Efficient SIMD Implementation of the BLS Signature Scheme Using Intel AVX-512

Ganqin Liu[1], Hao Cheng[1,2(✉)], Georgios Fotiadis[3], Jipeng Zhang[4] and Johann Großschädl[5]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China
ganqin.liu@mail.sdu.edu.cn
[2] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China
hao.cheng@sdu.edu.cn
[3] Digital Security and Trusted Computing Group, Ubitech Ltd., Athens, Greece
gfotiadis@ubitech.eu
[4] National University of Singapore, Singapore, Singapore
jp-zhang@outlook.com
[5] DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
johann.groszschaedl@uni.lu

Abstract. The BLS digital signature scheme, in particular its instantiation with the BLS12-381 curve, has become a cornerstone of modern blockchain protocols such as Ethereum Proof-of-Stake, due to its unique and attractive characteristics (e.g., support for non-interactive signature aggregation). Recently, Cheng et al. (CHES 2025) demonstrated that the enormous Single-Instruction-Multiple-Data (SIMD) computing power of the Intel AVX-512 extensions, when combined with carefully-designed vectorization strategies, can be effectively leveraged to speed up the computation of the optimal ate pairing on BLS12-381, a major component of BLS. This naturally raises the question of whether such SIMD-parallel processing can be exploited more extensively to benefit the entire BLS signature scheme. The present paper answers this question positively by presenting a highly SIMD-optimized BLS implementation using Intel AVX-512, especially the AVX-512IFMA instructions. In order to harness AVX-512 more efficiently for the performance-critical operations of BLS, we explored a wide range of optimization options, including various formulas and vectorization granularities for elliptic curve arithmetic operations, scalar multiplication, and hash-to-curve, as well as the fine-tuning and flexible use of different implementations of the finite-field arithmetic. Benchmarking results collected on an Intel Core i3-1005G1 ("Ice Lake") CPU show that our vectorized BLS software using AVX-512 is at least 1.57 times faster than an x64 assembly implementation of the widely-used `blst` library.

Keywords: Pairing-based cryptography, BLS signatures, SIMD, AVX-512

1 Introduction

The Boneh-Lynn-Shacham (BLS) digital signature [BLS01] was one of the first *pairing-based* schemes and, along with Boneh-Franklin's identity-based encryption [BF01] and Joux's tripartite Diffie-Hellman [Jou04], triggered the notion of *pairing-based cryptography*. The BLS scheme has attracted substantial interest for practical applications due to its attractive characteristics. In particular, the use of elliptic curves allows for short public keys and signatures (both are points in elliptic curve groups), comparable to established elliptic curve signature schemes, such as ECDSA [FIP23]. Besides the small keys and signatures,

it supports signature aggregation, a unique feature that allows multiple signatures to be combined into a single short signature, whose verification implies the validity of each individual signature. The fact that BLS is the *only* signature scheme that offers efficient, non-interactive signature aggregation increased its popularity, and nowadays the scheme enjoys widespread commercial adoption, especially in the blockchain sector. Ethereum is a renowned blockchain ecosystem that integrates BLS into its Proof-of-Stake design [Edg23]. In such large-scale networks with thousands of validators, the non-interactive aggregation property offers significant benefits, allowing one to aggregate a large amount of signatures on attested Ethereum blocks generated by different validators. This drastically reduces the number of exchanged messages across the network (bandwidth usage), as well as the cost of verifying the integrity of these messages [Edg23]. In addition, there are other blockchain networks that have also adopted BLS into their stack, such as Chia [Eng25] and Filecoin[1].

Despite sharing the foundation of elliptic curves, BLS is computationally more expensive than ECDSA. For example, [BBG+25] states that when instantiated with the BLS12-381 curve, BLS is 10 times slower than ECDSA for signing and around 30 times slower for verification, at 128-bit security. This is because BLS employs more expensive subroutines than the scalar multiplication used in ECDSA. The hash-to-curve map is one such subroutine, used in signing and verification, which maps a message of arbitrary size to a point in an elliptic curve group. Its performance is determined by costly operations, such as quadratic residuosity tests and square root extraction in finite fields, as well as cofactor clearing. Methods for optimizing hash-to-curve, while ensuring resistance against timing attacks, have been studied, e.g., in [WB19, CRT22, BP22]. Another costly operation in BLS verification is the pairing, whereby many efficient implementations use the optimal ate pairing [Ver10] on BLS12-381 [Bow17] as the standard secure instantiation. Verification requires the computation of two pairings and checking whether they are equal (valid signature) or not (invalid signature). Throughout the years, there has been extensive work on accelerating pairings, focusing especially on the BLS12-381 pairing, with state-of-the-art libraries such as `RELIC` [AGM+], Longa's implementation [Lon23], and `blst`[2].

Motivation. The attractive characteristics of BLS, in particular the native support for non-interactive signature aggregation, drive a strong demand for its acceleration. Although the cryptographic landscape is shifting towards Post-Quantum (PQ) cryptography, the replacement of digital signature schemes is not governed by the same urgency as key exchange or public-key encryption. Unlike confidentiality mechanisms, which are vulnerable to "store now, decrypt later" attacks by future quantum-enabled adversaries, authentication primitives are immune to retroactive compromise; they require security only at the time of verification. Furthermore, the currently NIST-standardized PQ schemes do not offer a satisfactory replacement for BLS in terms of aggregation support and bandwidth efficiency. Consequently, BLS signatures are expected to remain the dominant solution in the market for a considerable period before a viable PQ transition for aggregation becomes available. In addition, the acceleration of BLS operations yields immediate and sustained benefits, which extend beyond the signature scheme itself to other pairing-based constructions such as KZG [KZG10] and Groth16 [Gro16] zkSNARKs. The recent work in [CFGP25] demonstrated that the Intel AVX-512 (especially AVX-512IFMA) vector extensions, when combined with carefully-designed vectorization strategies, can significantly accelerate the computation of the optimal ate pairing on BLS12-381. In their evaluation, they integrated their vectorized routines into the (scalar) `blst` library, achieving a 1.86× speedup over the original x64 assembly implementation. In fact, `blst` is primarily designed for BLS signatures and is currently employed in many projects of Ethereum. Therefore, this naturally raises the question of whether AVX-512 can be exploited more extensively,

[1] The specification of Filecoin: https://spec.filecoin.io.
[2] See https://github.com/supranational/blst.

namely, to accelerate the entire BLS signature rather than only the pairing computation.

Contributions. This paper aims to answer the above question and presents, to the best of our knowledge, the first AVX-512 implementation of the BLS signature scheme. Notably, we adopt the same experimental setting in [CFGP25] for a consistent evaluation, which means the x64 assembly implementation of `blst` is used as the starting point for development as well as the baseline for performance comparison, and an Intel Core i3-1005G1 CPU is used for benchmarking software. In summary, the contributions of this paper are threefold:

- First, we improve the scalar implementation by integrating a more efficient hash-to-curve into `blst`: we take advantage of the state-of-the-art SwiftEC method of Chávez-Saab et al. [CRT22] to replace the built-in Wahby-Boneh method [WB19] (standardized in RFC 9380 [FHSS+23]), achieving at least 1.07× speedups for signing.

- Second, we propose various vectorization strategies for scalar multiplication and hash-to-curve, the two dominant operations in the paths of the BLS signature API. Specifically, we explore different parallelization granularities, including 2-way and 4-way, as well as a hybrid way that combines both, which is enabled by employing different finite-field implementations and/or leveraging AVX-512VL.

- Third, our vectorization strategies are designed and evaluated across different algorithmic variants, including: 1) various formulas for point doubling and point addition; 2) GLS, GLV, and wNAF for scalar multiplication; and 3) Wahby-Boneh and SwiftEC for hash-to-curve. In addition, we also revisit the selection of the optimal window size for the precomputed table used in the vectorized scalar multiplications.

All these optimizations and vectorization techniques are collected in `avxbls-sig`[3], a latency-optimized and constant-time implementation of BLS signature using AVX-512 (in particular the AVX-512IFMA) instructions. According to our benchmarks, our `avxbls-sig` outperforms `blst` by a factor of at least 1.57. The source code of our `avxbls-sig` is publicly available[4].

Organization. Section 2 provides background information on the BLS signature scheme and the Intel AVX-512 extension. In Section 3, we detail our vectorization strategies for the performance-critical operations identified by profiling. Section 4 then presents a comprehensive performance evaluation, comparing our vectorized `avxbls-sig` library with the scalar `blst` baseline. Finally, Section 5 concludes the paper and discusses future work.

2 Background

2.1 BLS signature scheme

Although in principle the BLS scheme can be instantiated with any pairing, current implementations use the optimal ate pairing on the BLS12-381 curve as standard option at 128-bit security. BLS12-381 [Bow17] is a member of the BLS family of "pairing-friendly" elliptic curves with embedding degree 12 and j-invariant 0 [BLS02]. This curve is defined over a prime field $\mathbb{F}_p$ of size 381 bits, with equation $E_1/\mathbb{F}_p : y^2 = x^3 + 4$ and order $\#E_1(\mathbb{F}_p) = h_1 \cdot r$, where r is a prime of size 255 bits and h_1 is a *cofactor* of size 126 bits.

[3] To clearly distinguish our *signature* implementation from the *optimal ate pairing* implementation of Cheng et al. [CFGP25], we call their library `avxbls-oap` throughout this paper.

[4] See https://github.com/ganqin-liu/avxbls-sig.

Algorithm 1: BLS signature scheme for $m \in \{1, 2\}$: $\mathsf{BLS} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

function $\mathsf{KeyGen}(m)$:
  $\mathrm{sk} \leftarrow_{\$} \mathbb{F}_r$, $\mathrm{pk} \leftarrow [\mathrm{sk}]P_m \in \mathbb{G}_m$
  return $(\mathrm{sk}, \mathrm{pk})$
function $\mathsf{Sign}(\mathrm{sk}, M, m)$:
  $H_M \leftarrow \mathsf{HashToCurve}(M) \in \mathbb{G}_{3-m}$
  $\sigma \leftarrow [\mathrm{sk}]H_M \in \mathbb{G}_{3-m}$
  return (σ)
function $\mathsf{Verify}(\mathrm{pk}, M, \sigma, m)$:
  $H_M \leftarrow \mathsf{HashToCurve}(M) \in \mathbb{G}_{3-m}$
  $e_1 \leftarrow e(\sigma, P_m)$, $e_2 \leftarrow e(H_M, \mathrm{pk})$
  if $e_1 = e_2$ **then**
    return Accept
  else
    return Reject

BLS pairing. The optimal ate pairing on BLS12-381 is a map $e : \mathbb{G}_2 \times \mathbb{G}_1 \to \mathbb{G}_\mathrm{T}$, where the *pairing groups* $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_\mathrm{T}$ are cyclic of same prime order r. More concretely, $\mathbb{G}_1 = E_1(\mathbb{F}_p)[r] \cap \ker(\pi_p - [1])$, where $\pi_p : E_1 \to E_1$ is the Frobenius endomorphism $(x, y) \mapsto (x^p, y^p)$. On the other hand, $\mathbb{G}_2 = E_2(\mathbb{F}_{p^2})[r] \cap \ker(\pi_p - [p])$, where $E_2/\mathbb{F}_{p^2}$ is the degree-6 twist of E_1 with order $\#E_2(\mathbb{F}_{p^2}) = h_2 \cdot r$ and equation $E_2/\mathbb{F}_{p^2} : y^2 = x^3 + 4(1+i)$, assuming the quadratic extension $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2 + 1\rangle$. The target group $\mathbb{G}_\mathrm{T}$ contains all r-th roots of unity μ_r in $\mathbb{F}_{p^{12}}$, where $k = 12$ is the embedding degree of the curve E_1. Assuming that $Q_1 \in \mathbb{G}_1$ and $Q_2 \in \mathbb{G}_2$, the optimal ate pairing on BLS12-381 is defined as $(Q_2, Q_1) \mapsto f^{(p^{12}-1)/r}$ and operates in two steps. First, the *Miller function* f is computed via the Miller loop, and in the second step, the Miller function is raised to the exponent $(p^{12} - 1)/r$, via the *final exponentiation*. We refer to the recent work of [CFGP25] for further details on the concrete description of the optimal ate pairing on BLS12-381.

BLS scheme. The BLS signature scheme consists of the usual $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ functions for key generation, signing, and signature verification, as described in Algorithm 1. Depending on the implementation, the scheme can operate in two modes:

- Minimal-signature-size ($m = 2$): signatures are elements of $\mathbb{G}_1$ and public keys lie in $\mathbb{G}_2$. Since we can store only the x-coordinate (compressed representation), a signature is 48 bytes and the public key is 96 bytes, because it is an element in $\mathbb{F}_{p^2}$.
- Minimal-public-key-size ($m = 1$): signatures are elements of $\mathbb{G}_2$ and public keys lie in $\mathbb{G}_1$. A signature is in $\mathbb{F}_{p^2}$ thus 96 bytes long, while the public key is 48 bytes.

In both modes of operation, the secret key is a random element in $\mathbb{F}_r$, therefore 32 bytes. The description in Algorithm 1 is formulated to capture both modes of operation, since the present work focuses on both cases $m \in \{1, 2\}$.

More precisely, we assume that P_1 and P_2 are the generators of the *source groups* $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. For mode $m \in \{1, 2\}$, we write $P_m \in \mathbb{G}_m$ (i.e., $P_m = P_1$ if $m = 1$, and $P_m = P_2$ if $m = 2$). In this case, a BLS signing key pair is denoted by $(\mathrm{sk}, \mathrm{pk})$, where $\mathrm{pk} = [\mathrm{sk}]P_m \in \mathbb{G}_m$. The Sign and Verify functions rely on the procedure HashToCurve for mapping a message M of arbitrary size to a point of order r in $\mathbb{G}_1$ or $\mathbb{G}_2$, depending on the mode of operation. We write $H_M \leftarrow \mathsf{HashToCurve}(M) \in \mathbb{G}_{3-m}$, where $M \in \{0, 1\}^*$ and the subscript M indicates "derived from message M". In addition to the HashToCurve operation, signature generation requires a scalar multiplication to construct the signature, represented as the point $\sigma \leftarrow [\mathrm{sk}]H_M \in \mathbb{G}_{3-m}$ (see [BBG+25] for details). Verification requires the computation of two pairings, namely $e(\sigma, P_m)$ and $e(H_M, \mathrm{pk})$, and checking if they are equal (accept) or not (reject). The verification process can be optimized by verifying a slightly modified equality:

$$e(\sigma, P_m) \stackrel{?}{=} e(H_M, \mathrm{pk}) \;\Leftrightarrow\; f_1^{\frac{p^{12}-1}{r}} \stackrel{?}{=} f_2^{\frac{p^{12}-1}{r}} \;\Leftrightarrow\; \left(f_2 \cdot f_1^{-1}\right)^{\frac{p^{12}-1}{r}} \stackrel{?}{=} 1$$

The latter equality test executes two Miller loops for computing the Miller functions f_1 and

f_2, but saves one final exponentiation at the cost of one multiplication and one inversion in $\mathbb{F}_{p^{12}}$, where the latter is free since it can be computed via conjugation.

