944
128s	128	2	(3, 7, 11)	SHAKE128	256	17	4,704
192f	192	2	(11, 23, 47)	SHAKE256	16	49	15,408
192s	192	2	(11, 23, 47)	SHAKE256	256	25	10,320
256f	256	3	(3, 7, 11, 19)	SHAKE256	16	65	31,360
256s	256	3	(3, 7, 11, 19)	SHAKE256	256	33	20,224

First, AIM3 replaces the addition of fixed constants to the input with an IV-dependent input affine layer. By increasing the degree of the large-field equations used in the attack, this change is designed to counter the Magic Pot attack that exploits AIM2’s simple input structure[5].

Second, AIM3 replaces the inverse Mersenne S-boxes with enhanced Mersenne S-boxes that combine Mersenne exponentiation with a field-inverse term. The new S-boxes are not one-to-one, limiting the applicability of attacks that simplify equations by inverting the original S-boxes [5, Section 6.3.1].

2.1 AVX2 and AVX-512

AVX2 supports integer-vector operations using 256-bit YMM registers. Alongside AVX2, the separate PCLMULQDQ extension uses 128-bit XMM registers to compute a $64 \times 64 \rightarrow 128$ -bit carry-less product[8].

Meanwhile, AVX-512 provides 32 512-bit ZMM registers in 64-bit mode. VPCLMULQDQ computes independent partial products together across the four 128-bit lanes of a ZMM register. In addition, VPTERNLOGQ performs bitwise logic combining three inputs, while VPROLQ rotates the bits within each 64-bit data element [10].

These capabilities are not limited to 512-bit operations. AVX-512VL allows these operations on 128-bit XMM and 256-bit YMM registers, so shorter vectors can also benefit from the expanded instruction set and register count [17].

2.2 Cortex-M55 and MVE

Cortex-M55 is an Armv8.1-M-based processor that supports SIMD through the M-profile Vector Extension (MVE), also called Helium. MVE provides eight 128-bit Q registers, each of which can hold four 32-bit data elements.

For polynomial multiplication, MVE provides VMULLB.P16 and VMULLT.P16. These instructions multiply the 16-bit values at even and odd positions, respectively, each producing four 32-bit partial products. In addition, VAND and VEOR perform bitwise AND and XOR operations on vector data [1].

Compared with x86, however, MVE has a smaller polynomial multiplication width and fewer vector registers. Large-field arithmetic therefore requires decomposition into smaller partial products and an arrangement of intermediate values adapted to the hardware.

3 Related Work

High-speed implementations of AIMer have focused on reducing the cost of field arithmetic and linear layers, and on processing independent computations in parallel. This section reviews these implementations, with particular attention to the x86 SIMD methods examined in this work and vector optimization studies relevant to its embedded extension.

3.1 x86 SIMD implementations

At the primitive level, Minwoo Lee et al. combined overlapping Mersenne exponentiations and precomputed IV-dependent matrices for the original 128-bit AIM-I. As a result, the execution time in their Apple M2 benchmark decreased from 38,482 to 1,181 ms [14].

Beyond individual primitive operations, other studies have accelerated the complete AIMer signing and verification procedures. Kwon et al. optimized AIM2-based AIMer with carry-less field arithmetic, grouped party operations, transposed matrix accumulation, and parallel SHAKE. They used PCLMULQDQ on x86 and PMULL on AArch64, with four-way and two-way SHAKE, respectively. Signing improved by $3.11\text{--}3.78\times$ over the reference on Haswell; the SHAKE-optimized AArch64 variant achieved up to $2.59\times$ on Apple M4 Max [12].

Building on the AIM2 SIMD approach, the AVX-512 study used wider polynomial products, ternary affine accumulation, and AVX-512VL Keccak. On an Intel Core i7-1165G7, it accelerated signing by $1.60\text{--}1.84\times$ and verification by $1.58\text{--}1.86\times$ relative to AVX2 across six variants [15]. To obtain these speedups, the study extended the AVX2 implementation to AVX-512 by grouping independent party computations in wider vectors and using fewer instructions for repeated operations. Figures 1 and 2 illustrate the polynomial partial-product computation and affine accumulation, respectively.

First, GF128 multiplication uses VPCLMULQDQ. Its 512-bit form independently computes polynomial products for four pairs of 64-bit operands. To exploit this operation, four party values are packed into one 512-bit ZMM register. The AVX2 implementation requires four PCLMULQDQ instructions to obtain the same four partial products, whereas the AVX-512 implementation obtains them with a single VPCLMULQDQ instruction. Partial-product recombination and modular reduction then yield the final field product for each party [15].

