Compressed Key Exchange Protocol from Orientations of Large Discriminant Using AVX-512

Yuhao Zheng[1], Jianming Lin[1,2(✉)], Yutong Liang[1], Yanzhen Ren[1], Huixin Zhang[1] and Chang-An Zhao[1,3]

[1] School of Mathematics, Sun Yat-sen University, Guangzhou, 510275, China.
zhengyh57@mail2.sysu.edu.cn
linjm76@mail.sysu.edu.cn
liangyt83@mail2.sysu.edu.cn
renyzh6@mail2.sysu.edu.cn
zhanghx225@mail2.sysu.edu.cn
zhaochan3@mail.sysu.edu.cn

[2] School of Mathematics, Jiaying University, Meizhou, 514015, China.

[3] Guangdong Key Laboratory of Information Security, Guangzhou, 510006, China.

Abstract. CSIDH (Commutative Supersingular Isogeny Diffie–Hellman) is a class-group-based key-exchange protocol operated on supersingular elliptic curves, which, at the time of its proposal, exhibited several attractive selling points such as non-interactivity. Unfortunately, CSIDH is vulnerable to the sub-exponentiation attack–Kuperberg's algorithm, thereby requiring large parameters to ensure security. A recent work based on oriented elliptic curves with large discriminants, proposed by Houben, allows for a significantly small base field (around 255 bits). We name this protocol CSIDH-LDO. However, the practicality of CSIDH-LDO is currently hindered by the necessity of performing multiple group actions and its huge public-key size.

In this paper, we address these bottlenecks by presenting highly optimized constant-time implementations alongside an effective public-key compression framework for CSIDH-LDO. We combine algorithmic improvements, specifically scalar multiplication by differential addition chains and isogeny computation on the twisted Edwards model, with *limb-slicing* to exploit parallelism via Intel's AVX-512 instructions. To resolve the architectural mismatch when mapping $r \in \{7, 13\}$ group actions to 8-lane SIMD registers, we evaluate two parallel scheduling strategies. While a dummy-based approach allows us to leverage efficient $\mathbb{F}_{p^2}$-arithmetic, our dummy-free strategy systematically exploits internal parallelism to fully saturate the 512-bit vectors without redundant computations. Benchmarks on an Intel Tiger Lake processor demonstrate our parallel implementations achieve speedups of up to 3.40× (AVX-512F) and 7.30× (AVX-512IFMA) over an optimized x64 assembly baseline. Furthermore, we establish a formal framework for public-key compression tailored for CSIDH-LDO. Specifically, we propose two efficient techniques that achieve a compact public key representation of approximately $(r + 2)\log_2(p)$ bits, thereby significantly reducing communication bandwidth. To the best of our knowledge, this work is the first to demonstrate the efficacy of SIMD parallelization combined with optimized compression for isogeny-based protocols derived from orientations of large discriminants.

Keywords: Post-Quantum Cryptography · Isogeny-Based Cryptography · Key Exchange Protocols from Orientations · AVX-512 · Public-Key Compression · Constant-Time Implementation

1 Introduction

Isogeny-based schemes serve as the fundamental block of post-quantum cryptography due to their small public-key size. The first isogeny-based protocol is the CRS scheme, proposed by Couveignes, Rostovtsev, and Stolbunov in [Cou06, RS06]. This non-interactive key exchange scheme is based on class group actions on ordinary curves. Nevertheless, it suffers from low efficiency due to the difficulty of constructing suitable parameters on ordinary curves. CSIDH [CLM+18] is a variant of CRS, operating on supersingular curves and yielding a crucially more efficient performance. A majority of research has focused on establishing the optimizations and constant-time variants of this protocol [BBC+21, CHMR25, CDV20, CFG+21, HLKA20, MCR19, MR18]. The main bottleneck of CSIDH lies in its vulnerability to subexponential-time attacks—Kuperberg's algorithm [Kup05], leading to the recent works adopting large parameters. Present-day estimates [CCSCD+24, CCJR22] illustrate that the required size of the characteristic p for CSIDH should be 4096 bits to achieve the NIST level 1, which is unsatisfactory as it dominates the efficiency of the field arithmetic. Consequently, CSIDH is currently impractical.