BLS hash-to-curve. The HashToCurve function is executed in four steps. First, a message of arbitrary size is hashed to a fixed-length bit string. The second step maps this bit string to elements in $\mathbb{F}_{p^m}$, depending on the mode of operation $m \in \{1, 2\}$. In the third step, an encoding function translates these field elements to points in the group $E_m(\mathbb{F}_{p^m})$, which typically requires expensive operations such as quadratic residuosity tests and square root extraction in $\mathbb{F}_{p^m}$. The last step is the *cofactor clearing*, which involves a scalar multiplication by the cofactor h_m. From an implementation perspective, HashToCurve must be efficient and resistant against timing attacks. Several methods have been proposed for constructing secure, indifferentiable hash functions to pairing-friendly elliptic curves. Wahby and Boneh [WB19] present a constant-time approach in which a field element is first mapped to a point on an auxiliary isogenous curve using the Simplified Shallue-van de Woestijne-Ulas (SSWU) method [Ula07], and then the point is mapped to the curve $E_m/\mathbb{F}_{p^m}$ with an isogeny. The `blst` library follows this procedure for hashing to the group $\mathbb{G}_m$ since it is also standardized in RFC 9380 [FHSS+23]. However, the state-of-the-art approach is the SwiftEC algorithm [CRT22], which can be used for pairing-friendly curves with $p \equiv 1 \bmod 3$, with either odd order or order divisible by 4, and j-invariant 0. The SwiftEC was later generalized by Koshelev [Kos25], expanding to all pairing-friendly elliptic curves with $p \equiv 1 \bmod 3$, including the ones with non-zero j-invariant. In this paper, we also consider the SwiftEC as a replacement of the Wahby-Boneh approach. We give more details on both methods and the description of our vectorization strategy in Section 3.4.

BLS layer-1 (L1) aggregation. A major advantage of the BLS scheme is the natural support for non-interactive signature aggregation. We briefly describe this property for both modes of operation $m \in \{1, 2\}$. Let $\sigma_1, \ldots, \sigma_N \in \mathbb{G}_{3-m}$ be signatures on the messages $M_1, \ldots, M_N \in \{0, 1\}^*$, from N distinct users with public keys $\mathrm{pk}_i = [\mathrm{sk}_i]P_m \in \mathbb{G}_m$, for $i = 1, \ldots, N$. The L1 aggregate signature is a sum of points $\sigma_{\mathrm{agg}} = \sigma_1 + \ldots + \sigma_N$, where $\sigma_i = [\mathrm{sk}_i]H_{M_i}$ and $H_{M_i} = \mathsf{HashToCurve}(M_i) \in \mathbb{G}_{3-m}$. Verifying the N signatures individually requires the execution of the Verify function N times, thus $2N$ Miller loop computations and N final exponentiations. The aggregation property allows the verification of the aggregated signature σ_{agg} at the cost of $N + 1$ Miller loops and only 1 final exponentiation. This is obtained through the equivalent verification equation:

$$e(\sigma_{\mathrm{agg}}, P_m) \stackrel{?}{=} \prod_{i=1}^{N} e(H_{M_i}, \mathrm{pk}_i) \;\Leftrightarrow\; f^{\frac{p^{12}-1}{r}} \stackrel{?}{=} \prod_{i=1}^{N} f_i^{\frac{p^{12}-1}{r}} \;\Leftrightarrow\; \left[\left(\prod_{i=1}^{N} f_i\right) \cdot f^{-1}\right]^{\frac{p^{12}-1}{r}} \stackrel{?}{=} 1$$

with the additional cost of N multiplications and 1 conjugation in $\mathbb{F}_{p^{12}}$.

`blst` implements L1 aggregation by integrating the functions Aggregate and FinalVerify in the BLS stack. Given N signatures $\sigma_1, \ldots, \sigma_N$, messages $M_1, \ldots, M_N$ and public keys $\mathrm{pk}_1, \ldots, \mathrm{pk}_N$, L1 aggregation requires N executions of the Aggregate function, where in each execution i, the function performs two *subgroup membership tests* on the points σ_i and pk_i to ensure that they belong in the correct elliptic curve subgroup (i.e., in $\mathbb{G}_1$ or $\mathbb{G}_2$). The IETF BLS signature specification [BBG+25] mandates these tests as a mitigation strategy against small subgroup attacks [LL97]. Then the function updates the aggregate signature as $\sigma_{\mathrm{agg}} = \sigma_{\mathrm{agg}} + \sigma_i$, it computes the point $H_{M_i} = \mathsf{HashToCurve}(M_i)$, the Miller function f_i and updates the accumulated product of Miller functions $F = F \cdot f_i$. FinalVerify performs the actual verification of the aggregated signature by computing the Miller function f and the value $(F \cdot f^{-1})^{(p^{12}-1)/r}$, and checking whether it is equal to 1.

BLS layer-2 (L2) randomized aggregation. BLS also allows aggregating multiple L1 aggregate signatures and executing a batch verification, thus validating a single signature with one pairing equation (L2 aggregation). This is discussed in Buterin's post [But19] and formalized in [ANS21]. Moreover, in his post [But19], Buterin emphasizes on a specific L2 aggregation instance, in which the same n messages are signed by many different public keys, which is a central component of Ethereum's Proof-of-Stake (PoS) protocol. This L2 aggregation case is supported by `blst` and integrated in our `avxbls-sig` software.

Given n messages $M_1, \ldots, M_n \in \{0,1\}^*$, N public keys $\mathrm{pk}_1, \ldots, \mathrm{pk}_N \in \mathbb{G}_m$ and $(n \cdot N)$ BLS signatures $\sigma_{i,j} \in \mathbb{G}_{3-m}$, for $i = 1, \ldots, n$ and $j = 1, \ldots, N$, the L2 aggregation in `blst` proceeds as follows. First, it executes subgroup membership tests for all BLS signatures $\sigma_{i,j}$ and public keys pk_j, it constructs the aggregate signatures $\sigma_{\mathrm{agg}_i} = \sigma_{i,1} + \ldots + \sigma_{i,N}$, for $i = 1, \ldots, n$ and the aggregate public key $\mathrm{pk} = \mathrm{pk}_1 + \ldots + \mathrm{pk}_N$. Then it performs n calls of the Aggregate function with the randomization option enabled, where in each execution i it performs two subgroup membership tests for the signature σ_{agg_i} and the aggregated public key pk, generates a 64-bit random scalar $r_i \in \mathbb{F}_r$ and computes the points $[r_i]\sigma_{\mathrm{agg}_i}$, $\mathrm{pk}'_i = [r_i]\mathrm{pk}$, $H_{M_i} = \mathsf{HashToCurve}(M_i)$. The function updates the L2 aggregate signature $\sigma^* = \sigma^* + [r_i]\sigma_{\mathrm{agg}_i}$, computes the Miller function f_i and updates the product $F = F \cdot f_i$. The verification is derived by the pairing equality:

$$e(\sigma^*, P_m) \stackrel{?}{=} \prod_{i=1}^{n} e(H_{M_i}, \mathrm{pk}'_i) \;\Leftrightarrow\; f^{\frac{p^{12}-1}{r}} \stackrel{?}{=} \prod_{i=1}^{n} f_i^{\frac{p^{12}-1}{r}} \;\Leftrightarrow\; \left[\left(\prod_{i=1}^{n} f_i\right) \cdot f^{-1}\right]^{\frac{p^{12}-1}{r}} \stackrel{?}{=} 1$$

Finally, `blst` executes the FinalVerify function to validate the L2 aggregate signature σ^*, which computes the Miller function f and the value $(F \cdot f^{-1})^{(p^{12}-1)/r}$, and checks whether it is equal to 1.

Alternatively, instead of computing the point $\mathrm{pk}'_i = [r_i]\mathrm{pk}$, one can compute the point $H'_{M_i} = [r_i]H_{M_i}$. By the bilinearity of the pairing, the use of H'_{M_i} or pk'_i in verification is equivalent. In practice, however, one typically applies the scalar to the element in $\mathbb{G}_1$, since scalar multiplication in $\mathbb{G}_1$ is substantially cheaper than in $\mathbb{G}_2$. Furthermore, although the subgroup membership tests for public keys are disabled in `blst`, in our benchmarks we execute subgroup membership tests for $\mathrm{pk}_1, \ldots, \mathrm{pk}_N$ and the aggregated public key pk, both in `blst` and our `avxbls-sig` software, following the recommendation in [BBG+25].

Rogue-key attacks. [BBG+25] discusses the concept of *rogue-key attacks*, which target the aggregation functionality of BLS signatures. In this class of attacks, the adversary uses a specially crafted public key (a "rogue key") to forge an aggregate BLS signature, causing a successful verification of an invalid signature. Such attacks occur when different public keys are used to sign the same message during aggregation. According to [BBG+25, Section 3], a simple countermeasure against rogue-key attacks is to require all messages signed in an aggregate signature to be distinct. `blst` follows this approach for L1 aggregation and provides a function `Uniq`, that can be used to verify that all signed messages are different, at negligible cost. This protection mechanism against rogue-key attacks is not suitable in application scenarios where the same messages must be signed by different public keys, such as the L2 aggregation functionality that is implemented in Ethereum's PoS protocol [But19]. An alternative countermeasure in this case, is to use a separate public key validation step, called *Proof-of-Possession* (PoP) [RY07]. The PoP scheme creates a new point by hashing the public key $H_{\mathrm{pk}} \leftarrow \mathsf{HashToCurve}(\mathrm{pk})$ and sets $R \leftarrow [\mathrm{sk}]H_{\mathrm{pk}}$ as the "proof of possession" for the public key pk. Verification requires the reconstruction of the point $H'_{\mathrm{pk}} \leftarrow \mathsf{HashToCurve}(\mathrm{pk})$ and the verification of the pairing equality $e(H'_{\mathrm{pk}}, \mathrm{pk}) = e(R, P_m)$. Since this validation has to be performed for all public keys, the PoP countermeasure adds significant overhead to the overall application. The `blst` library does not include PoP validation tests, however, it assumes that these tests are handled externally by

the application[5]. This is also the case in Ethereum, where as described in Buterin's post [But19], the Ethereum PoS protocol includes PoP validation tests for public keys, which is evident in the implementation of the Prysm PoS consensus client[6]. Following `blst`, we do not include an implementation of the PoP scheme in our `avxbls-sig` software.

2.2 AVX-512

AVX-512 is the third generation of Intel Advanced Vector eXtension (AVX) family, which augments the computing capability with 32 512-bit vector registers (`zmm0-zmm31`) and supports a comprehensive mask-based execution model that facilitates high instruction-level parallelism. Critical to our implementation is one subextension named Integer Fused Multiply-Add (IFMA), which introduces two specialized instructions `vpmadd52luq` and `vpmadd52huq`. Concretely, each AVX-512IFMA instruction performs parallel multiplications on eight pairs of the packed 52-bit unsigned integers to generate eight 104-bit intermediate products, whose lower or higher halves are then accumulated into corresponding eight 64-bit elements of the destination vector register. Note that the platform we use to conduct experiments and benchmark software is an Intel Ice Lake Core i3-1005G1 CPU, where the IFMA instruction exhibits a latency of 4 clock cycles and a Cycles Per Instruction (CPI) of 1 [AR19].

Another important subextension we utilized widely is Vector Length (VL). Instead of introducing new instructions, AVX-512VL extends most AVX-512 operations to be able to operate on 128-bit `xmm` and 256-bit `ymm` registers, which simplifies and facilitates greatly the vectorization for $E_1(\mathbb{F}_p)$ operations in this work. Furthermore, the 256-bit IFMA instruction, i.e., the instruction working with `ymm` registers, has a CPI of 0.5, which is equally efficient to the 512-bit IFMA instruction (when the needed ports are all available). However, most other instructions such as `vpaddq` and `vpandq` possess a higher efficiency in their 512-bit version than 256-bit version. Therefore, we use AVX-512VL, namely 256-bit instructions for only the 2-way vectorized case of the $E_1(\mathbb{F}_p)$ operations, and for all other cases, we opt for 512-bit instructions.

2.3 Notation

When analyzing the computational cost of different operations, we use M_m, S_m, A_m, and I_m to denote multiplication, squaring, addition (or subtraction), and inversion in the finite field $\mathbb{F}_{p^m}$, respectively. Additionally, we employ the superscript 2 (resp. 4) to denote the corresponding 2-way (resp. 4-way) vectorized operations, e.g., M_m^2 represents a 2-way vectorized multiplication in $\mathbb{F}_{p^m}$. Then when describing the vectorization pattern, we define a $(w \times x \times y \times z)$-way vectorized implementation for an elliptic curve operation: w curve operations are executed in parallel; each of the curve operations performs x $\mathbb{F}_{p^2}$ operations in parallel; each of the $\mathbb{F}_{p^2}$ operations executes y $\mathbb{F}_p$ operations in parallel; and each of the $\mathbb{F}_p$ operations works on z 64-bit elements of a vector. Analogously, for $\mathbb{F}_{p^2}$ (resp. $\mathbb{F}_p$) operations, we define an $(x \times y \times z)$-way (resp. $(y \times z)$-way) vectorized implementation. Finally, in the description of some Algorithms, colored background is used to distinguish clearly the different parallelization granularities: *orange* for 4-way parallelization, *blue* for 2-way parallelization, and *red* for scalar (1-way) execution.