Next, affine accumulation uses VPTERNLOGQ. This instruction evaluates a three-input bitwise logical function in a single instruction. The implementation supplies a mask derived from an input bit, the corresponding matrix row, and the current accumulator as its three inputs. The AVX2 implementation uses two

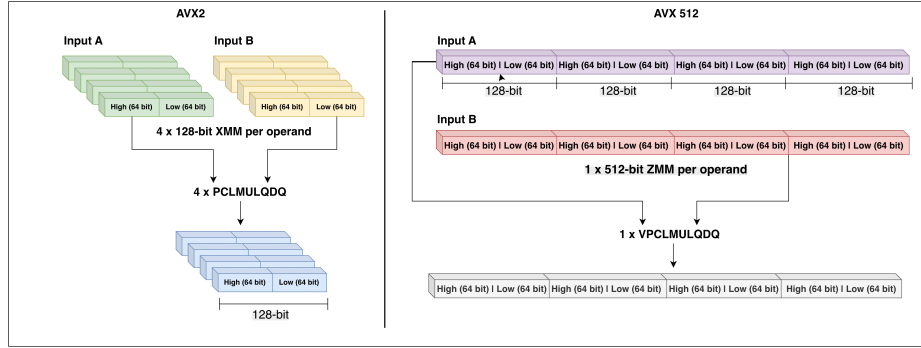


Fig. 1. Four-party GF128 multiplication on AVX2 and AVX-512. The illustrated 64×64 -bit partial-product step uses four PCLMULQDQ instructions on AVX2 and one VPCLMULQDQ on AVX-512. Partial-product recombination and modular reduction are performed independently for each party.

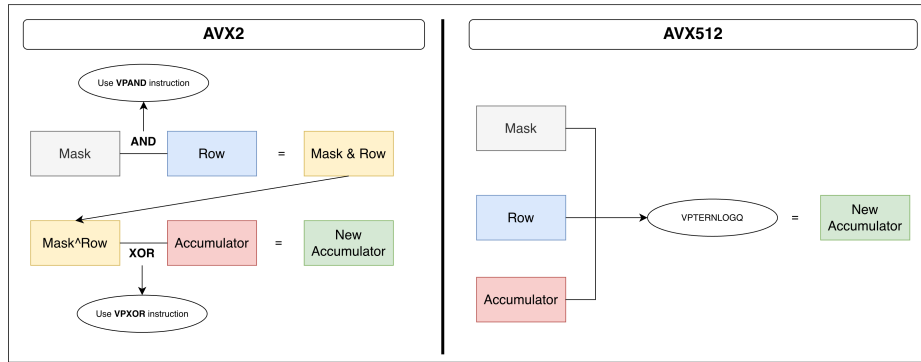


Fig. 2. One partial-product step of four-party GF128 multiplication. AVX2 uses four PCLMULQDQ instructions, whereas AVX-512 uses one VPCLMULQDQ instruction.

instructions: AND to select the row and XOR to accumulate the selected value. The AVX-512 implementation obtains the same updated accumulator with a single VPTERNLOGQ instruction [15].

In SHAKE, four independent Keccak states are processed together by placing corresponding 64-bit words in a 256-bit YMM register. AVX-512VL enables VPROLQ and VPTERNLOGQ on this layout, shortening the instruction sequences for rotations and Boolean operations [15].

These results establish end-to-end gains from the combined field, affine, and SHAKE optimizations for AIM2-based AIMer.

3.2 Embedded and MVE optimization

On resource-constrained devices, whether AIMer can run depends on the available memory. Kannwischer et al. ported the earlier AIMer reference to pqm4, but only three of its twelve parameter sets ran on their 640-KiB Cortex-M4 board [11]. Kwon et al. subsequently applied per-repetition streaming and assembly field routines to enable all six recommended AIM2-based parameter sets to run. Their speed-oriented implementation reused precomputed trees, reducing signing time by 16.75–39.01% compared with their memory-optimized implementation, but requiring more stack memory [12].

While these AIMer implementations target Cortex-M4, studies on Cortex-M55 MVE have explored polynomial arithmetic for lattice-based schemes. Becker et al. vectorized striding Toom–Cook/Karatsuba and NTT arithmetic for Saber, arranging coefficients and interleaving loads with arithmetic within eight Q registers. As a result, their M55 NTT matrix–vector routines achieved $3.4\text{--}3.5\times$ speedups over earlier Cortex-M4 implementations [2,18,20].