The complexity of Kuperberg's algorithm is associated with the order of the class group $\mathrm{Cl}(\mathcal{O})$, where $\mathcal{O}$ is an imaginary order. In the situation of CSIDH, it holds that $\mathcal{O} = \mathbb{Z}[\sqrt{-p}]$, and the discriminant of this order is $\mathrm{Disc}(\mathcal{O}) = -4p$. Additionally, the class number $\#\mathrm{Cl}(\mathcal{O})$ can be estimated as $|\mathrm{Disc}(\mathcal{O})|^{\frac{1}{2}}$. Consequently, the quantum security of the class-group-based key exchange protocols is bounded by $\mathrm{Disc}(\mathcal{O})$. If one can construct a variant of CSIDH such that $\mathrm{Disc}(\mathcal{O})$ is a power of p, then the size of the field characteristic can be reduced. Inspired by this, a sequence of works has investigated appropriately increasing the discriminant of the imaginary quadratic order to shorten the size of the parameters.

This can be accomplished by working with elliptic curves oriented by imaginary quadratic orders, which are not generated by the Frobenius endomorphism π. The schemes SCALLOP [FFK+23], SCALLOP-HD [CLP24], together with PEARL-SCALLOP [ABE+24] operate with supersingular curves over the extension field $\mathbb{F}_{p^2}$, possessing the orientations by the imaginary quadratic orders of discriminants $\mathrm{Disc}(\mathcal{O}) \leq 4p^2$. A more recent work [CIL+26] improved this bound by combining the orientations of CSIDH and SCALLOP, and considering an orientation by $\mathcal{O} = \mathbb{Z}[\sqrt{-dp}]$, $d \leq p^2$, leading to a discriminant approximately of size p^3.

Houben [Hou26] proposed an effective method to generate and represent orientations with respect to arbitrarily large discriminants, and constructed a non-interactive constant-time key exchange protocol, called CSIDH-LDO in this paper, that adopts the class group action using such orientations. The quadratic order of CSIDH-LDO is $\mathbb{Z}[\sigma]$, where σ is an endomorphism of E with degree $M^r, r \in \mathbb{Z}$, and M is a large factor of $p+1$. Thus σ can be split into r isogenies $\sigma = \varphi_r \circ \cdots \circ \varphi_1$, resulting in r oriented elliptic curves $E_1, \cdots, E_r$. Since the discriminant $4N(\sigma) - \mathrm{Tr}(\sigma)^2$ of $\mathbb{Z}[\sigma]$ is extremely large, the size of p can be reduced to around 255 bits, thereby crucially enhancing the efficiency of the field arithmetic. Therefore, CSIDH-LDO exhibits considerable potential within isogeny-based cryptography. However, the cost of CSIDH-LDO is prohibitively high due to the restricted effective class group action, as it is required to perform r large M-isogenies respectively on the r oriented curves to keep track of the $\mathbb{Z}[\sigma]$-orientation. Moreover, the public-key size of CSIDH-LDO is too large to transmit for many real-world applications. Consequently, there is a strong command for an optimized implementation and a compact public-key compression for this key exchange protocol. To the best of our knowledge, there is no prior work that systematically investigates batching these r group actions in parallel using SIMD instructions or develops concrete approaches for the public-key compression. Motivated by this gap, we aim to develop a high-performance **parallel** implementation for the r time-consuming group operations and to establish a framework for efficient and compact public-key compression to CSIDH-LDO.

1.1 Our Contributions

Our contributions are summarized as follows.

1.1.1 Optimized Implementations

In this work, we present highly optimized, constant-time implementations of the isogeny-based key exchange protocol proposed by Houben [Hou26]. We evaluate two parameter configurations achieving NIST level 1 security: a conservative choice ($r = 13$) and an aggressive choice ($r = 7$). By bridging algorithmic optimizations with a tailored Single Instruction, Multiple Data (SIMD) framework, we demonstrate the high parallel efficiency and practical viability of the CSIDH-LDO protocol. Our software is available at `https://github.com/zyuh6/CSIDH-LDO`. Specific contributions for our efficient implementation are summarized as follows.

• **Algorithmic Improvements.** We accelerate scalar multiplication by combining small prime decomposition with differential addition chains [Mon92, CVCCD+19, BBC+21, BCL25], outperforming the standard Montgomery ladder by approximately 30%. Additionally, we transition the underlying curve arithmetic to the twisted Edwards model to leverage the more efficient Vélu's and $\sqrt{}$élu formulas [MR18].