3 Implementation

Profiling. The BLS signature scheme features a multi-level structure: the *high-level* functions, i.e., key generation, signing, aggregation, and final verification, are composed of

[5] See, e.g., line 783 in https://github.com/supranational/blst/blob/master/bindings/go/blst.go
[6] Line 119: https://github.com/OffchainLabs/prysm/blob/develop/crypto/bls/blst/signature.go

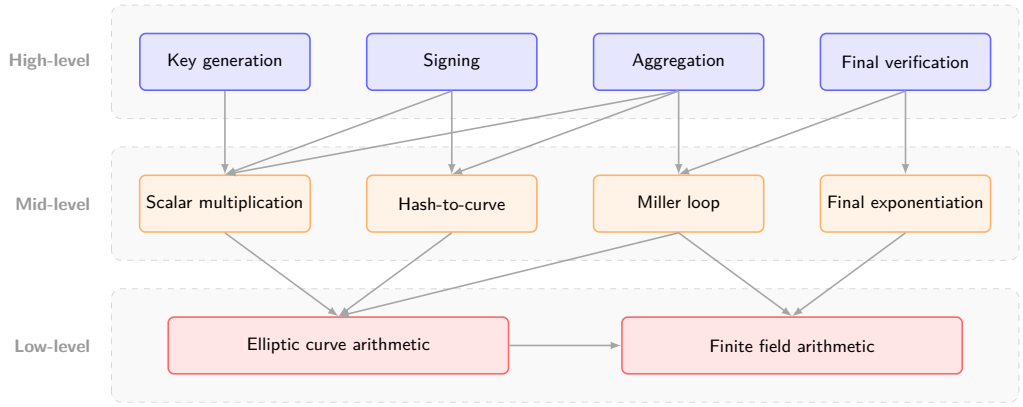

Figure 1: Level-wise structure of the BLS signature scheme.

Table 1: Profiling of high-level operations in `blst`, i.e., signing, aggregation, and (final) verification on an Intel i3-1005G1 CPU. Since the key generation in $\mathbb{G}_1$ (resp. $\mathbb{G}_2$) relies solely on the E_1 (resp. E_2) scalar multiplication, it is therefore not listed for simplicity.

Operation	Signing		Aggregation (L1)		Aggregation (L2)		Verification
	$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_1$	$\mathbb{G}_2$	-
E_1 scalar mul.	62.41%	-	13.11%	9.65%	27.63%	13.50%	-
E_2 scalar mul.	-	50.24%	15.25%	11.12%	12.67%	22.83%	-
hash-to-curve	37.59%	49.76%	17.32%	39.25%	14.50%	31.24%	-
Miller loop	-	-	52.70%	38.44%	43.82%	30.62%	42.29%
final expo.	-	-	-	-	-	-	56.87%
other	-	-	1.62%	1.54%	1.38%	1.81%	0.84%

some *mid-level* operations such as scalar multiplication, hash-to-curve, Miller loop, and final exponentiation; these mid-level operations, in turn, consist of *low-level* elliptic curve and finite field operations. For a clearer understanding, we provide a graphic description in Figure 1. In order to identify the performance-critical operations, we conduct a performance profiling for the baseline, namely the x64 implementation of BLS signature in `blst`, on an Intel Ice Lake Core i3-1005G1 CPU. The profiling results of high-level and mid-level operations are shown in Table 1 and Table 2, respectively.

Target for vectorizing. Table 1 indicates that the mentioned four mid-level operations dominate the performance of BLS scheme: together, they constitute more than 90% of the execution time in each high-level function. Given that the vectorization of Miller loop and final exponentiation has been studied in [CFGP25], in this work we focus on the remaining two operations, namely scalar multiplication and hash-to-curve. Table 2 provides a lower-level breakdown. We observe that vectorizing modular inversion and square-root extraction, which are dominant in hash-to-curve, is inherently challenging due to serial nature of their addition chains. Consequently, our vectorization effort concentrates on three performance-critical elliptic-curve operations (Section 3.2): point doubling, point addition, and unified point doubling-and-addition (DADD), over both $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. These three building blocks form the foundation for three different scalar-multiplication formulas, which we analyze systematically in Section 3.3. Finally, the hash-to-curve process (including map to isogenous curve, isogeny map, etc.) is discussed separately in Section 3.4.

Table 2: Profiling of mid-level operations in `blst`, i.e., scalar multiplication and hash-to-curve, on an Intel i3-1005G1 CPU.

Operation	Scalar multiplication				Hash-to-curve	
	$\mathbb{G}_1$ GLV	$\mathbb{G}_2$ GLS	$\mathbb{G}_1$ wNAF	$\mathbb{G}_2$ wNAF	$\mathbb{G}_1$	$\mathbb{G}_2$
point doubling	38.65%	24.59%	57.39%	55.67%	30.45%	46.12%
point addition	4.32%	6.05%	12.49%	13.75%	-	-
point DADD	34.20%	49.68%	23.65%	28.80%	8.07%	15.72%
map to isogenous curve	-	-	-	-	51.62%	34.64%
isogeny map	-	-	-	-	7.50%	1.81%
$\mathbb{F}_p$ inversion	14.66%	7.90%	-	-	-	-
other	8.17%	11.78%	6.47%	1.78%	2.36%	1.71%

3.1 $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ operations

Cheng et al. [CFGP25] in `avxbls-oap` developed various vectorized implementations of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ operations on BLS12-381 curve, using a 48-bit-per-limb representation, and the vectorization strategies are based on a prior implementation of SIKE [CFGR22]. In this work, we take advantage of their $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ implementations, with two more new optimization techniques.

Using AVX-512VL. Unlike the optimal ate pairing, which operates on the $\mathbb{F}_{p^{12}}$ tower extension, the scalar multiplication and hash-to-curve routines are based mainly on elliptic curve operations over $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$, which are difficult to be vectorized *fully* in 4-way, i.e., usually 2-way or a mixed-granularity (*hybrid*) combination of 4-way and 2-way is deployed. This means, for the case of $E_2(\mathbb{F}_{p^2})$, our vectorized implementation would need both $(4 \times 2 \times 1)$-way and $(2 \times 2 \times 2)$-way $\mathbb{F}_{p^2}$ implementations, while for the case of $E_1(\mathbb{F}_p)$, it would need (4×2)-way and (2×4)-way $\mathbb{F}_p$ implementations. However, a problem is that the (2×4)-way is not used in [CFGP25] and thus not implemented, and more importantly, not efficient in practice. To address this, we develop a shortened (2×2)-way implementation as a replacement, which utilizes the 256-bit vector length instead of 512-bit thanks to AVX-512VL, and according to our experiments it is more efficient than (2×4)-way.

Lazy reduction in $\mathbb{F}_p$ addition and subtraction. The implementation of $\mathbb{F}_p$ addition (resp. $\mathbb{F}_p$ subtraction) in both `blst` and `avxbls-oap` contains two steps: 1) it performs integer addition $t = a + b - p$ (resp. integer subtraction $t = a - b$); 2) it then checks whether t is negative, and depending on the checking result it will perform in constant-time either $r = t + p$ (if $t < 0$) or $r = t$ (if $t \geq 0$). We observe that this structure leaves room for optimization. More concretely, inside an elliptic curve operation, it is common that consecutive field additions/subtractions are needed to compute the required operand for a multiplication. On the other hand, the implemented Montgomery reduction allows inputs up to pR, where $R = 2^{384} > 9p$. This implies that even after multiple additions/subtractions without modular reduction, the intermediate results remain within $[0, pR)$, which is acceptable for subsequent multiplications. Hence, the full two-step reduction is not always mandatory.

Based on this observation, we optimize the relevant operations such that they now need only a single step. In the additive case, this reduces to a plain addition $a + b$, since the intermediate result remains non-negative and within the Montgomery interval. In the subtractive case, we add a minimal multiple of p to prevent the overall value from becoming negative (i.e., computing $a - b + np$). While addition can save also carry propagation, subtraction still requires it to ensure each limb to be non-negative. We apply this method by systematically checking all vectorized $\mathbb{F}_p$ additions/subtractions, which, according to our experiments, improves the efficiency of $E_1(\mathbb{F}_p)$ operations by at least 10%.

Algorithm 2: `point_dbl` with hybrid 2-/4-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: Point $P = (X_P, Y_P, Z_P)$ in Jacobian coordinates.
Output: Point $R = [2]P = (X_R, Y_R, Z_R)$ in Jacobian coordinates.

1	$Z_R \leftarrow Y_P + Z_P$			
2	$t_0 \leftarrow X_P^2$	$s_0 \leftarrow Y_P^2$	$ZZ \leftarrow Z_P^2$	$Z_R \leftarrow Z_R^2$
3	$t_1 \leftarrow X_P + X_P$	$s_1 \leftarrow t_0 + t_0$	$t_2 \leftarrow s_0 + s_0$	$Z_R \leftarrow Z_R - s_0$
4	$s_0 \leftarrow s_1 + t_0$	$Z_R \leftarrow Z_R - ZZ$	8 $t_0 \leftarrow s_1 - X_R$	
5	$t_0 \leftarrow s_0 \cdot s_0$	$s_1 \leftarrow t_1 \cdot t_2$	9 $Y_R \leftarrow s_0 \cdot t_0$	$s_1 \leftarrow t_2 \cdot t_2$
6	$t_1 \leftarrow s_1 + s_1$		10 $s_1 \leftarrow s_1 + s_1$	
7	$X_R \leftarrow t_0 - t_1$		11 $Y_R \leftarrow Y_R - s_1$	
12	**return** $R = (X_R, Y_R, Z_R)$			

3.2 Elliptic curve arithmetic operations

Point doubling (`point_dbl`). Let $P = (X_P, Y_P, Z_P)$ be a point in $E_m(\mathbb{F}_{p^m})$ in Jacobian coordinates, for $m \in \{1, 2\}$. Then, $R = [2]P = (X_R, Y_R, Z_R)$ is computed as:

$$R = (X_R, Y_R, Z_R) = \left(9X_P^4 - 8X_P Y_P^2,\ 3X_P^2(12X_P Y_P^2 - 9X_P^4) - 8Y_P^4,\ 2Y_P Z_P\right)$$

`blst` implements the `dbl-2009-l` formula[7] specified in the Explicit-Formulas Database (EFD), which requires $5S_m + 2M_m + 11A_m$ operations depending on the field choice $\mathbb{F}_{p^m}$.

In our analysis, we considered two different strategies for the optimal vectorization of the doubling operation. The first is described in Algorithm 2 and adopts a *hybrid* 2-/4-way fashion based on the `dbl-2009-alnr` formula[8] from the EFD, with a total cost of $1S_m^4 + 2M_m^2 + 2A_m^4 + 6A_m^2$. Specifically, this hybrid approach employs 4-way vectorization for the initial steps (lines 2–3) and then transitions to a 2-way vectorization (lines 4–11).

Due to instruction dependencies, a full parallelization of the addition operations cannot be achieved. Between the options of modifying data flow and leaving some lanes idle, we chose the latter to preserve instruction efficiency and avoid costly data rearrangement. The second approach is a completely 2-way vectorization of the `dbl-2009-l` formula used in `blst` with a total cost of $1M_m^2 + 3S_m^2 + 8A_m^2$ (Algorithm 9 in Appendix A). The benchmarks in Table 3 show that the hybrid 2-/4-way approach offers a better speedup, and therefore we choose this method for integration in our `avxbls-sig` software.

Remark 1. We note that the point doubling also appears in the `line_dbl` function in `blst`, used in the Miller loop. In [CFGP25] the authors present a 4-way vectorization of `line_dbl`, however this is optimized to include both the point doubling and line computation, and thus it cannot be directly adopted to our scenario where only the point doubling is considered.

Point addition (`point_add`). Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ be points in Jacobian coordinates, in $E_m(\mathbb{F}_{p^m})$, for $m \in \{1, 2\}$. Their sum $R = P + Q = (X_R, Y_R, Z_R)$ is computed as follows:

$$\begin{aligned}
X_R &= 4(Y_Q Z_P^3 - Y_P Z_Q^3)^2 - 4(X_Q Z_P^2 + X_P Z_Q^2)(X_Q Z_P^2 - X_P Z_Q^2)^2 \\
Y_R &= 2(Y_Q Z_P^3 - Y_P Z_Q^3)\left(4X_P Z_Q^2 (X_Q Z_P^2 - X_P Z_Q^2)^2 - X_R\right) - 8Y_P Z_Q^3 (X_Q Z_P^2 - X_P Z_Q^2)^3 \\
Z_R &= 2Z_P Z_Q (X_Q Z_P^2 - X_P Z_Q^2).
\end{aligned}$$

The `blst` library adopts the `add-2007-bl` formula[9] from the EFD for point addition with a cost of $5S_m + 11M_m + 13A_m$ operations, depending on the choice of the field $\mathbb{F}_{p^m}$.

[7] https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#doubling-dbl-2009-l
[8] https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#doubling-dbl-2009-alnr
[9] https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-2007-bl

Algorithm 3: `point_add` with hybrid 1-/4-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: Points $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ in Jacobian coordinates.
Output: Point $R = P + Q = (X_R, Y_R, Z_R)$ in Jacobian coordinates.