Taken together, the AIMer SIMD studies reviewed here evaluate AIM2-based implementations, while the M55 performance results discussed above concern lattice kernels. However, these studies do not address the implementation and evaluation of AIM3’s field, affine, and party computations on these SIMD targets[6,7]. Our work addresses this research gap by applying existing optimization principles to AIM3 and evaluating their effectiveness.

4 Implementation

This section describes how we apply the SIMD principles reviewed in Section 3 through AVX-512 adaptation, an AIM3-specific AVX2 implementation, and an extension to Cortex-M55 MVE.

4.1 x86 SIMD Implementation for AIMer v3

AVX-512 Adaptation Our AVX-512 implementation adapts the field arithmetic, party grouping, affine accumulation, and parallel SHAKE methods of the earlier AIM2 study [15]. We use the four-party GF128 layout described in Section 3 for repeated squaring in AIM3’s MPC evaluation. In doing so, we use the existing field multiplication and squaring kernels according to AIM3’s S-box and MPC evaluation sequences.

In addition, for GF192 and GF256, each input is represented by two 128-bit portions. GF192 zero-pads the upper 64 bits of the second portion. Four parties are still processed together: their first portions occupy one ZMM, and their second portions occupy another.

For affine operations, we use the party grouping and VPTERNLOGQ-based accumulation technique of the earlier AIM2 implementation, reviewed in Section 3 [15, Sec. 4.2]. We apply this technique not only to AIM3’s output affine layer but also to its newly introduced input affine layer, following AIM3’s MPC

procedure for the matrices and constants. Parties share the public matrix but retain their own input-bit masks, and affine constants are added only to the designated last party.

For GF128, four parties' 128-bit results fill a single 512-bit ZMM without unused space and are accumulated together. In contrast, the adapted GF192 kernel places one party's 192-bit result in a 256-bit YMM, using a masked load to read its three 64-bit words and zero the remaining 64 bits. This layout handles the existing 24-byte representation without a separate rearrangement across parties. Although GF192 uses YMM rather than the ZMM used for GF128, AVX-512VL allows the same VPTERNLOGQ-based affine accumulation technique at this width.

For GF256, the adapted kernel again uses ZMM registers: two parties' 256-bit results fill one register and are accumulated together.

AVX2 Implementation The AVX2 implementation follows the parallelization principles of the AVX-512 implementation described above, but adjusts the layout and evaluation order to the register width and available instructions. For GF128 squaring, PCLMULQDQ operates on 128-bit XMM registers. Instead of packing four parties into one 512-bit ZMM register, we place each party in a separate XMM register and interleave their partial-product calculations. For GF192 and GF256, each input's two 128-bit portions occupy separate XMM registers, and two parties are processed together.

For parallel affine evaluation, we use 256-bit YMM registers. GF128 therefore holds two party results per register instead of four. GF192 retains one YMM per party, while GF256 changes from two parties per ZMM to one party per YMM. Since VPTERNLOGQ is unavailable on AVX2, matrix data selection and accumulation use separate VPAND and VPXOR instructions.

For GF128, Algorithm 1 shows how four of these YMM accumulators process eight parties together. At each input-bit position, the same matrix data is copied into both 128-bit halves of a YMM and reused across the four accumulators. Each pair of parties supplies its own masks for the VPAND–VPXOR update. Repeating this step over all 128 input bits produces eight independent results. The algorithm shows the matrix–vector part of the affine calculation; the affine constant is handled separately.

Here, $x_p[j]$ is bit j of party p 's input, numbered from the least significant bit. Mask expands a bit to 128 all-zero or all-one bits. Pack128 places its two arguments in the lower and upper halves of a YMM, respectively, and Unpack128 returns the eight results in party order.

Parallel SHAKE For SHAKE, we reuse the existing parallel Keccak implementations [15, Sec. 4.3] to process four independent hash computations together. Corresponding 64-bit words from the four computations are packed into one 256-bit YMM register, with the same layout in AVX2 and AVX-512. We apply this approach to AIM3 commitment computation and the generation of pseudorandom data for each party.