• **SIMD Parallelization and Dummy-Free Scheduling.** We design parallelization strategies explicitly optimized for SIMD architectures. To address the hardware misalignment when executing an odd number of group actions ($r \in \{7, 13\}$) on 8-lane vector registers, we propose and compare two paradigms: a standard *external parallelism* approach (which inevitably wastes SIMD lanes on dummy operations but utilizes high-efficiency $\mathbb{F}_{p^2}$-arithmetic) and our highly optimized *dummy-free internal parallelism* approach. The latter dynamically mixes various parallelization widths (e.g., 8-, 4-, 2-, and 1-way) and descends into the arithmetic hierarchy to keep the 8 vector lanes fully saturated with zero redundant computation. The practical security and implementation details of our scheme, including constant-time execution and memory security, are detailed in Section 5.1.

• **Optimized Instantiations via Limb-Slicing.** We apply the *limb-slicing* technique [CGT+21, CFG+21, ZHZ+24] to our SIMD framework. We provide a constant-time C/x64 baseline, an AVX-512F version utilizing a redundant radix-2^{29} representation, and an AVX-512IFMA version leveraging `vpmadd52` instructions with a radix-2^{52} representation.

• **Performance Evaluation and SOTA Comparison.** We benchmark our protocol against state-of-the-art schemes (see Table 7). In the sequential x64 setting, CSIDH-LDO outperforms dCSIDH by factors of 1.29× for the 2048-bit discriminant and 3.63× for the 4096-bit discriminant. For $r = 7$, we also compare against dCTIDH-2048-194, which evaluates approximately 1.13× faster than our sequential CSIDH-LDO. More importantly, our protocol demonstrates a significantly higher aptitude for parallelization. Compared to our x64 baseline, our dummy-free AVX-512F and AVX-512IFMA implementations achieve substantial speedups of **3.40**× and **7.30**×, respectively. In contrast, prior attempts to parallelize other CSIDH-like protocols using IFMA (e.g., [CFG+21]) yielded limited performance gains of up to 1.71× over their baselines. This highlights the structural advantages of our approach in modern parallel computing environments.

1.1.2 Public-Key Compression

For the public-key compression of CSIDH-LDO, our main contributions are as follows.

• **An Algorithmic Framework.** In both cases, with and without the twist trick, we provide a concrete framework for public-key compression, allowing for the compressed public-key sizes of around $(2r + 2)\log_2(p)$ and $(4r + 2)\log_2(p)$, respectively. A security analysis is also addressed in Section 5.2 for our compressed CSIDH-LDO.

• **Effective Approaches for a Compact Compression.** We proposed two effective methods for a more compact key compression by further reducing the size of an M-torsion point $P \in E[M, \sigma]$ ($P_s \in E[M_s, \sigma]$ and $P_t \in E[M_t, \sigma]$ when leveraging the twist trick).

- Method I generalizes the techniques in [LLC+24] to the CSIDH-LDO setting. For the generation of the torsion basis, we extend the algorithms in [LLC+24] to construct a basis U, V such that $\langle U, V \rangle = E[M]$, where $M = \prod_{i=1}^{n} \ell_i^{e_i}$ is the product of prime powers. We further address the issue that both the reduced Tate pairings $e_{T,M}(P, U)$ and $e_{T,M}(P, V)$ are not generators of the group of primitive M-th roots of unity μ_M via the Chinese Remainder Theorem, with a small additional n-bit label to transmit. The public-key size can be reduced to $(r+2)\log_2(p) + rn$ bits in the case where the twist trick is not employed ($(2r+2)\log_2(p) + 2rn$ bits if leveraging the twist trick).

- Method II improves the generation of the basis U, V. By generating U to form a basis with P, and simultaneously accomplishing the pairing computation $e_{T,M}(P, U)$ incurring only a minor computational overhead using the properties of bilinear pairings, we significantly eliminate computational redundancy and further simplify the procedure for solving discrete logarithms. This approach also facilitates the decompression of $Q \in E[M, \widehat{\sigma}]$ and avoids the transmission of the extra labels required in Method I, resulting in a smaller public-key size of around $(r+2)\log_2(p)$ in the situation where the twist trick is not leveraged ($(2r+2)\log_2(p)$ bits when using the twist trick).