1 $ZZ_Q \leftarrow Z_Q \cdot Z_Q$	$ZZ_P \leftarrow Z_P \cdot Z_P$	$t_0 \leftarrow Y_P \cdot Z_Q$	$s_0 \leftarrow Y_Q \cdot Z_P$
2 $t_1 \leftarrow X_P \cdot ZZ_Q$	$s_1 \leftarrow X_Q \cdot ZZ_P$	$t_0 \leftarrow t_0 \cdot ZZ_Q$	$s_0 \leftarrow s_0 \cdot ZZ_P$
3 $t_2 \leftarrow s_1 - t_1$	$s_2 \leftarrow s_0 - t_0$	$t_3 \leftarrow t_1 + s_1$	
4 $t_4 \leftarrow t_2 \cdot t_2$	$s_3 \leftarrow s_2 \cdot s_2$	$s_4 \leftarrow t_0 \cdot t_2$	$Z_R \leftarrow Z_P \cdot Z_Q$
5 $X_R \leftarrow t_3 \cdot t_4$	$s_0 \leftarrow s_4 \cdot t_4$	$t_0 \leftarrow t_1 \cdot t_4$	$Z_R \leftarrow Z_R \cdot t_2$
6 $X_R \leftarrow s_3 - X_R$		8 $Y_R \leftarrow s_2 \cdot Y_R$	
7 $Y_R \leftarrow t_0 - X_R$		9 $Y_R \leftarrow Y_R - s_0$	
10 **return** $R = (X_R, Y_R, Z_R)$			

We considered three different approaches for the vectorization of the point addition. The straightforward approach is to deploy a 2-way vectorization of the existing `add-2007-bl` formula. This is described in Algorithm 10 (Appendix A) and requires $6M_m^2 + 2S_m^2 + 8A_m^2$ field operations. Building upon the baseline implementation of `blst`, we implemented a second variant that retains the `add-2007-bl` structure but introduces minor formula rearrangements to enable partial 4-way parallelism. The result is a hybrid 2-/4-way vectorized implementation with a total cost of $2M_m^4 + 3M_m^2 + 1S_m^2 + 1A_m^4 + 6A_m^2$ operations, described in Algorithm 11 (Appendix A). Finally, to maximize 4-way vectorization, we derived a third, hybrid variant based on the `add-1986-cc` formula[10] from the EFD, where the main bulk of field operations is executed in 4-way, with only the last field multiplication executed in scalar mode. This approach is described in Algorithm 3 and yields a total cost of $4M_m^4 + 1M_m + 1A_m^4 + 3A_m$ field operations, offering better speedup compared to the other methods. In this case, the constant factors that appear in the formulas for X_R, Y_R and Z_R are excluded from the computation, as they are canceled out when computing the affine point $R = (X_R/Z_R^2, Y_R/Z_R^3)$, hence they do not affect the construction of $P + Q$.

Unified point doubling and addition (`point_DADD`). `blst` includes a *unified point doubling and addition (DADD)* function that executes either point doubling or point addition, depending on whether the two input points are equal or not, using the same instructions for both cases. Point DADD is the most performance-critical operation in the scalar multiplication for both curves $E_1/\mathbb{F}_p$ and $E_2/\mathbb{F}_{p^2}$. The `blst` implementation follows a constant time switch between point addition and point doubling, while it is also designed to handle in constant time the case where the resulting point is the point at infinity ($\mathcal{O}$).

Let $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ be two points in $E_m(\mathbb{F}_{p^m})$, for $m \in \{1, 2\}$, not necessarily distinct. We denote the point DADD operation as $R = P \oplus Q$, to distinguish from the normal point addition described previously. Then the point $R = (X_R, Y_R, Z_R)$ is computed with the following unified formulas:

$$R = (X_R, Y_R, Z_R) = \left(S^2 - U^2 \cdot T,\ S^2(U^2 \cdot U_1 - X_R) - U^3 \cdot S_1,\ U \cdot ZZ\right)$$

where the intermediate values U, S, T, U_1, S_1, ZZ are derived from different formulas, depending on whether point addition is executed (if $P \neq Q$), or point doubling (if $P = Q$). In particular, when point addition is executed, the intermediate values are defined as:

$$S = (Y_Q Z_P^3 - Y_P Z_Q^3), \qquad U = (X_Q Z_P^2 - X_P Z_Q^2), \qquad T = (X_Q Z_P^2 + X_P Z_Q^2)$$
$$U_1 = X_P Z_Q^2, \qquad S_1 = Y_P Z_Q^3, \qquad ZZ = Z_P Z_Q$$

On the other hand, when point doubling is executed, they are computed as:

$$S = 3X_P^2, \quad U = 2Y_P, \quad T = 2X_P, \quad U_1 = X_P, \quad S_1 = Y_P, \quad ZZ = Z_P$$

[10] https://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html#addition-add-1986-cc

Algorithm 4: `point_DADD` with hybrid 2-/4-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: Points $P = (X_P, Y_P, Z_P)$, $Q = (X_Q, Y_Q, Z_Q)$ in Jacobian coordinates
Output: Point $R = P \oplus Q = (X_R, Y_R, Z_R)$ in Jacobian coordinates

1	$t_0 \leftarrow Z_P \cdot Z_P$	$s_0 \leftarrow Z_Q \cdot Z_Q$	$w_0 \leftarrow X_P \cdot X_P$	$Z_R \leftarrow Z_P \cdot Z_Q$
2	$t_1 \leftarrow Y_P \cdot Z_Q$	$a_S \leftarrow Y_Q \cdot Z_P$	$U_1 \leftarrow X_P \cdot s_0$	$a_U \leftarrow X_Q \cdot t_0$
3	$d_U \leftarrow Y_P + Y_P$	$d_S \leftarrow w_0 + w_0$	$a_T \leftarrow a_U + U_1$	$a_U \leftarrow a_U - U_1$
4	$(U, X_R, Z_R) \leftarrow \texttt{cselect}((d_U, X_P, Z_P), (a_U, U_1, Z_R), \texttt{IsZero}(a_U))$			
5	$t_2 \leftarrow t_1 \cdot s_0$	$a_S \leftarrow a_S \cdot t_0$	$U \leftarrow U \cdot U$	$Z_R \leftarrow Z_R \cdot U$
6	$a_S \leftarrow a_S - t_2$	$d_S \leftarrow d_S + w_0$	$d_T \leftarrow X_P + X_P$	
7	$(S, S', T, U', V) \leftarrow \texttt{cselect}((d_S, d_S, d_T, d_U, Y_P), (a_S, a_S, a_T, a_U, t_2), \texttt{IsZero}(a_U))$			
8	$Y_R \leftarrow X_R \cdot U$	$S \leftarrow U \cdot U'$	$X_R \leftarrow S \cdot S$	$U \leftarrow U \cdot T$
9	$X_R \leftarrow X_R - U$		11 $Y_R \leftarrow Y_R \cdot S'$	$S \leftarrow S \cdot V$
10	$Y_R \leftarrow Y_R - X_R$		12 $Y_R \leftarrow Y_R - S$	
13	**return** $R = (X_R, Y_R, Z_R)$			

The point DADD in `blst` requires $13M_m + 5S_m + 10A_m$ operations for curves of the form $y^2 = x^3 + b_m$, covering both BLS12-381 and its degree six twist.

We explored two distinct strategies to vectorize this complex operation. The first approach is a completely 2-way vectorized implementation (Algorithm 12 in Appendix A). This method computes the intermediate terms in parallel pairs, incurring a total cost of $7M_m^2 + 2S_m^2 + 7A_m^2$ field operations, plus a constant-time selection step[11] (line 7) to blend the addition/doubling parameters. The second variant follows our hybrid 2-/4-way fashion, shown in Algorithm 4. We observe that the initial calculation of the six intermediate variables (S, U, T, U_1, S_1, ZZ) involves a set of independent multiplications and squarings that can be effectively batched. Crucially, to achieve this 4-way parallelism, we introduce an additional selection step (line 4) within the execution flow. By adjusting the operands dynamically, this step unifies the arithmetic structure of point doubling and addition, allowing the 512-bit IFMA instructions to operate on uniform data despite the algorithmic differences. Once these terms are computed, the algorithm transitions to a 2-way layout to compute the final coordinates of R, which exhibit tighter data dependencies. This hybrid design requires $4M_m^4 + 1M_m^2 + 2A_m^4 + 3A_m^2$ field operations. Our benchmarks show that the hybrid approach outperforms the purely 2-way strategy, and thus we adopt the hybrid 2-/4-way method in `avxbls-sig`.

3.3 Scalar multiplication

Following `blst`, in our framework, the windowed Non-Adjacent Form (wNAF) method [JT01] serves as the core arithmetic engine for all scalar multiplication operations. Depending on the scalar's bit-length and properties, we categorize the operations into three distinct scenarios: 1) For full-length 255-bit scalars (e.g., private keys) in $\mathbb{F}_r$, we employ the Gallant-Lambert-Vanstone (GLV) [GLV01] and Galbraith-Lin-Scott (GLS) [GLS09] decomposition methods for $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, to split the scalar into smaller sub-scalars before applying a multi-dimensional wNAF; 2) For short 64-bit scalars (e.g., in L2 randomized aggregation), we apply the standard wNAF directly, as the decomposition overhead outweighs the benefits for such bit-lengths; 3) For subgroup membership testing, we perform a classical left-to-right multiplication by a fixed scalar z.

Unified wNAF kernel and precomputation strategy. A central design objective of our framework is to unify precomputation, table lookup, and point arithmetic across

[11] `IsZero` outputs 1 (resp. 0) if the input is zero (resp. non-zero), via a constant-time check using bitwise logical instructions. `cselect` selects the 1st (resp. 2nd) input in constant-time if the 3rd input is 1 (resp. 0).

all variable-scalar multiplication methods, including GLV/GLS and standard wNAF. Irrespective of whether scalar decomposition is applied, the core computation ultimately relies on the same wNAF kernel. For a given window size w, we precompute a base table $T_0 = \{P, [2]P, \ldots, [2^{w-1}]P\}$. To mitigate cache timing attacks, we adapt the lookup strategy from the `blst` library to the AVX-512 setting: we scan each row individually of the entire table to conceal cache access patterns, and extract the target entry using AVX-512 masked instructions based on the Booth encoding [Boo51] of the scalar. We note that the window size $w = 5$, empirically chosen in `blst`, is not necessarily optimal for our vectorized implementation. Accordingly, we empirically evaluate multiple window sizes to identify the configuration that maximizes performance on our target architecture.

Remark 2. We have also evaluated an alternative strategy, which precomputes a window of points, normalizes them to affine coordinates, and performs point addition using a *mixed-coordinate* addition formula (e.g., [LM08, Section 4]). Although mixed addition formulas are faster than full Jacobian coordinate formulas in scalar implementation, in the SIMD setting the performance relies on managing data dependencies rather than raw operation count and mixed addition formulas do not alleviate these dependencies. In particular, under SIMD execution, this alternative strategy does not reduce the total number of vector multiplications; it only reduces a few 4-way vector multiplications to 2-way using mixed addition. As a result, the overall performance gain is offset by the cost of field inversions required for affine normalization. Therefore, following the `blst` implementation, in our `avxbls-sig` software we consider full (non-mixed) addition formulas.

GLV and GLS decomposition. `blst` uses endomorphism-based decompositions to accelerate scalar multiplication in $\mathbb{G}_1$ and $\mathbb{G}_2$. These methods split the large scalar in $\mathbb{F}_r$ into multiple smaller sub-scalars, transforming a single high-degree scalar multiplication into a multi-scalar multiplication, which is then evaluated using an interleaved wNAF approach. For $\mathbb{G}_1$, the GLV method [GLV01] decomposes a 255-bit scalar $\ell \in \mathbb{F}_r$ into two 128-bit sub-scalars ℓ_0 and ℓ_1 using the endomorphism $\phi(x, y) = (\zeta x, y)$, where ζ is a primitive third root of unity in $\mathbb{F}_p$. Then for any $P \in \mathbb{G}_1$, the scalar multiplication is computed as $[\ell]P = \ell_0 P + [\ell_1]\phi(P)$. For $\mathbb{G}_2$, the GLS method [GLS09] uses an efficient endomorphism $\psi = \phi^{-1} \circ \pi_p \circ \phi$, where $\phi : E_2(\mathbb{F}_{p^2}) \to E_1(\mathbb{F}_{p^{12}})$ is the twisting isomorphism, to decompose a 255-bit scalar $\ell \in \mathbb{F}_r$ into four 64-bit sub-scalars $\ell_0, \ell_1, \ell_2, \ell_3$. Then for any $Q \in \mathbb{G}_2$, the scalar multiplication is evaluated as $[\ell]Q = \sum_{i=0}^{3} [\ell_i]\psi^i(Q)$ in interleaved wNAF fashion. Within our unified framework, the auxiliary lookup tables T_i are derived by applying the corresponding endomorphisms to the base table T_0 (e.g., $T_1[j] = \phi(T_0[j])$).

Two precomputation strategies exist for such multi-dimensional settings: 1) constructing a multi-dimensional table and performing one lookup per dimension in each iteration (as in `blst`); 2) building a single combined table via point pre-addition or explicit endomorphism mappings, such as [FM25][12], enabling a single lookup per iteration. The optimal choice depends on the cost ratio between point addition and doubling. As shown in Section 3.2, our vectorized formulas yield larger speedups for point addition than doubling. Consequently, we adopt the first approach and process the sub-scalars in an interleaved manner using the shared vectorized lookup and point DADD (see Algorithm 4) subroutines.

Standard wNAF. For scalar multiplication involving short randomized scalars, such as in L2 aggregation, the scalar length is typically far below 255 bits (e.g., 64 bits in `blst`, following the recommendation in [ANS21]). In this context, the overhead of GLV/GLS decomposition outweighs its benefits. We therefore bypass the decomposition step and apply the standard wNAF method directly. This configuration relies solely on the base table T_0 and shares the same windowing logic, constant-time lookup kernel, and point arithmetic primitives as the endomorphism-based methods described above.

[12] This approach necessitates an $\mathbb{F}_p$ inversion, which is costly in our setting.

Subgroup membership testing. We adopt the endomorphism-based tests proposed by Scott [Sco21] to perform subgroup membership testing efficiently. Instead of a full scalar multiplication by the group order r, membership is verified by checking the conditions $\phi(P) = [-z^2]P$ for $\mathbb{G}_1$ and $\psi(Q) = [z]Q$ for $\mathbb{G}_2$. Unlike the general wNAF cases, $z = -\texttt{0xd201000000010000}$ is fixed with low Hamming weight for BLS12-381. This allows the computation to be implemented as a classical left-to-right traversal consisting of a sparse sequence of point doublings interleaved with a minimal number of additions.