Algorithm 1 Eight-party GF128 matrix–vector multiplication on AVX2

Require: Eight 128-bit inputs $x_0, \dots, x_7$;
shared 128-bit matrix entries $M[0], \dots, M[127]$
Ensure: Eight 128-bit matrix–vector products $r_0, \dots, r_7$

```

1:  $R_0, R_1, R_2, R_3 \leftarrow 0$ 
2: for  $j = 127, 126, \dots, 0$  do
3:    $D \leftarrow \text{Pack128}(M[j], M[j])$ 
4:   for  $k = 0, \dots, 3$  do
5:      $K \leftarrow \text{Pack128}(\text{Mask}(x_{2k}[j]), \text{Mask}(x_{2k+1}[j]))$ 
6:      $T \leftarrow \text{VPAND}(D, K)$ 
7:      $R_k \leftarrow \text{VPXOR}(R_k, T)$ 
8:   end for
9: end for
10: return  $\text{Unpack128}(R_0, R_1, R_2, R_3)$ 

```

Although the layout is the same, the instructions used for rotations and nonlinear operations differ. For a general 64-bit rotation, AVX2 shifts in both directions using VPSLLQ and VPSRLQ, then combines the results with VPOR. AVX-512VL performs the same rotation with a single VPROLQ instruction. AVX2 instead uses VPSHUFb for rotations by 8 and 56 bits. In the nonlinear step, AVX2 uses separate VPANDN and VPXOR instructions for each update, whereas AVX-512VL combines them into a single VPTERNLOGQ instruction.

4.2 Cortex-M55 MVE Implementation

The Cortex-M55 implementation adapts the field arithmetic, party parallelization, and affine accumulation principles used on x86 to MVE. Since MVE uses 128-bit Q registers and 16-bit polynomial multiplication instructions, we divide field elements into smaller pieces for computation and group data at corresponding positions from four parties.

Four-party layout. We retain the x86 principle of processing independent parties together, but change the layout to fit MVE’s 128-bit Q registers. For GF128, AVX-512 places four 128-bit party values in one 512-bit ZMM register. On MVE, we divide each party’s value into 32-bit words and place the corresponding words from four parties in one 128-bit Q register.

A GF128 value consists of four 32-bit words, so this layout represents the complete data of four parties with four 128-bit vectors. The same arrangement uses six vectors for GF192 and eight for GF256. We apply this layout in AIM3’s MPC evaluation to perform field operations in parallel across the four parties represented in each vector.

Polynomial field arithmetic. General field multiplication uses VMULLB.P16 and VMULLT.P16. We divide both inputs into 16-bit pieces, replicate one piece

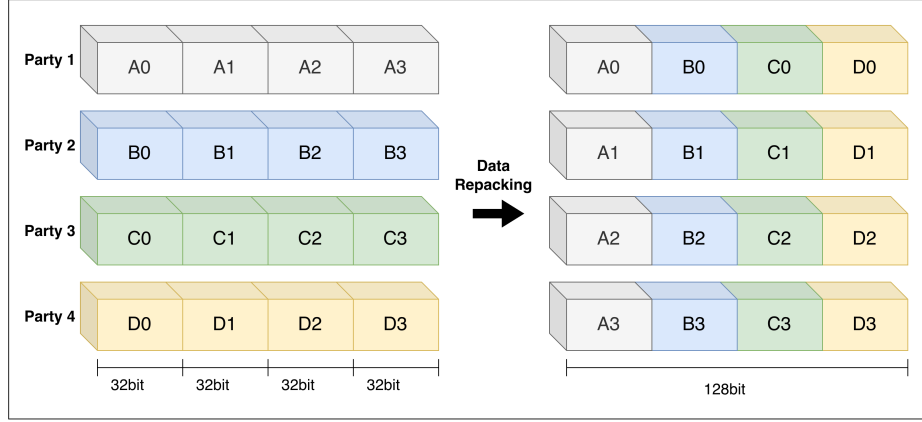


Fig. 3. Word-wise repacking of four GF128 party values for MVE.

from the first input across a Q register, and multiply it by eight pieces from the second input. Each instruction independently multiplies four pairs of 16-bit polynomials to produce four 32-bit partial products, so the two instructions together compute eight partial products [1]. We repeat this process over the input pieces and combine the partial products with XOR at their corresponding positions, then apply scalar modular reduction to obtain the field product.

For repeated squaring in AIM3’s MPC evaluation, we use the four-party layout described above. In a binary-field square, cross terms cancel under XOR, so the squares of the individual 16-bit pieces suffice to construct the full polynomial square. Algorithm 2 describes the four-party repeated-squaring procedure on MVE.

Here, word_j indexes 32-bit words from the least significant end; Halfwords views each vector as 16-bit pieces without rearrangement. Reduce4 performs modular reduction with P16 multiplication, shifts, and XOR in the same party layout, and Unpack32 reconstructs the four outputs.