• **Performance of the Compression.** We make optimized implementations of public-key compression for CSIDH-LDO at NIST level 1 using Method II, across x64, AVX-512F, and IFMA platforms. Our experimental results demonstrate that the efficient compression procedure implemented by x64, AVX-512F, and IFMA account respectively for only 2.65%, 5.68%, and 9.07% of the total overhead of compressed CSIDH-LDO at the "conservative" NIST level 1, but achieve a crucial **93.1%** reduction in terms of public-key size.

1.2 Related Work

Building upon the work of Cheng *et al.* [CFGR22, CFG+21], this paper also leverages the AVX-512 instruction set to parallelize implementations of isogeny-based key exchange protocols. Likewise, the works [LLWZ23, LLC+24] have also contributed to the key compression of isogeny-based key exchange protocols, as the present paper does. Collectively, all these works demonstrate the significant potential of the AVX-512 instruction set for accelerating isogeny-based cryptographic schemes and benefit to the field of public-key compression.

1.3 Outline of This Paper

The preliminaries are presented in Section 2, involving the key exchange protocol from orientations with large discriminants, the AVX-512 architecture, and the biextensions. A detailed explanation of batching and evaluating the group action in parallel is provided in Section 3. Section 4 illustrates a concrete framework for the public-key compression, and introduces two effective methods for a more compact compression. Section 5 addresses our security considerations. The experimental result is summarized in Section 6. Finally, our conclusion is drawn in Section 7.

2 Preliminaries

In this section, we introduce the preliminaries required in this paper, including the key exchange protocol from orientations of large discriminants, the AVX-512 architecture, together with the definitions of biextensions and cubical arithmetic.

2.1 Key Exchange Protocols from Orientations of Large Discriminants

In this subsection, we introduce CSIDH-LDO, which is proposed by Houben in [Hou26]. Denote by $i : (x, y) \to (-x, \sqrt{-1} \cdot y)$ and $\pi : (x, y) \to (x^p, y^p)$ the automorphism and the p-power Frobenius endomorphism of the initial supersingular curve $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$, where $p \equiv 3 \bmod 4$, and $\mathbb{F}_{p^2} \cong \mathbb{F}_p[i]/\langle i^2 + 1\rangle$ is a finite field.

Starting with E_0, the group action of this protocol is dealt with a maximal order $\mathcal{O} = \mathbb{Z}[\sigma]$ with discriminant $4N(\sigma) - \mathrm{Tr}(\sigma)^2$, where $\sigma = a + bi + c\frac{i+\pi}{2} + d\frac{1+i\pi}{2},\ a, b, c, d \in \mathbb{Z}$ is an endomorphism of E_0. Besides, it satisfies that $N(\sigma) = M^r,\ M = \prod_{i=1}^{n} \ell_i^{e_i}$ and $M \mid p + 1$, where $\ell_i,\ i = 1, \cdots, n$ are some prime factors. Therefore, the endomorphism σ splits into a chain of r M-isogenies $\varphi_j,\ j = 1, \cdots, r$

$$E_1 \xrightarrow{\varphi_1} E_2 \xrightarrow{\varphi_2} \ldots \xrightarrow{\varphi_{r-1}} E_r \xrightarrow{\varphi_r} E_{r+1} \cong E_1.$$

We call E_0 an $\mathcal{O}$-oriented curve. To perform class group actions in constant time, the full kernel representations are required. For each $E_j,\ j = 1, \cdots, r$, define $\langle P_j\rangle = \ker(\varphi_j) = E_j[M, \sigma],\ \langle Q_j\rangle = \ker(\widehat{\varphi_{j-1}}) = E_j[M, \widehat{\sigma}]$, then $(E_j, P_j, Q_j),\ j = 1, \cdots, r$ is a full kernel representation. We now describe how to evaluate a group action. Let $\mathfrak{a} = \prod_{i=1}^{n} \mathfrak{l}_i^{s_i},\ s_i \le e_i$ denote an $\mathcal{O}$-ideal, where $\mathfrak{l}_i = (\ell_i, \sigma),\ i = 1, \cdots, n$ is the prime ideal split from $\ell_i,\ i = 1, \cdots, n$ in $\mathcal{O}$, respectively. For any $1 \le j \le r$, we define the walk-right and walk-left isogenies as follows:

$$E_j \xrightarrow{\varphi_j^+} E_j' \xleftarrow{\varphi_j^-} E_{j+1},$$

where $\ker(\varphi_j^+) = \langle [M/(\prod_{i=1}^{n} \ell_i^{s_i})]P_j\rangle$, $\ker(\varphi_j^-) = \langle [\prod_{i=1}^{n} \ell_i^{s_i}]Q_{j+1}\rangle$. It is worth noting that φ_j^+ corresponds to the action $\mathfrak{a}$. Moreover, it follows from [Hou26, Proposition 3.1] that $(E_j', P_j', Q_j') = (E_j', \varphi_j^-(P_{j+1}), \varphi_j^+(Q_j)),\ j = 1, \cdots, r$ is a full kernel representation for the orientation on E_1' induced by the isogeny $\varphi_1^+ : E_1 \to E_1'$. Thus, we can keep track of the orientation to ensure that the group action commutes. Furthermore, for each $1 \le j \le r$ it yields that $\varphi_j^+ \circ \widehat{\varphi_j^-} = \varphi_j \sim \prod_{i=1}^{n} \mathfrak{l}_i^{e_i}$. As a result, every group action only relies on $e_i,\ i = 1, \ldots, n$, which is fixed. Hence, it can be evaluated in constant time. The following algorithm [Hou26, Algorithm 1] illustrates the computational process for the group action $[\prod_{i=1}^{n} \mathfrak{l}_i^{s_i}] * E$ using the full kernel representation. The fundamental sub-algorithms $\texttt{Multiply}(E, P, N)$ and $\texttt{Isogeny}(E, \mathcal{L}, K, N)$ respectively compute the scalar multiplication $[N]P$ and the N-isogeny $\varphi(Q)$ for $Q \in \mathcal{L}$ with kernel $\langle K\rangle$.

Once we have Algorithm 1, the key exchange protocol can be designed as follows.

(1) Fixing the initial curve $E_0 : y^2 = x^3 + x$, along with the full kernel representation $\mathcal{E} := \{(E_j, P_j, Q_j) \mid 1 \le j \le r\}$ of a $\mathbb{Z}[\sigma]$-orientation of E_0 as public parameters.

(2) Alice chooses her secret key $\mathsf{sk}_{\mathsf{Alice}} = (a_1, \cdots, a_n) \in \mathbb{Z}_{\ge 0}^n$, where $0 \le a_i \le Be_i,\ B \in \mathbb{Z}_{>0}$. Then she computes the class group action $\mathcal{E}_{\mathrm{Alice}} = \prod_{i=1}^{n} \mathfrak{l}_i^{a_i} * \mathcal{E}$ executing Algorithm 1 for B times, and sends $\mathcal{E}_{\mathrm{Alice}}$ to Bob.

(3) Bob chooses his secret key $\mathsf{sk}_{\mathsf{Bob}} = (b_1, \cdots, b_n) \in \mathbb{Z}_{\ge 0}^n$, where $0 \le b_i \le Be_i,\ B \in \mathbb{Z}_{>0}$. Then he computes the class group action $\mathcal{E}_{\mathrm{Bob}} = \prod_{i=1}^{n} \mathfrak{l}_i^{b_i} * \mathcal{E}$ using B applications of Algorithm 1, and sends $\mathcal{E}_{\mathrm{Bob}}$ to Alice.

(4) Alice computes $\mathcal{E}_{AB} = \prod_{i=1}^{n} \mathfrak{l}_i^{a_i} * \mathcal{E}_{\mathrm{Bob}}$, and Bob computes $\mathcal{E}_{BA} = \prod_{i=1}^{n} \mathfrak{l}_i^{b_i} * \mathcal{E}_{\mathrm{Alice}}$. The j-invariants of $E_{AB}^{(j)} \cong E_{BA}^{(j)},\ j = 1, \cdots, r$ are the shared key.