3.4 Hashing to pairing groups

The BLS scheme necessitates hashing arbitrary-length bit strings to the elliptic curve subgroup $\mathbb{G}_m$, for $m \in \{1, 2\}$. This construction should respect the usual security requirements of an ideal collision-resistant hash function, and thus behave as closely as possible to a conventional random oracle, which is formally defined as *indifferentiability from a random oracle* [BCI+10]. Usually, this hash operation is constructed as the composition of four functions, denoted by $\mathsf{HashToCurve} = [h_m] \circ H \circ \eta \circ \chi$, and defined as follows:

- $\chi : \{0,1\}^* \to \{0,1\}^\ell$. A message expansion function, mapping to an ℓ-bit string.
- $\eta : \{0,1\}^\ell \to \mathbb{F}_{p^m}$. A mapping that converts the bit string into n field elements.
- $H : \mathbb{F}_{p^m} \to E_m(\mathbb{F}_{p^m})$. Admissible encoding to the group of points of an elliptic curve.
- $[h_m] : E_m(\mathbb{F}_{p^m}) \to \mathbb{G}_m$. Scalar multiplication by the cofactor $h_m \mid \#E_m(\mathbb{F}_{p^m})$.

While `blst` employs the Wahby-Boneh (WB) method [WB19], recent literature suggests that SwiftEC [CRT22] offers superior efficiency. The primary distinction between both approaches lies in the construction of the encoding function H. In `avxbls-sig`, we implement and evaluate vectorized strategies for both methods.

3.4.1 Wahby-Boneh method (RFC 9380)

RFC 9380 [FHSS+23] specifies the WB hash-to-curve method [WB19], which underlies the `blst` implementation. In `blst`, the function χ is instantiated as `expand_message_xmd`, which outputs a pseudorandom string of length $m \cdot 512$ bits for $m \in \{1, 2\}$ from an input message $M \in \{0,1\}^*$. The routine `hash_to_field` then parses this string into $u \in \mathbb{F}_{p^m}$. The admissible encoding H maps u to a curve point in two stages. First, two deterministically derived field elements are each mapped via the Simplified Shallue–van de Woestijne–Ulas (SSWU) map [Ula07] (`map_to_curve_sswu` in `blst`) to obtain $Q'_0, Q'_1 \in E'_m(\mathbb{F}_{p^m})$ on the isogenous curve $E'_m/\mathbb{F}_{p^m} : y^2 = x^3 + A'_m x + B'_m$ [WB19, Section 4.3]; their sum $Q' = Q'_0 + Q'_1$ is distributed uniformly over $E'_m(\mathbb{F}_{p^m})$. Second, an isogeny maps Q' to $Q \in E_m(\mathbb{F}_{p^m})$. In practice, the cost of `expand_message_xmd` is negligible, and the SSWU step is dominated by square-root extraction with limited SIMD potential; consequently, we focus on the remaining arithmetic. Specifically, the isogeny evaluation and cofactor clearing account for most of the vectorizable work in hash-to-curve and are the main targets of our vectorization strategy.

$\mathbb{F}_p$ Isogeny map from E'_1 to E_1 (`isogeny_map_to_E1`). The SSWU curve $E'_1/\mathbb{F}_p$ constructed in [WB19] is 11-isogenous to the BLS12-381 curve $E_1/\mathbb{F}_p$. Consequently, a point in $E'_1(\mathbb{F}_p)$ can be mapped to $E_1(\mathbb{F}_p)$ by evaluating a degree-11 isogeny, using Vélu's formulas [Vél71]. Given a point $Q' = (X, Y, Z) \in E'_1(\mathbb{F}_p)$ in Jacobian coordinates, the corresponding affine point $Q = (x_Q, y_Q) \in E_1(\mathbb{F}_p)$ is obtained as:

$$x_Q = \frac{x_{\text{num}}}{x_{\text{den}}} = \frac{\sum_{i=0}^{11} k_{1,i} X^i \left(Z^2\right)^{11-i}}{Z^2 \sum_{i=0}^{10} k_{2,i} X^i \left(Z^2\right)^{10-i}}, \quad y_Q = \frac{y_{\text{num}}}{y_{\text{den}}} = \frac{Y \sum_{i=0}^{15} k_{3,i} X^i \left(Z^2\right)^{15-i}}{Z^3 \sum_{i=0}^{15} k_{4,i} X^i \left(Z^2\right)^{15-i}}$$

Algorithm 5: `isogeny_map_to_E1` with 2-/4-way parallelization at $\mathbb{F}_p$ level.

Input: Point $Q' = (X, Y, Z) \in E_1'(\mathbb{F}_p)$, isogeny map coefficients $\{k_{i,j}\} \in \mathbb{F}_p$.
Output: Point $Q = (X_Q, Y_Q, Z_Q) \in E_1(\mathbb{F}_p)$ in Jacobian coordinates.

1	$x_n \leftarrow k_{1,11} \cdot X$	$z_{14} \leftarrow Z^2$	$y_n \leftarrow k_{3,15} \cdot Y$	
2	$z_{13} \leftarrow z_{14}^2$	$M_{14} \leftarrow z_{14} \cdot k_{4,14}$	$N_{14} \leftarrow z_{14} \cdot k_{3,14}$	$T_9 \leftarrow z_{14} \cdot k_{2,9}$
3	$y_d \leftarrow X + M_{14}$	$y_n \leftarrow y_n + N_{14}$	$x_d \leftarrow T_9 + X$	
4	$M_{13} \leftarrow z_{13} \cdot k_{4,13}$	$y_d \leftarrow y_d \cdot X$	$N_{13} \leftarrow z_{13} \cdot k_{3,13}$	$y_n \leftarrow y_n \cdot X$
5	$y_d \leftarrow y_d + M_{13}$	$y_n \leftarrow y_n + N_{13}$		
6	$z_{12} \leftarrow z_{13} \cdot z_{14}$	$y_d \leftarrow y_d \cdot X$	$y_n \leftarrow y_n \cdot X$	$z_{11} \leftarrow z_{13}^2$
7	$z_{10} \leftarrow z_{11} \cdot z_{14}$	$M_{12} \leftarrow z_{12} \cdot k_{4,12}$	$N_{12} \leftarrow z_{12} \cdot k_{3,12}$	$z_9 \leftarrow z_{11} \cdot z_{13}$
8	$y_d \leftarrow y_d + M_{12}$	$y_n \leftarrow y_n + N_{12}$		
9	$z_8 \leftarrow z_9 \cdot z_{14}$	$y_d \leftarrow y_d \cdot X$	$y_n \leftarrow y_n \cdot X$	$z_7 \leftarrow z_9 \cdot z_{13}$
10	$z_6 \leftarrow z_7 \cdot z_{14}$	$M_{11} \leftarrow z_{11} \cdot k_{4,11}$	$N_{11} \leftarrow z_{11} \cdot k_{3,11}$	$z_5 \leftarrow z_7 \cdot z_{13}$
11	$y_d \leftarrow y_d + M_{11}$	$y_n \leftarrow y_n + N_{11}$		
12	$z_4 \leftarrow z_5 \cdot z_{14}$	$y_d \leftarrow y_d \cdot X$	$y_n \leftarrow y_n \cdot X$	$z_3 \leftarrow z_5 \cdot z_{13}$
13	$z_2 \leftarrow z_3 \cdot z_{14}$	$M_{10} \leftarrow z_{10} \cdot k_{4,10}$	$N_{10} \leftarrow z_{10} \cdot k_{3,10}$	$z_1 \leftarrow z_3 \cdot z_{13}$
14	$y_d \leftarrow y_d + M_{10}$	$y_n \leftarrow y_n + N_{10}$		
15	$z_0 \leftarrow z_1 \cdot z_{14}$	$y_d \leftarrow y_d \cdot X$	$y_n \leftarrow y_n \cdot X$	$x_d \leftarrow x_d \cdot X$
16	**for** j **from** 10 **to** 4 **by** 1 **do**			
17	$U_j \leftarrow z_{j+4} \cdot k_{1,j}$	$M_{j-1} \leftarrow z_{j-1} \cdot k_{4,j-1}$	$N_{j-1} \leftarrow z_{j-1} \cdot k_{3,j-1}$	$T_{j-2} \leftarrow z_{j+3} \cdot k_{2,j-2}$
18	$x_n \leftarrow x_n + U_j$	$y_d \leftarrow y_d + M_{j-1}$	$y_n \leftarrow y_n + N_{j-1}$	$x_d \leftarrow x_d + T_{j-2}$
19	$x_n \leftarrow x_n \cdot X$	$y_d \leftarrow y_d \cdot X$	$y_n \leftarrow y_n \cdot X$	$x_d \leftarrow x_d \cdot X$
20	**end for**			
21	$U_3 \leftarrow z_7 \cdot k_{1,3}$	$M_2 \leftarrow z_2 \cdot k_{4,2}$	$M_1 \leftarrow z_1 \cdot k_{4,1}$	$T_1 \leftarrow z_6 \cdot k_{2,1}$
22	$x_n \leftarrow x_n + U_3$	$y_d \leftarrow y_d + M_2$	$x_d \leftarrow x_d + T_1$	
23	$x_n \leftarrow x_n \cdot X$	$y_d \leftarrow y_d \cdot X$	$M_0 \leftarrow z_0 \cdot k_{4,0}$	$x_d \leftarrow x_d \cdot X$
24	$y_d \leftarrow y_d + M_1$			
25	$z_t \leftarrow z_{14} \cdot z$	$y_d \leftarrow y_d \cdot X$	$N_2 \leftarrow z_2 \cdot k_{3,2}$	$T_0 \leftarrow z_5 \cdot k_{2,0}$
26	$y_d \leftarrow y_d + M_0$	$y_n \leftarrow y_n + N_2$	$x_d \leftarrow x_d + T_0$	
27	$U_2 \leftarrow z_6 \cdot k_{1,2}$	$y_d \leftarrow y_d \cdot z_t$	$y_n \leftarrow y_n \cdot X$	$x_d \leftarrow x_d \cdot z_{14}$
28	$x_n \leftarrow x_n + U_2$			
29	$x_n \leftarrow x_n \cdot X$	$Z_Q \leftarrow y_d \cdot x_d$	$N_1 \leftarrow z_1 \cdot k_{3,1}$	$N_0 \leftarrow z_0 \cdot k_{3,0}$
30	$y_n \leftarrow y_n + N_1$			
31	$U_1 \leftarrow z_5 \cdot k_{1,1}$	$X_Q \leftarrow Z_Q \cdot y_d$	$y_n \leftarrow y_n \cdot X$	$Y_Q \leftarrow Z_Q^2$
32	$x_n \leftarrow x_n + U_1$	$y_n \leftarrow y_n + N_0$		
33	$x_n \leftarrow x_n \cdot X$	$U_0 \leftarrow z_4 \cdot k_{1,0}$	$y_n \leftarrow y_n \cdot y$	$Y_Q \leftarrow Y_Q \cdot x_d$
34	$x_n \leftarrow x_n + U_0$			
35	$X_Q \leftarrow X_Q \cdot x_n$	$Y_Q \leftarrow Y_Q \cdot y_n$		
36	**return** $Q = (X_Q, Y_Q, Z_Q)$			

where the values $k_{i,j} \in \mathbb{F}_p$ are fixed constants defined in [FHSS+23, Appendix E.2]. The point Q is converted to Jacobian coordinates by computing:

$$Z_Q = x_{\text{den}} \cdot y_{\text{den}}, \quad X_Q = x_{\text{num}} \cdot y_{\text{den}} \cdot Z_Q, \quad Y_Q = y_{\text{num}} \cdot (x_{\text{den}})^3 \cdot (y_{\text{den}})^2 \tag{1}$$

The scalar implementation of `isogeny_map_to_E1` in `blst` incurs a cost of $9S_1 + 116M_1$, with performance dominated by the evaluation of the y-coordinate terms due to the higher degree of the associated polynomials (i.e., 15 versus 11). To mitigate this, our vectorized implementation (Algorithm 5) employs a 4-way parallel strategy for the vast majority of the computation, effectively interleaves the evaluation of these high-degree polynomials with the requisite Z-power exponentiations and fall back to 2-way parallelism only for the final reconstruction step. This scheduling maximizes multiplication throughput, reducing the total arithmetic cost to $31M_1^4 + 1M_1^2 + 18A_1^4 + 1A_1^2$. As detailed in Section 3.1, all vector additions in this procedure are executed without intermediate carry propagation or modular reduction.

Algorithm 6: `isogeny_map_to_E2` with 2-/4-way parallelization at $\mathbb{F}_{p^2}$ level.

Input: Point $Q' = (X, Y, Z) \in E_2'(\mathbb{F}_{p^2})$, isogeny map coefficients $\{k_{i,j}\} \in \mathbb{F}_{p^2}$.
Output: Point $Q = (X_Q, Y_Q, Z_Q) \in E_2$ in Jacobian coordinates.