Vector affine accumulation. Following the AVX2 method described above, affine operations select matrix data according to each input bit and accumulate it with XOR. An input bit of zero produces an all-zero mask, while a bit of one produces an all-ones mask. MVE’s VAND uses this mask to select either the matrix data or zero, and VEOR accumulates the selected value into the result [1]. Repeating this process for every input bit computes the matrix–vector product.

In AIM3’s MPC evaluation, we use VAND and VEOR with the four-party layout described above to compute the parties’ matrix–vector products in parallel, using each party’s input bits.

In summary, the x86 and Cortex-M55 implementations apply shared principles of field arithmetic, affine computation, and party parallelization to their respective

Algorithm 2 Four-party repeated squaring on Cortex-M55 MVE**Require:** Four inputs $a_0, \dots, a_3 \in \text{GF}(2^n)$, $n \in \{128, 192, 256\}$;squaring count $e \geq 0$ and field modulus f_n **Ensure:** $b_p = a_p^{2^e} \bmod f_n$ for each party $p \in \{0, 1, 2, 3\}$

```

1:  $m \leftarrow n/32$ 
2: for  $j = 0, \dots, m-1$  do
3:    $Q_j \leftarrow (\text{word}_j(a_0), \text{word}_j(a_1), \text{word}_j(a_2), \text{word}_j(a_3))$ 
4: end for
5: for  $t = 1, \dots, e$  do
6:   for  $j = 0, \dots, m-1$  do
7:      $H \leftarrow \text{Halfwords}(Q_j)$ 
8:      $P_{2j} \leftarrow \text{VMULLB.P16}(H, H)$ 
9:      $P_{2j+1} \leftarrow \text{VMULLT.P16}(H, H)$ 
10:   end for
11:    $Q \leftarrow \text{Reduce4}(P, f_n)$ 
12: end for
13: return  $\text{Unpack32}(Q)$ 

```

hardware. On x86, we adapted existing AVX-512 techniques to AIM3 and developed an AVX2 implementation covering the same operations. On Cortex-M55, we extended these shared principles to AIM3’s computations and implemented them using MVE instructions and data layouts. These implementations preserve the mathematical structure and proof procedures of AIMer v3. The next section evaluates their performance against the reference implementation on each platform.

5 Performance

This section evaluates the effect of SIMD optimization on AIMer v3 key generation, signing, and verification. For both x86 and Cortex-M55, we compare end-to-end performance against the platform’s reference implementation and also analyze the performance of the targeted kernels.

5.1 Experimental Setup

Table 2 summarizes the environments. On x86, measurements were pinned to a fixed core and run at a fixed frequency. On Cortex-M55, code ran without an operating system, with the same memory layout conditions and low-memory execution schedule for the reference and MVE implementations. Because the platforms use different cycle counters, performance gains are evaluated against the reference implementation on each platform.

End-to-end performance was measured in seven independent runs, with 50 samples per parameter set, implementation, and operation in each run. The reported cycle count is the median of the seven run medians. The coefficient of variation (CV) is the sample standard deviation of these seven medians divided by their mean, expressed as a percentage.

Table 2. Measurement conditions.

	x86	Cortex-M55
Processor	Core i7-1165G7	STM32N657 NUCLEO
Clock	Fixed 2.8 GHz; Turbo off	Fixed 600 MHz
Execution	Pinned CPU; kernel run with sibling offline	Single-core bare metal
Compiler	GCC 13.3.0, -O3	GCC 15.2.1, -O3
Additional flags	-fomit-frame-pointer	Auto-vectorization off
E2E	7 runs; 50 samples/run; 10 warm-ups	7 runs; 50 samples/run; 10 warm-ups
Kernels	15 runs; 75 samples/run; 10 warm-ups	31 samples; PMU counter
Timer	Serialized invariant TSC	CPU-clocked PMU/SysTick

5.2 x86 Results

After adapting AIM2’s SIMD optimization methods to AIM3, we confirmed that the AVX2 and AVX-512 implementations reproduced the outputs of all 100 official AIMer v3 known-answer test (KAT) vectors byte for byte for each of the six parameter sets. Differential tests of field arithmetic, affine operations, and party batch operations also produced the same results as the reference implementation. These results provide experimental evidence that the existing SIMD methods can be applied to AIM3 while preserving its computational results.

In addition, Tables 3 and 4 show that the earlier x86 methods remain effective after the AIM2-to-AIM3 transition. AVX2 accelerates signing and verification by $3.14\text{--}4.41\times$, while AVX-512 reaches $4.59\text{--}6.55\times$. Relative to AVX2, AVX-512 is $1.43\text{--}1.52\times$ faster for signing and verification. Key generation gains less because it lacks the large party dimension available to the MPC-in-the-Head proof.