1	$y_n \leftarrow X \cdot k_{3,3}$	$x_n \leftarrow X \cdot k_{1,3}$	$z_2 \leftarrow Z^2$	
2	$z_1 \leftarrow z_2^2$	$M_2 \leftarrow z_2 \cdot k_{4,2}$	$z_t \leftarrow z_2 \cdot Z$	$T_1 \leftarrow z_2 \cdot k_{2,1}$
3	$y_d \leftarrow X + M_2$	$x_d \leftarrow T_1 + X$		
4	$z_0 \leftarrow z_1 \cdot z_2$	$y_d \leftarrow y_d \cdot X$	$M_1 \leftarrow z_1 \cdot k_{4,1}$	$x_d \leftarrow x_d \cdot X$
5	$y_d \leftarrow y_d + M_1$			
6	$N_2 \leftarrow z_2 \cdot k_{3,2}$	$y_d \leftarrow y_d \cdot X$	$M_0 \leftarrow z_0 \cdot k_{4,0}$	$T_0 \leftarrow z_1 \cdot k_{2,0}$
7	$y_n \leftarrow y_n + N_2$	$y_d \leftarrow y_d + M_0$	$x_d \leftarrow x_d + T_0$	
8	$y_n \leftarrow y_n \cdot X$	$y_d \leftarrow y_d \cdot z_t$	$U_2 \leftarrow z_2 \cdot k_{1,2}$	$x_d \leftarrow x_d \cdot z_2$
9	$x_n \leftarrow x_n + U_2$			
10	$N_1 \leftarrow z_1 \cdot k_{3,1}$	$Z_Q \leftarrow y_d \cdot x_d$	$x_n \leftarrow x_n \cdot X$	$N_0 \leftarrow z_0 \cdot k_{3,0}$
11	$y_n \leftarrow y_n + N_1$			
12	$y_n \leftarrow y_n \cdot X$	$X_Q \leftarrow Z_Q \cdot y_d$	$U_1 \leftarrow z_1 \cdot k_{1,1}$	$Y_Q \leftarrow Z_Q^2$
13	$y_n \leftarrow y_n + N_0$	$x_n \leftarrow x_n + U_1$		
14	$y_n \leftarrow y_n \cdot Y$	$U_0 \leftarrow z_0 \cdot k_{1,0}$	$x_n \leftarrow x_n \cdot X$	$Y_Q \leftarrow Y_Q \cdot x_d$
15	$x_n \leftarrow x_n + U_0$			
16	$X_Q \leftarrow X_Q \cdot x_n$	$Y_Q \leftarrow Y_Q \cdot y_n$		
17	**return** $Q = (X_Q, Y_Q, Z_Q)$			

$\mathbb{F}_{p^2}$ Isogeny map from E_2' to E_2 (`isogeny_map_to_E2`). The SSWU curve $E_2'/\mathbb{F}_{p^2}$ constructed in [WB19] is 3-isogenous to $E_2/\mathbb{F}_{p^2}$, allowing for an efficient mapping of points via a degree-3 isogeny using Vélu's formulas [Vél71]. For a point $Q' = (X, Y, Z) \in E_2'(\mathbb{F}_{p^2})$ in Jacobian coordinates, the affine image $Q = (x_Q, y_Q) \in E_2(\mathbb{F}_{p^2})$ is given by:

$$x_Q = \frac{x_{\text{num}}}{x_{\text{den}}} = \frac{\sum_{i=0}^{3} k_{1,i} X^i \left(Z^2\right)^{3-i}}{Z^2\left(X^2 + \sum_{i=0}^{1} k_{2,i} X^i \left(Z^2\right)^{2-i}\right)}, y_Q = \frac{y_{\text{num}}}{y_{\text{den}}} = \frac{Y \sum_{i=0}^{3} k_{3,i} X^i \left(Z^2\right)^{3-i}}{Z^3\left(X^3 + \sum_{i=0}^{2} k_{4,i} X^i \left(Z^2\right)^{3-i}\right)}$$

where the coefficients $k_{i,j} \in \mathbb{F}_{p^2}$ are constants from [FHSS+23, Appendix E.3]. The point Q is converted to Jacobian coordinates via Equation 1. The `blst` implementation of the 3-isogeny requires $30M_2 + 3S_2$ operations. Our implementation (Algorithm 6) accelerates this by introducing a hybrid 2-/4-way strategy. By initially computing the factors for x_{den} and y_{den} while deferring the numerator terms, we fuse the polynomial evaluations and Z-power computations into a unified pipelined stream, resulting in a reduced computational cost of $8M_2^4 + 1M_2^2 + 6A_2^4 + 1A_2^2$ field operations in $\mathbb{F}_{p^2}$.

3.4.2 SwiftEC

The main difference between SwiftEC [CRT22] and the WB method lies in the construction of the function H. Unlike WB, SwiftEC directly maps field elements to the group $E_m(\mathbb{F}_{p^m})$, without the need of an auxiliary curve, thereby eliminating several field multiplications and avoiding exceptional cases. To do so, SwiftEC requires two pseudorandom field elements and one extra random bit. Therefore, we use the same function `expand_message_xmd` of `blst` to extract a bit string of size $(m \cdot 1024 + 1)$ bits, given a message $M \in \{0,1\}^*$. In addition, we use the `hash_to_field` function of `blst` to convert this bit string into one random bit $s \in \{0,1\}$ and two field elements u, t in $\mathbb{F}_p$ $(m = 1)$ or in $\mathbb{F}_{p^2}$ $(m = 2)$.

The SwiftEC admissible encoding to the elliptic curve groups $E_m(\mathbb{F}_{p^m})$ is defined as the function $H : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \times \{0,1\} \to E_m(\mathbb{F}_{p^m})$, where $m \in \{1,2\}$. Given $u, t \in \mathbb{F}_{p^m}$ and $s \in \{0,1\}$, the function computes a point Q on the curve $E_m/\mathbb{F}_{p^m} : y^2 = g_m(x) = x^3 + b_m$.

Algorithm 7: `SwiftEC` with hybrid 2-/4-way parallelization at $\mathbb{F}_p$ level.

Input: $u, t \in \mathbb{F}_p$, $s \in \{0,1\}$, $\tau = \sqrt{-3} \in \mathbb{F}_p$, $b = 4$.
Output: $Q = (x_Q, y_Q) \in E_1(\mathbb{F}_p)$.

1 $(h_0, h_1) \leftarrow (u^2, t^2)$
2 $(h_0, x_3) \leftarrow (h_0 \cdot u, u \cdot \tau)$
3 $(h_2, h_3) \leftarrow (h_0 + b, h_1 + h_1)$
4 $(h_2, \diagdown) \leftarrow (h_2 - h_1, \diagdown)$
5 $(h_3, \diagdown) \leftarrow (h_2 + h_3, \diagdown)$
6 $(v, x_3) \leftarrow (h_2 \cdot x_3, x_3 \cdot t_2)$
7 $(x_3, y) \leftarrow (x_3 + x_3, h_3 + h_3)$
8 $(x_1, w) \leftarrow (u \cdot h_3, x_3 \cdot h_3)$
9 $(x_1, w) \leftarrow (v - x_1, w + w)$
10 $(x_1, y) \leftarrow (x_1 \cdot x_3, y \cdot y)$
11 $w \leftarrow w^{-1}$
12 $(x_1, x_3) \leftarrow (x_1 \cdot w, y \cdot w)$
13 $(x_3, \diagdown) \leftarrow (x_3^2, \diagdown)$
14 $(x_2, x_3) \leftarrow (x_1 + u, x_3 + u)$
15 $(x_2, \diagdown) \leftarrow (-x_2, \diagdown)$
16 $(t, y_2, y_3, \diagdown) \leftarrow (x_1^2, x_2^2, x_3^2, \diagdown)$
17 $(t, y_2, y_3, \diagdown) \leftarrow (x_1 \cdot t, y_2 \cdot x_2, y_3 \cdot x_3, \diagdown)$
18 $(t, y_2, y_3, \diagdown) \leftarrow (t + b, y_2 + b, y_3 + b, \diagdown)$
19 $c_2 \leftarrow \texttt{IsSquare}(y_2), c_3 \leftarrow \texttt{IsSquare}(y_3)$
20 $\texttt{cswap}(x_1, x_2, c_2)$, $\texttt{cswap}(t, y_2, c_2)$
21 $\texttt{cswap}(x_1, x_3, c_3)$, $\texttt{cswap}(t, y_3, c_3)$
22 $t \leftarrow \sqrt{t}$
23 $\texttt{cswap}(t, -t, \texttt{sgn}(t) \wedge s)$
24 $x_Q \leftarrow x_1$, $y_Q \leftarrow t$
25 **return** $Q = (x_Q, y_Q)$

Algorithm 8: `SwiftEC` with hybrid 2-/4-way parallelization at $\mathbb{F}_{p^2}$ level.

Input: $u, t \in \mathbb{F}_{p^2}$, $s \in \{0,1\}$, $\tau = \sqrt{-3} \in \mathbb{F}_p$, $b = 4(1+i)$.
Output: $Q = (x_Q, y_Q) \in E_2(\mathbb{F}_{p^2})$.

1 $(x_1, y_1) \leftarrow (u^2, t^2)$
2 $(s_1[0], s_1[1], \diagdown, \diagdown) \leftarrow (u[0] \cdot \tau, u[1] \cdot \tau, \diagdown, \diagdown)$ // At $\mathbb{F}_p$ level
3 $(x_1, s_2) \leftarrow (x_1 \cdot u, s_1 \cdot t)$
4 $(x_1, h_3) \leftarrow (x_1 + b, y_1 + y_1)$
5 $(x_1, \diagdown) \leftarrow (x_1 - y_1, \diagdown)$
6 $(y_1, \diagdown) \leftarrow (x_1 + h_3, \diagdown)$
7 $(z_1, y) \leftarrow (s_2 + s_2, y_1 + y_1)$
8 $(w, y, v, x_1) \leftarrow (y_1 \cdot z_1, y^2, y_1 \cdot u, x_1 \cdot s_1)$
9 $(v, w) \leftarrow (x_1 - v, w + w)$
10 $w \leftarrow w^{-1}$
11 $(v, z_1) \leftarrow (z_1 \cdot v, y \cdot w)$
12 $(x_1, z_1) \leftarrow (w \cdot v, z_1^2)$
13 $(y_1, z_1) \leftarrow (x_1 + u, z_1 + u)$
14 $(y_1, \diagdown) \leftarrow (-y_1, \diagdown)$
15 $(t, u, v, \diagdown) \leftarrow (x_1^2, y_1^2, z_1^2, \diagdown)$
16 $(t, u, v, \diagdown) \leftarrow (x_1 \cdot t, y_1 \cdot u, z_1 \cdot v, \diagdown)$
17 $(t, u, v, \diagdown) \leftarrow (t + b, u + b, v + b, \diagdown)$
18 $c_2 \leftarrow \texttt{IsSquare}(u), c_3 \leftarrow \texttt{IsSquare}(v)$
19 $\texttt{cswap}(x_1, y_1, c_2)$, $\texttt{cswap}(t, u, c_2)$
20 $\texttt{cswap}(x_1, z_1, c_3)$, $\texttt{cswap}(t, v, c_3)$
21 $t \leftarrow \sqrt{t}$
22 $\texttt{cswap}(t, -t, \texttt{sgn}(t) \wedge s)$
23 $x_Q \leftarrow x_1$, $y_Q \leftarrow t$
24 **return** $Q = (x_Q, y_Q)$

The SwiftEC approach constructs three candidate x_Q-coordinates $x_1, x_2, x_3 \in \mathbb{F}_{p^m}$ as:

$$x_{1,2} = \frac{u\big(\tau(u^3 + b_m - t^2) \mp (u^3 + b_m + t^2)\big)}{2(u^3 + b_m + t^2)} \quad \text{and} \quad x_3 = \frac{3u^3t^2 - (u^3 + b_m + t^2)^2}{3u^2t^2}$$

where $\tau = \sqrt{-3} \in \mathbb{F}_{p^m}$, and computes the evaluations $v_1 = g_m(x_1)$, $v_2 = g_m(x_2)$ and $v_3 = g_m(x_3)$. The method guaranties that at least one of the three evaluations will be a perfect square in $\mathbb{F}_{p^m}$. Following the `RELIC` implementation of SwiftEC [AGM+], we take $x_Q = x_1$ as valid coordinate and $v_Q = v_1$ as valid evaluation, meaning that v_1 is a square in $\mathbb{F}_{p^m}$. Then we execute two square root tests for the evaluations v_2 and v_3. If some evaluation is a perfect square, x_Q and v_Q are updated. The method outputs the affine point $Q = (x_Q, y_Q)$, where $y_Q = (-1)^s\sqrt{v_Q}$ and s is the random bit given as input.

SwiftEC hashing to $\mathbb{G}_1$. We follow the implementation in `RELIC` [AGM+] (function `ep_map_swift_impl`) which we integrate in `blst`. This implementation requires $7S_1 + 12M_1 + 1I_1$ operations plus the cost of two quadratic residuosity tests and one square root extraction, which constitute the main bottleneck for SwiftEC. For the quadratic residuosity tests we use the function `blst_fp_is_square` of `blst`, which computes the Legendre symbol of $a \in \mathbb{F}_p$. For extracting the square root in $\mathbb{F}_p$ we use the `blst_fp_sqrt` function of `blst`, which computes the reciprocal square root. In Algorithm 7[13] we describe a hybrid 2-/4-way vectorized implementation of SwiftEC for hashing to $\mathbb{G}_1$, with a total cost of

[13]The names of subroutines in Algorithm 7 and 8, e.g., `IsSquare`, follow the same notation of [CRT22].

$1M_1^4+1S_1^4+1A_1^4+5M_1^2+2S_1^2+7A_1^2+1I_1$ operations, plus two Legendre symbol computations and one square root extraction. Algorithm 7 further parallelizes all independent field operations prior to inversion and square-root extraction, which are executed in scalar mode due to their inherent serial nature. Although the Legendre symbol algorithm can be performed in 2-way parallelization, it is difficult to utilize more than two lanes in AVX-512 registers; therefore, we did not consider this approach. Lines 16–18 intentionally use the 4-way SIMD structure while leaving one lane unused; nevertheless, this approach consistently outperforms a pure 2-way design due to the full use of AVX-512 register.

SwiftEC hashing to $\mathbb{G}_2$. We integrate the `RELIC` function `ep2_map_swift` in `blst`, which requires $7S_2 + 11M_2 + 2M_1 + 1I_2$ operations plus two quadratic residuosity tests and one square root extraction in $\mathbb{F}_{p^2}$. For the quadratic residuosity in $\mathbb{F}_{p^2}$ we use the `blst` function `blst_fp2_is_square`, which, given an element $a = a_0+a_1 i \in \mathbb{F}_{p^2}$, checks whether $a_0^2+a_1^2$ is a square in $\mathbb{F}_p$. This function performs $2S_1+1A_1$ and one Legendre computation in $\mathbb{F}_p$. For extracting the square root in $\mathbb{F}_{p^2}$ we use the `blst` function `blst_fp2_sqrt` which computes the reciprocal square root with $2S_1 + 2M_1 + 3A_1$ plus one call of `blst_fp_sqrt` and one call of `recip_sqrt_fp`. In Algorithm 8 we present our 4-way to 2-way vectorized variant of SwiftEC for hashing to $\mathbb{G}_2$ with a cost of $2M_2^4+1S_2^4+1A_2^4+3M_2^2+1S_2^2+7A_2^2+1M_1^2+1I_2$, plus two quadratic residuosity tests and one square root extraction in $\mathbb{F}_{p^2}$.

3.4.3 Cofactor clearing

The final step maps a point $Q \in E_m(\mathbb{F}_{p^m})$ to the prime-order subgroup via $P = [h_m]Q$. To avoid expensive scalar multiplications by the full cofactors, we adopt the optimization trick of Wahby-Boneh [WB19] for $\mathbb{G}_1$ and the endomorphism-based optimization of Budroni-Pintore [BP22] for $\mathbb{G}_2$. These methods decompose the operation into smaller sparse scalar multiplications, which depend on the parameter z, allowing us to reuse the vectorized point arithmetic kernels described in Section 3.2. Since h_1 is a perfect square and $\sqrt{h_1}$ divides $p-1$, the trick of Wahby and Boneh [WB19] simplifies the cofactor clearing in $\mathbb{G}_1$ to a single sparse multiplication $P = [z-1]Q$. For $\mathbb{G}_2$, computing $[h_2]Q$ is optimized using the endomorphism ψ on the twisted curve [BP22] (see also Section 3.3). The computation is reduced to a linear combination of efficient operations $P = [z^2 - z - 1]Q + [z - 1]\psi(Q) + [2]\psi^2(Q)$. This formulation replaces the multiplication by the large cofactor h_2 with operations bounded by the bit-length of z, significantly improving performance.

4 Evaluation

Benchmarking setup. Our benchmarking experiments were conducted on an Intel Ice Lake Core i3-1005G1 CPU, clocked at 1.2 GHz. In addition, the source code of `blst` and our `avxbls-sig` were compiled using GCC 13.3, under optimization level `-O3`, and TurboBoost was disabled when measuring the execution time.

Compatibility with standards. We ensure compatibility with existing standards and reference implementations. All APIs are validated against `blst`, with identical outputs observed across extensive testing, and our hash-to-curve implementation matches the RFC 9380 test vectors. In addition, we support both a WB-compatible mode (aligned with RFC 9380 and [BBG+25]) and a SwiftEC mode, where the latter provides a non-RFC alternative for improved performance.

Constant-time implementation. Our `avxbls-sig` meets the usual requirements for constant-time implementation: it possesses the fixed operation sequence and does not contain any conditional statements that rely on secret data. In particular, one noteworthy

Table 3: Benchmarks of the low-level operations in `blst` (x64 assembly) and `avxbls-sig` (AVX-512, various Algorithms) on an Intel Core i3-1005G1 CPU.

Operation	Implementation	E_1 cycles	E_2 cycles
point doubling	`blst`	1154 (1.00×)	2634 (1.00×)
	Algorithm 2	867 (1.33×)	1758 (1.50×)
	Algorithm 9	-	1893 (1.39×)
point addition	`blst`	2422 (1.00×)	6338 (1.00×)
	Algorithm 3	1285 (1.89×)	2897 (2.19×)
	Algorithm 11	-	3211 (1.97×)
	Algorithm 10	-	3597 (1.76×)
point DADD	`blst`	2615 (1.00×)	7108 (1.00×)
	Algorithm 4	1403 (1.86×)	3088 (2.30×)
	Algorithm 12	-	3879 (1.83×)
isogeny map	`blst`	17731 (1.00×)	13206 (1.00×)
	Algorithm 5/6	8324 (2.13×)	5688 (2.32×)

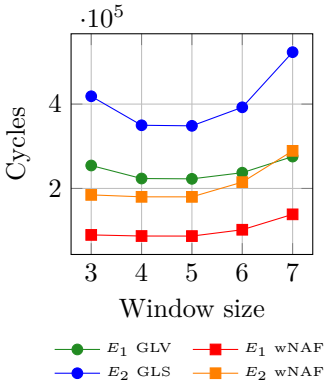

Figure 2: Cycle counts of vectorized scalar multiplications under different window sizes.

detail is that, as described in Section 3.3, during table lookups, we always check the whole table to prevent cache timing attacks.

Low level: curve operations. Table 3 presents the cycle counts for three low-level elliptic curve operations, as well as the computation and evaluation of the degree-11 and 3 isogenies used in WB hash-to-curve, i.e., isogeny maps to E_1 and E_2, respectively. Compared to the x64 assembly implementation in `blst`, our vectorized routines demonstrate promising speedups: most of the operations on E_2 curve achieve a speedup factor of more than two. This is slightly higher than that of their E_1 counterparts, because the use of $\mathbb{F}_{p^2}$ layer (by E_2) offers more opportunities for vectorization than using only the $\mathbb{F}_p$ layer (by E_1). On the other hand, point doubling benefits less from vectorization (i.e., 1.33× on E_1 and 1.50× on E_2), due to tighter dependencies among its underlying field operations.

Mid level: scalar multiplication. As described in Section 3.3, we evaluated different window sizes for the precomputation table used in our vectorized scalar multiplications. Figure 2 reports the cycle counts for GLV/GLS using a full-size scalar in $\mathbb{F}_r$ and wNAF using a 64-bit scalar. Although point addition benefits more from vectorization than point doubling (see Table 3), the optimal window size remains $w = 5$.

Mid level: hash-to-curve. Table 4 presents the benchmarking results of the mid- and high-level operations. As discussed in Section 3.4, we consider two hash-to-curve methods, namely WB and SwiftEC; SwiftEC is faster in both the scalar and vector settings. Moreover, vectorization yields a higher speedup for hash-to-curve on $\mathbb{G}_2$ than $\mathbb{G}_1$ (1.66× vs. 1.46×), because 1) a larger portion of the hash-to-curve on $\mathbb{G}_2$ is vectorized (see Table 2) and 2) the low-level operations used by $\mathbb{G}_2$ possess a larger speedup than that of $\mathbb{G}_1$ (per Table 4).

In addition, Aranha et al. [AHST23] present an optimized scalar implementation of SwiftEC-based hash-to-curve on BLS12-381, reporting a larger speedup (2.05×) compared to the counterpart in our implementation (1.26×). In fact, this is because their used baseline implementation is *substantially different* from our case. In detail, per [AHST23, Table 2], their baseline is based on constant-time SWU with generic exponentiation (implemented in C) for quadratic residuosity and square-root computations, while their optimized implementation is based on SwiftEC with JumpDivsteps approach (implemented in assembly), both parts (i.e., hashing and the field arithmetic) were improved. In contrast,

Table 4: Benchmarks of the mid-level and high-level operations in `blst` (x64 assembly) and `avxbls-sig` (AVX-512) on an Intel Core i3-1005G1 CPU.

Operation	Implementation	Method	$\mathbb{G}_1$ cycles	$\mathbb{G}_2$ cycles
scalar multiplication	`blst`	GLS/GLV	387,478 (1.00×)	725,531 (1.00×)
	`avxbls-sig`		222,419 (1.74×)	353,161 (2.05×)
	`blst`	wNAF	137,972 (1.00×)	323,871 (1.00×)
	`avxbls-sig`		87,241 (1.58×)	179,661 (1.80×)
	`blst`	subgroup testing	172,544 (1.00×)	206,072 (1.00×)
	`avxbls-sig`		124,276 (1.39×)	126,692 (1.63×)
hash-to-curve	`blst`	WB	233,057 (1.00×)	724,388 (1.00×)
	`blst`	SwiftEC	185,259†(1.26×)	608,432†(1.19×)
	`avxbls-sig`	WB	198,994 (1.17×)	544,249 (1.33×)
	`avxbls-sig`	SwiftEC	159,621 (1.46×)	435,366 (1.66×)
key generation	`blst`	-	387,478 (1.00×)	725,531 (1.00×)
	`avxbls-sig`	-	222,419 (1.74×)	353,161 (2.05×)
signing	`blst`	WB	620,710 (1.00×)	1,448,372 (1.00×)
	`blst`	SwiftEC	572,439†(1.08×)	1,351,024†(1.07×)
	`avxbls-sig`	WB	421,626 (1.47×)	889,010 (1.63×)
	`avxbls-sig`	SwiftEC	383,591 (1.62×)	783,187 (1.85×)
aggregation (L1)	`blst`	WB	1,331,298 (1.00×)	1,834,576 (1.00×)
	`blst`	SwiftEC	1,288,370†(1.03×)	1,707,089†(1.07×)
	`avxbls-sig`	WB	887,915 (1.50×)	1,235,908 (1.48×)
	`avxbls-sig`	SwiftEC	844,640 (1.58×)	1,121,705 (1.64×)
aggregation (L2)	`blst`	WB	1,598,511 (1.00×)	2,289,181 (1.00×)
	`blst`	SwiftEC	1,555,600†(1.03×)	2,182,367†(1.05×)
	`avxbls-sig`	WB	1,059,100 (1.51×)	1,508,535 (1.52×)
	`avxbls-sig`	SwiftEC	1,021,200 (1.57×)	1,400,269 (1.64×)
final verification	`blst`	-	2,398,751 (1.00×)	2,398,751 (1.00×)
	`avxbls-sig`*	-	1,309,851 (1.83×)	1,309,851 (1.83×)

† Our scalar implementation that uses the SwiftEC-based hash-to-curve in `blst`.
* Our vectorized final verification is fully based on the subroutines provided by `avxbls-oap` (e.g., Miller loop and final exponentiation).

the field arithmetic operations in our baseline are already the highly-optimized assembly implementations, including, e.g., the Pornin's algorithm [Por20] for quadratic residuosity and the dedicated addition chains for square-root computations. As a result, the speedup data reported in [AHST23] cannot be directly compared with ours.

High-level: API. Thanks to the layered structure of BLS, the low- and mid-level improvements propagate well up to the API: the high-level operations are finally accelerated by at least 1.57× on $\mathbb{G}_1$ and 1.64× on $\mathbb{G}_2$. These gains are lower than those of `avxbls-oap` (1.83×, per Table 4), which focuses exclusively on vectorizing the optimal ate pairing. The reason is that the $\mathbb{F}_{p^{12}}$ tower extension used by pairing offers better opportunities to make full and efficient use of eight parallel lanes of AVX-512. In contrast, in this work, the $E_1(\mathbb{F}_p)$ operations have to be vectorized with the less efficient 256-bit instructions, and most curve operations cannot be vectorized in completely 4-way.

Speedups attributed to the vectorization strategies in this work. As stated in Table 4, the speedup of the vectorized implementation of final verification is fully due to `avxbls-oap`, i.e., [CFGP25]. However, recalling Section 2.1, we stress that the final verification is only *part of* the aggregated verification, which contains also the aggregate function (computes aggregated signature, hash-to-curve, and Miller functions). Therefore, the speedup of

vectorized implementation of the complete aggregated verification does not rely only on the pairing optimization in [CFGP25], but also (significantly) on vectorized operations presented in this work. Quantitatively, as indicated by Table 1, the Miller loop and final exponentiation take 0% in key generation and signing, less than 50% in L1/L2 aggregation, and nearly 100% in final verification. We thus conclude: regarding the results in Table 4 at the API level, 1) the speedups of key generation and signing are *entirely* due to the vectorization strategies presented in this work; 2) the speedups of aggregation (in aggregated verification) are *mostly* from this work; 3) the speedup of final verification (in aggregated verification) is derived from the integration of [CFGP25].

5 Conclusion

Summary. In this paper, we demonstrated that the enormous parallel computing capability of AVX-512 can be effectively leveraged to accelerate the entire BLS signature on the BLS12-381 curve. In addition to incorporating the vectorized finite-field and pairing implementations of [CFGP25] and the SwiftEC hash-to-curve of [CRT22], we fine-tuned the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ implementations (i.e., using lazy reduction in a wider range and AVX-512VL-based optimization), proposed various vectorization strategies for new curve formulas (i.e., point doubling, addition, and DADD), re-evaluated the optimal window size in looking up wNAF precomputation table under vectorized setting, and presented sophisticated vectorization strategies for WB and SwiftEC hash-to-curve computations. Through the combined application of these optimization techniques, we improved the latency of the BLS scheme by a factor of at least 1.57 on an Intel Ice Lake Core CPU.

More broadly, our results highlight that full-width vectorization is not always optimal, and that judicious transitions between different granularities of parallelism are essential for maximizing performance on modern microarchitectures. Regarding the extensibility, on one hand, the arithmetic techniques developed herein are not bound to a specific security level; the BLS scheme can be easily scaled to meet higher security requirements when instantiated with pairing-friendly curves suggested, e.g., in [AFG24, MAF20]. On the other hand, our vectorization techniques can be extended to other platforms, which usually equip smaller SIMD computing power than eight-lane AVX-512 (e.g., ARM NEON), by simply adopting the vectorization strategies for curve operations directly while developing a reduced-lane version of $\mathbb{F}_{p^2}$ implementation.

Future work. At present, the majority of AVX-512 cryptographic software focuses on optimization and evaluation at the primitive level; however, their impact on real-world applications remains largely unexplored, with only a few studies investigating this aspect (e.g., [ZHZ+24, ZZD+24]). Therefore, an important direction for future work is to integrate our vectorized BLS implementation into a blockchain system and examine to what extent the resulting efficiency gains translate to application-level performance. Accordingly, we will further optimize our software using techniques such as multiprocessing to fully exploit all available parallelism at the system level.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62502275), in part by the Horizon Digital Europe programme (managed by the EU Cybersecurity Competence Centre-ECCC) under the project PiQASO (GA No. 101190366), and in part by the Fonds National de la Recherche (FNR) Luxembourg via the CORE project CryptoFin (C22/IS/17415825).

References

[AFG24] D.F. Aranha, G. Fotiadis, and A. Guillevic. A short-list of pairing-friendly curves resistant to the special TNFS algorithm at the 192-bit security level. *IACR Communications in Cryptology (CIC)*, 1(3):44, 2024. https://doi.org/10.62056/angyl86bm.

[AGM+] D.F. Aranha, C.P.L. Gouvêa, T. Markmann, R.S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[AHST23] D.F. Aranha, B.S. Hvass, B. Spitters, and M. Tibouchi. Faster constant-time evaluation of the Kronecker symbol with application to elliptic curve hashing. In *Computer and Communications Security (CCS)*, pages 3228–3238. ACM, 2023. https://doi.org/10.1145/3576915.3616597.

[ANS21] J.P. Aumasson, Q.T.M. Nguyen, and A. Sanso. Security of BLS batch verification, 2021. https://ethresear.ch/t/security-of-bls-batch-verification/10748.

[AR19] A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 673–686. ACM, 2019. https://doi.org/10.1145/3297858.3304062.