5.3 Cortex-M55 Results

The MVE implementation passed all 600 official KAT vectors on the Cortex-M55 board, with 100 vectors for each of the six parameter sets.

As shown in Table 5, the methods described in Section 4.2 accelerate key generation by $1.50\text{--}1.85\times$, signing by $1.28\text{--}1.91\times$, and verification by $1.38\text{--}1.87\times$ over the portable reference across all six parameter sets. These end-to-end gains show that the field, affine, and party-parallel principles used on x86 remain effective when adapted to MVE.

5.4 Kernel-Level Results

Figure 4 shows how kernel speedups vary with field size. GF and affine cover single-input kernels, whereas Party/MPC evaluates the combined effect of field arithmetic, affine operations, and party parallelization described in Section 4.

Table 3. AVX2 end-to-end results in millions of invariant-TSC ticks. Parentheses give the CV in percent.

Set	Operation	Reference	AVX2	Speedup ($\times$)
128f	Key generation	0.535 (1.97)	0.217 (2.41)	2.453
128s	Key generation	0.561 (1.94)	0.235 (2.62)	2.417
192f	Key generation	1.381 (2.41)	0.628 (1.22)	2.206
192s	Key generation	1.430 (1.31)	0.648 (2.64)	2.204
256f	Key generation	3.740 (2.87)	1.712 (0.46)	2.186
256s	Key generation	3.827 (0.87)	1.756 (1.21)	2.178
128f	Signing	6.661 (3.65)	2.070 (3.24)	3.220
128s	Signing	51.178 (2.27)	15.950 (1.41)	3.214
192f	Signing	20.021 (2.49)	5.308 (2.34)	3.770
192s	Signing	154.040 (0.51)	39.528 (2.53)	3.879
256f	Signing	42.658 (2.82)	10.392 (0.31)	4.104
256s	Signing	317.747 (0.57)	73.607 (1.49)	4.324
128f	Verification	6.203 (3.42)	1.973 (3.03)	3.142
128s	Verification	51.550 (2.26)	15.854 (1.29)	3.246
192f	Verification	18.541 (2.26)	5.174 (2.17)	3.580
192s	Verification	154.063 (0.69)	39.322 (2.96)	3.897
256f	Verification	39.109 (2.82)	10.113 (0.21)	3.868
256s	Verification	318.911 (1.07)	72.277 (1.60)	4.412

Table 4. AVX-512 end-to-end results in millions of invariant-TSC ticks. Parentheses give the CV in percent.

Set	Operation	Reference	AVX-512	Speedup ($\times$)	
				vs. Reference	vs. AVX2
128f	Key generation	0.535 (1.97)	0.199 (1.00)	2.680	1.092
128s	Key generation	0.561 (1.94)	0.210 (1.66)	2.668	1.119
192f	Key generation	1.381 (2.41)	0.431 (2.73)	3.212	1.457
192s	Key generation	1.430 (1.31)	0.446 (2.57)	3.151	1.453
256f	Key generation	3.740 (2.87)	1.170 (2.74)	3.197	1.464
256s	Key generation	3.827 (0.87)	1.205 (1.57)	3.170	1.457
128f	Signing	6.661 (3.65)	1.437 (0.20)	4.633	1.440
128s	Signing	51.178 (2.27)	10.857 (0.26)	4.723	1.469
192f	Signing	20.021 (2.49)	3.674 (3.49)	5.447	1.445
192s	Signing	154.040 (0.51)	27.493 (1.59)	5.611	1.438
256f	Signing	42.658 (2.82)	6.905 (3.04)	6.183	1.505
256s	Signing	317.747 (0.57)	49.547 (1.72)	6.411	1.486
128f	Verification	6.203 (3.42)	1.350 (0.79)	4.588	1.461
128s	Verification	51.550 (2.26)	10.863 (0.58)	4.746	1.460
192f	Verification	18.541 (2.26)	3.561 (3.40)	5.205	1.453
192s	Verification	154.063 (0.69)	27.462 (1.57)	5.594	1.432
256f	Verification	39.109 (2.82)	6.634 (2.71)	5.898	1.524
256s	Verification	318.911 (1.07)	48.687 (1.63)	6.547	1.485

Table 5. Cortex-M55 end-to-end results in millions of CPU cycles. Parentheses give the CV in percent.