[BBG+25] D. Boneh, J. Bradley, S. Gorbunov, R.S. Wahby, H. Wee, C.A. Wood, and Z. Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-06, Internet Engineering Task Force, 2025. https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/06/.

[BCI+10] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology (CRYPTO)*, LNCS 6223, pages 237–254. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-14623-7_13.

[BF01] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Advances in Cryptology (CRYPTO)*, LNCS 2139, pages 213–229. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44647-8_13.

[BLS01] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology (ASIACRYPT)*, LNCS 2248, pages 514–532. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-45682-1_30.

[BLS02] P.S.L.M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks (SCN)*, LNCS 2576, pages 257–267. Springer-Verlag, 2002. https://doi.org/10.1007/3-540-36413-7_19.

[Boo51] A.D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[Bow17] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction. Electric Coin Co. Blog Post, 2017. https://electriccoin.co/blog/new-snark-curve.

[BP22] A. Budroni and F. Pintore. Efficient hash maps to G2 on BLS curves. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 33(3):261–281, 2022. https://doi.org/10.1007/s00200-020-00453-9.

[But19] V. Buterin. Fast verification of multiple BLS signatures, 2019. https://ethresear.ch/t/fast-verification-of-multiple-bls-signatures/5407.

[CFGP25] H. Cheng, G. Fotiadis, J. Großschädl, and D. Page. Fast AVX-512 implementation of the optimal ate pairing on BLS12-381. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2025(4):848–872, 2025. https://doi.org/10.46586/tches.v2025.i4.848-872.

[CFGR22] H. Cheng, G. Fotiadis, J. Großschädl, and P.Y.A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022. https://doi.org/10.46586/tches.v2022.i2.41-68.

[CRT22] J. Chávez-Saab, F. Rodríguez-Henríquez, and M. Tibouchi. SwiftEC: Shallue-van de woestijne indifferentiable function to elliptic curves - faster indifferentiable hashing to elliptic curves. In *Advances in Cryptology (ASIACRYPT)*, LNCS 13791, pages 63–92. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-22963-3_3.

[Edg23] B. Edgington. Upgrading Ethereum: A technical handbook on Ethereum's move to proof of stake and beyond. Edition 0.3: Capella [WIP], 2023. https://eth2book.info/capella.

[Eng25] J. England. Crypto signatures and why chia chose BLS. Chia blog, 2025. https://www.chia.net/2025/11/25/crypto-signatures-and-why-chia-chose-bls/.

[FHSS+23] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C.A. Wood. Hashing to Elliptic Curves. Internet Engineering Task Force (IETF) Request for Comments (RFC) 9380, 2023. https://www.rfc-editor.org/info/RFC:9380.

[FIP23] Digital Signature Standard (DSS). National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 186-5, 2023. https://csrc.nist.gov/pubs/fips/186-5/final.

[FM25] K.M. Faraoun and N.E. Mrabet. Optimizing and securing GLV multiplication over BLS pairings-friendly curves. *Designs, Codes and Cryptography (DCC)*, 93(9):3641–3669, 2025. https://doi.org/10.1007/s10623-025-01646-4.

[GLS09] S.D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *Advances in Cryptology (EUROCRYPT)*, LNCS 5479, pages 518–535. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-01001-9_30.

[GLV01] R.P. Gallant, R.J. Lambert, and S.A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology (CRYPTO)*, LNCS 2139, pages 190–200. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44647-8_11.

[Gro16] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology (EUROCRYPT)*, LNCS 9666, pages 305–326. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-49896-5_11.

[Jou04] A. Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology (JOC)*, 17(4):263–276, 2004. https://doi.org/10.1007/s00145-004-0312-y.

[JT01] M. Joye and C. Tymen. Compact encoding of non-adjacent forms with applications to elliptic curve cryptography. In *Public Key Cryptography (PKC)*, LNCS 1992, pages 353–364. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44586-2_26.

[Kos25] D. Koshelev. Simultaneously simple universal and indifferentiable hashing to elliptic curves. In *Progress in Cryptology (AFRICACRYPT)*, LNCS 15651, pages 395–413. Springer-Verlag, 2025. https://doi.org/10.1007/978-3-031-97260-7_18.

[KZG10] A. Kate, G.M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology (ASIACRYPT)*, LNCS 6477, pages 177–194. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-17373-8_11.

[LL97] C.H. Lim and P.J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology (CRYPTO)*, pages 249–263. Springer-Verlag, 1997. https://link.springer.com/chapter/10.1007/BFb0052240.

[LM08] P. Longa and A. Miri. New composite operations and precomputation scheme for Elliptic Curve Cryptosystems over prime fields. In *Public Key Cryptography (PKC)*, LNCS 4939, pages 229–247. Springer-Verlag, 2008. https://doi.org/10.1007/978-3-540-78440-1_14.

[Lon23] P. Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(3):445–472, 2023. https://doi.org/10.46586/tches.v2023.i3.445-472.

[MAF20] N.B. Mbang, D.F. Aranha, and E. Fouotsa. Computing the optimal ate pairing over elliptic curves with embedding degrees 54 and 48 at the 256-bit security level. *International Journal of Applied Cryptography (IJACT)*, 4(1):45–59, 2020. https://doi.org/10.1504/IJACT.2020.107167.

[Por20] T. Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020. https://eprint.iacr.org/2020/972.

[RY07] T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology (EUROCRYPT)*, LNCS 4515, pages 228–245. Springer-Verlag, 2007. https://doi.org/10.1007/978-3-540-72540-4_13.

[Sco21] M. Scott. A note on group membership tests for G1, G2 and GT on BLS pairing-friendly curves. Cryptology ePrint Archive, Paper 2021/1130, 2021. https://eprint.iacr.org/2021/1130.

[Ula07] M. Ulas. Rational points on certain hyperelliptic curves over finite fields. *Bulletin of the Polish Academy of Sciences. Mathematics*, 55(2):97–104, 2007. https://doi.org/10.4064/ba55-2-1.

[Vél71] J. Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris, Série A*, 273(4):238–241, 1971.

[Ver10] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory (TIT)*, 56(1):455–461, 2010. https://doi.org/10.1109/TIT.2009.2034881.

[WB19] R.S. Wahby and D. Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(4):154–179, 2019. https://doi.org/10.13154/tches.v2019.i4.154-179.

[ZHZ+24] J. Zhang, J. Huang, L. Zhao, D. Chen, and Ç.K. Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In *USENIX Security Symposium*, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-jipeng.

[ZZD+24] J. Zheng, H. Zhu, Y. Dong, Z. Song, Z. Zhang, Y. Yang, and Y.Zhao. Faster post-quantum TLS 1.3 based on ML-KEM: implementation and assessment. In *European Symposium on Research in Computer Security (ESORICS)*, LNCS 14983, pages 123–143. Springer-Verlag, 2024. https://doi.org/10.1007/978-3-031-70890-9_7.

A Additional vectorization strategies

Algorithm 9: `point_dbl` with 2-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: $P = (X_P, Y_P, Z_P)$
Output: $R = [2]P = (X_R, Y_R, Z_R)$

1	$A \leftarrow X_P^2$	$B \leftarrow Y_P^2$
2	$t_0 \leftarrow X_P + B$	$s_0 \leftarrow A + A$
3	$t_0 \leftarrow t_0^2$	$C \leftarrow B^2$
4	$t_0 \leftarrow t_0 - A$	$E \leftarrow s_0 + A$
5	$t_0 \leftarrow t_0 - C$	$s_0 \leftarrow C + C$
6	$D \leftarrow t_0 + t_0$	$s_0 \leftarrow s_0 + s_0$
7	$F \leftarrow E^2$	
8	$t_0 \leftarrow F - D$	$s_0 \leftarrow s_0 + s_0$
9	$X_R \leftarrow t_0 - D$	$zz \leftarrow Z_P + Z_P$
10	$t_0 \leftarrow D - X_R$	
11	$t_0 \leftarrow E \cdot t_0$	$Z_R \leftarrow zz \cdot Y_R$
12	$Y_R \leftarrow t_0 - s_0$	
13	**return** $R = (X_R, Y_R, Z_R)$	

Algorithm 10: `point_add` with 2-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: P, Q in Jacobian coordinates.
Output: $R = P + Q = (X_R, Y_R, Z_R)$

1	$ZZ_P \leftarrow Z_P^2$	$ZZ_Q \leftarrow Z_Q^2$
2	$S_2 \leftarrow Z_P \cdot ZZ_P$	$S_1 \leftarrow Z_Q \cdot ZZ_Q$
3	$S_2 \leftarrow Y_Q \cdot S_2$	$S_1 \leftarrow Y_P \cdot S_1$
4	$U_1 \leftarrow X_P \cdot ZZ_Q$	$U_2 \leftarrow X_Q \cdot ZZ_P$
5	$H \leftarrow U_2 - U_1$	$t_0 \leftarrow S_2 - S_1$
6	$t_1 \leftarrow H + H$	$R \leftarrow t_0 + t_0$
7	$I \leftarrow t_1^2$	$t_0 \leftarrow R^2$
8	$J \leftarrow H \cdot I$	$V \leftarrow U_1 \cdot I$
9	$t_1 \leftarrow t_0 - J$	$Z_R \leftarrow Z_P + Z_Q$
10	$t_2 \leftarrow S_1 \cdot J$	$Z_R \leftarrow Z_R \cdot Z_R$
11	$X_R \leftarrow t_1 - V$	$Z_R \leftarrow Z_R - ZZ_P$
12	$X_R \leftarrow X_R - V$	$Z_R \leftarrow Z_R - ZZ_Q$
13	$Y_R \leftarrow V - X_R$	
14	$Z_R \leftarrow H \cdot Z_R$	$Y_R \leftarrow R \cdot Y_R$
15	$Y_R \leftarrow Y_R - t_2$	
16	$Y_R \leftarrow Y_R - t_2$	
17	**return** $R = (X_R, Y_R, Z_R)$	

Algorithm 11: `point_add` with hybrid 2-/4-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: Points $P = (X_P, Y_P, Z_P)$, $Q = (X_Q, Y_Q, Z_Q)$ in Jacobian coordinates.
Output: Point $R = P + Q = (X_R, Y_R, Z_R)$ in Jacobian coordinates.

1	$ZZ_P \leftarrow Z_P^2$	$ZZ_Q \leftarrow Z_Q^2$	$t_0 \leftarrow Y_P \cdot Z_Q$	$s_0 \leftarrow Y_Q \cdot Z_P$
2	$t_0 \leftarrow t_0 \cdot ZZ_Q$	$s_1 \leftarrow X_Q \cdot ZZ_P$	$t_1 \leftarrow X_P \cdot ZZ_Q$	$s_0 \leftarrow s_0 \cdot ZZ_P$
3	$t_2 \leftarrow s_0 - t_0$	$s_2 \leftarrow s_1 - t_1$	$s_3 \leftarrow ZZ_P + ZZ_Q$	$t_3 \leftarrow Z_P + Z_Q$
4	$t_4 \leftarrow t_2 + t_2$	$s_4 \leftarrow s_2 + s_2$	9 $X_R \leftarrow X_R - Y_R$	
5	$t_5 \leftarrow t_4^2$	$s_5 \leftarrow s_4^2$	10 $X_R \leftarrow X_R - Y_R$	
6	$t_6 \leftarrow s_5 \cdot s_2$	$Z_R \leftarrow t_3 \cdot t_3$	11 $Y_R \leftarrow Y_R - X_R$	$s_5 \leftarrow t_5 + t_5$
7	$X_R \leftarrow t_5 - t_6$	$Z_R \leftarrow Z_R - s_3$	12 $Y_R \leftarrow Y_R \cdot t_4$	$Z_R \leftarrow Z_R \cdot s_2$
8	$t_5 \leftarrow t_0 \cdot t_6$	$Y_R \leftarrow t_1 \cdot s_5$	13 $Y_R \leftarrow Y_R - s_5$	
14	**return** $R = (X_R, Y_R, Z_R)$			

Algorithm 12: `point_DADD` with 2-way parallelization at $\mathbb{F}_{p^m}$ level.

Input: Points $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$ in Jacobian coordinates.
Output: Point $R = P \oplus Q = (X_R, Y_R, Z_R)$ in Jacobian coordinates.

1	$d_T \leftarrow X_P + X_P$	$d_U \leftarrow Y_P + Y_P$	5 $a_S \leftarrow Z_P \cdot Y_Q$	$t_1 \leftarrow Z_Q \cdot Y_P$
2	$w_0 \leftarrow X_P^2$	$ZZ \leftarrow Z_P \cdot Z_Q$	6 $a_S \leftarrow a_S \cdot t_0$	$S_1 \leftarrow t_1 \cdot s_0$
3	$d_S \leftarrow w_0 + w_0$		7 $a_U \leftarrow X_Q \cdot t_0$	$U_1 \leftarrow X_P \cdot s_0$
4	$t_0 \leftarrow Z_P^2$	$s_0 \leftarrow Z_Q^2$	8 $A_T \leftarrow a_U + U_1$	$A_U \leftarrow a_U - U_1$
9	$A_S \leftarrow a_S - S_1$	$d_S \leftarrow d_S + w_0$		
10	$(S, U, T) \leftarrow$ `cselect`$((d_S, d_U, d_T), (A_S, A_U, A_T),$ `IsZero`$(a_U))$			
11	$(U_1, S_1, ZZ) \leftarrow$ `cselect`$((X_P, Y_P, Z_P), (U_1, S_1, ZZ),$ `IsZero`$(a_U))$			
12	$U_2 \leftarrow U^2$	$S_2 \leftarrow S^2$	16 $Y_R \leftarrow Y_R - X_R$	
13	$t_1 \leftarrow U_2 \cdot T$	$S \leftarrow U \cdot U_2$	17 $Y_R \leftarrow Y_R \cdot S_2$	$S \leftarrow S \cdot S_1$
14	$Z_R \leftarrow ZZ \cdot U$	$Y_R \leftarrow U_2 \cdot U_1$	18 $Y_R \leftarrow Y_R - S$	
15	$X_R \leftarrow S_2 - t_1$		19 **return** $R = (X_R, Y_R, Z_R)$	