Set	Operation	Reference	MVE	Speedup ($\times$)
128f	Key generation	5.111 (0.0008)	3.398 (0.0008)	1.504
128s	Key generation	5.114 (0.0005)	3.401 (0.0029)	1.504
192f	Key generation	14.734 (0.0010)	9.394 (0.0011)	1.568
192s	Key generation	14.741 (0.0020)	9.405 (0.0028)	1.567
256f	Key generation	44.617 (0.0033)	24.076 (0.0034)	1.853
256s	Key generation	44.618 (0.0038)	24.065 (0.0052)	1.854
128f	Signing	109.601 (0.0093)	84.542 (0.0192)	1.297
128s	Signing	865.809 (0.0068)	675.338 (0.0067)	1.282
192f	Signing	332.600 (0.0749)	218.566 (0.0257)	1.522
192s	Signing	2483.245 (0.0320)	1674.099 (0.0201)	1.484
256f	Signing	919.164 (0.1450)	510.380 (0.0666)	1.801
256s	Signing	7295.376 (0.2133)	3819.631 (0.0546)	1.911
128f	Verification	62.705 (0.0049)	45.421 (0.0022)	1.381
128s	Verification	511.841 (0.0021)	369.991 (0.0053)	1.383
192f	Verification	195.722 (0.0950)	125.417 (0.0292)	1.561
192s	Verification	1497.803 (0.0227)	986.513 (0.0031)	1.518
256f	Verification	550.743 (0.1529)	318.723 (0.0577)	1.728
256s	Verification	4535.277 (0.0946)	2424.624 (0.0074)	1.871

On x86, the GF speedups increase with field size, while the AVX2 and AVX-512 curves remain close. The difference between the two backends is more apparent in affine and party/MPC computation, where AVX-512 provides higher speedups at all three field sizes. For single-input affine operations, the AVX2 speedup decreases from $2.738\times$ at GF128 to $1.248\times$ at GF256, whereas AVX-512 retains a $2.376\times$ speedup at GF256.

On Cortex-M55, GF operations are accelerated by $3.014\text{--}3.192\times$ across the three field sizes. In contrast, the affine speedup is $1.136\times$ at GF128 and $1.156\times$ at GF192, but rises to $1.699\times$ at GF256.

When field and affine operations are used together in the four-party MPC path, the Cortex-M55 speedups reach $2.306\text{--}2.833\times$. These results measure the combined implementation, rather than the effect of party grouping alone.

For the SHAKE kernels, which are not included in the figure, the geometric-mean speedups across the measured x1 and x4 paths are $2.279\times$ for AVX2 and $4.462\times$ for AVX-512. Cortex-M55 retains the reference SHAKE implementation.

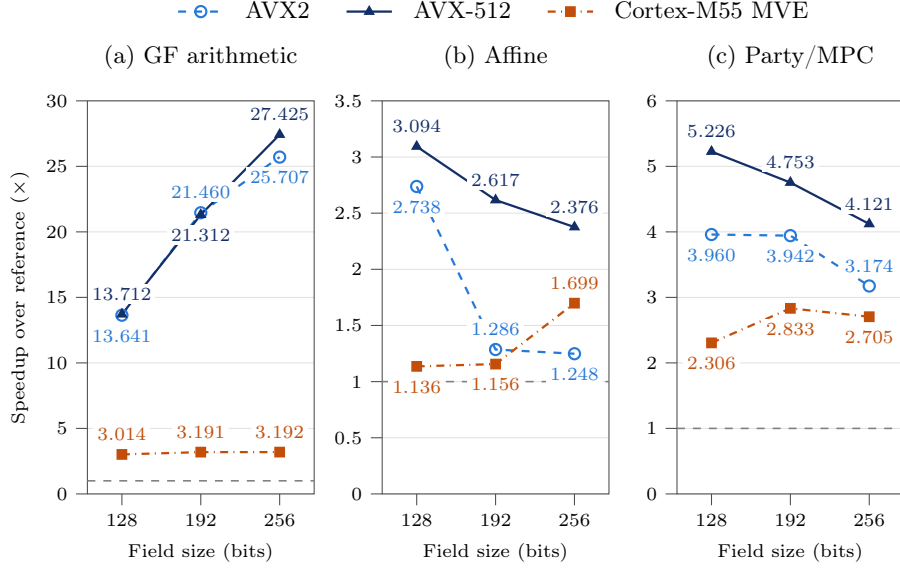


Fig. 4. Kernel speedups over each platform’s reference by field size. Points are geometric means across the f/s variants; GF also groups multiplication, squaring, and inversion. Affine includes single-input matrix–vector multiplication only. MPC batch sizes are 16/256 parties on x86 and four on MVE. Panel scales differ; absolute cycles are not compared across platforms.

6 Conclusion

The measured gains extend beyond individual kernels to complete AIMER v3 key generation, signing, and verification, while correctness checks provide experimental evidence that the reference outputs are preserved. These results show the practical value of reusing computational principles while allowing each implementation to use different instructions and data layouts.

The evaluation is limited to one x86 processor and one Cortex-M55 board, and M55 SHAKE uses the same implementation as the reference. The correctness tests do not establish side-channel resistance. Future work includes evaluation on additional processors and boards, SHAKE optimization suited to MVE, and side-channel analysis to assess the security of the optimized implementations beyond functional correctness.

References

1. Arm: Arm MVE Intrinsics, 2021q4 edn. (Jan 2022), https://arm-software.github.io/acle/mve_intrinsics/mve.html, accessed September 9, 2026
2. Becker, H., Mera, J.M.B., Karmakar, A., Yiu, J., Verbauwhede, I.: Polynomial multiplication on embedded vector architectures. Cryptology ePrint Archive (2021)

3. Biryukov, A., García Fernández, P., Udovenko, A.: Magic pot: Cryptanalysis of full aim2 in the standard and related-/reused-key settings using new elimination framework. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 63–90. Springer (2026)
4. Canright, D.: A very compact s-box for aes. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 441–455. Springer (2005)
5. Cho, J., Ha, J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, J., Lee, S., Moon, D., Son, M., et al.: The aimer signature scheme (2024)
6. De Figueiredo, L.H., Stolfi, J.: Affine arithmetic: concepts and applications. Numerical algorithms **37**(1), 147–158 (2004)
7. Deschamps, J.P.: Hardware implementation of finite-field arithmetic. McGraw-Hill, Inc. (2009)
8. Guide, P.: Intel® 64 and ia-32 architectures software developer’s manual. Volume 3B: system programming guide, Part 2(11), 0–40 (2011)
9. Heath, D.: Mpc in the head (2025)
10. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2: Instruction Set Reference, A–Z (2026), <https://cdrdv2-public.intel.com/922478/325383-092-sdm-vol-2abcd.pdf>, revision 092; accessed September 9, 2026
11. Kannwischer, M.J., Krausz, M., Petri, R., Yang, S.Y.: pqm4: Benchmarking nist additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive (2024)
12. Kwon, J., Lee, S., Lee, B., Seo, H., Cho, J.: Efficient implementations of aimer post-quantum signature scheme for low-end to high-end iot devices. IEEE Internet of Things Journal (2025)
13. Lee, J., Cho, A.J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, S., Moon, D., Son, M., Yoon, H., et al.: The aimer signature scheme (2024)
14. Lee, M., Jang, K., Kwon, H., Sim, M., Song, G., Seo, H.: High-speed implementation of aim symmetric primitives within aimer digital signature. Cryptology ePrint Archive (2023)
15. Lee, S.W., Eum, S.W., Seo, H.J.: Accelerating the aimer post-quantum signature with avx-512: A field-keccak speedup analysis. Cryptology ePrint Archive, Paper 2026/1417 (2026), <https://eprint.iacr.org/2026/1417>
16. Marsh, J.: Arm Helium Technology: M-Profile Vector Extension (MVE) for Arm Cortex-M Processors: Reference Book. Arm Education Media (2021), <https://www.arm.com/resources/education/books/mve-reference-book>
17. Ozturk, E., Zimmerman, D., Yap, K., Cornu, M., Kantecki, T.: Post-quantum cryptography: Accelerating Open Quantum Safe library with Intel AVX-512 Keccak 1600 implementation. Technology Guide 367250-001US, Intel Corporation (2025), <https://builders.intel.com/solutionslibrary/post-quantum-cryptography-accelerating-open-quantum-safe-library-with-intel-avx-512-keccak-1600-implementation>
18. Satriawan, A., Mareta, R., Lee, H.: A complete beginner guide to the number theoretic transform (ntt). Cryptology ePrint Archive (2024)
19. Sim, M., Eum, S., Song, G., Lee, M., Kim, S., Song, M., Seo, H.: Kpqclean ver2: Comprehensive benchmarking and analysis of kpqc algorithm round 2 submissions. Cryptology ePrint Archive (2024)
20. Zhu, Y., Liu, Z., Pan, Y.: When ntt meets karatsuba: preprocess-then-ntt technique revisited. In: International conference on information and communications security. pp. 249–264. Springer (2021)