The Twist Trick. It follows from [Hou26, Section 6.2] that the twist trick can be exploited to optimize this protocol. When lacking torsion on the base curve $E/\mathbb{F}_{p^2}$, one can consider the torsion of the quadratic twist $E_t/\mathbb{F}_{p^2}$ of E. It holds that $\#E_t(\mathbb{F}_{p^2}) = (p-1)^2$.

Algorithm 1 [Hou26] Evaluating the class group action on full kernel representations

Require: A full kernel representation $\{(E_j, P_j, Q_j) \mid 1 \le j \le r\}$ of a $\mathbf{Z}[\sigma]$-orientation on E, together with a vector of integers $(s_1, \ldots, s_n) \in [0, e_i]^n$.
Ensure: A full kernel representation $\{(E'_j, P'_j, Q'_j)\}$ of a $\mathbf{Z}[\sigma]$-orientation on $E' = \prod_{i=1}^n [\mathfrak{l}_i^{s_i}] * E$.

for $j = 1, \ldots, r$ **do**
  $R_0 \leftarrow (E_j, P_j, Q_j)$;
  $R_1 \leftarrow (E_{j+1}, Q_{j+1}, P_{j+1})$;
  $m \leftarrow \prod_{i=1}^n \ell_i^{e_i}$;
  for $i = 1, \ldots, n$ **do**
    for $t = 1, \ldots, e_i$ **do**
      $b \leftarrow t > s_i$;
      `cswap`(R_0, R_1, b); ▷ Swap R_0 and R_1 if b is True
      $m \leftarrow m/\ell_i$; $K \leftarrow$ `Multiply`$(R_0[1], m)$;
      $R_0 \leftarrow$ `Isogeny`(R_0, K, ℓ_i);
      $R_1[1] \leftarrow$ `Multiply`$(R_1[1], \ell_i)$;
      `cswap`(R_0, R_1, b); ▷ Swap R_0 and R_1 if b is True
    end for
  end for
  assert $R_0[0] = R_1[0]$;
  $R_0[1] \leftarrow R_1[2]$;
  $R'_j \leftarrow R_0$;
end for
return $R'_1, \ldots, R'_r$;

Consequently, by selecting appropriate parameters $M_s = \prod_{i=1}^n \ell_i^{e_i}$ and $M_t = \prod_{i=1}^m {\ell'_i}^{e'_i}$ such that $M_s \mid p+1$, $M_t \mid p-1$, $\gcd(M_s, M_t) = 1$, one can construct an orientation of norm $N(\sigma) = (M_s M_t)^r$ and perform the group action using the torsion points with x-coordinate in $\mathbb{F}_{p^2}$ but y-coordinates in $\mathbb{F}_{p^4}$. This allows the parameter p to be smaller at the same security level. The full kernel representation becomes $(E_j, P_{sj}, Q_{sj}, P_{tj}, Q_{tj})$, $j = 1, \cdots, r$, where $P_{sj} \in E_j[M_s, \sigma], Q_{sj} \in E_j[M_s, \widehat{\sigma}], P_{tj} \in E_j[M_t, \sigma], Q_{tj} \in E_j[M_t, \widehat{\sigma}]$ and $\sigma = (\varphi_{tr} \circ \varphi_{sr}) \circ \cdots \circ (\varphi_{t1} \circ \varphi_{s1})$. Assume that the ideal with respect to the group action is $\mathfrak{a} * \mathfrak{b}$, where $\mathfrak{a} = \prod_{i=1}^n \mathfrak{l}_i^{s_i}$, $s_i \le e_i$, $\mathfrak{b} = \prod_{i=1}^n \mathfrak{l}_i'^{t_i}$, $t_i \le e'_i$. Correspondingly, the walk-right and walk-left isogenies are defined as follows:

$$E_j \xrightarrow{\varphi^+_{sj}} E_{j_{\text{left}}} \xrightarrow{\varphi^+_{tj}} E'_j \xleftarrow{\varphi^-_{tj}} E_{j_{\text{right}}} \xleftarrow{\varphi^-_{sj}} E_{j+1},$$

where

$$\ker(\varphi^+_{sj}) = \left\langle \left[M_s \Big/ \left(\prod_{i=1}^n \ell_i^{s_i} \right) \right] P_{sj} \right\rangle, \ \ker(\varphi^-_{sj}) = \left\langle \left[\prod_{i=1}^n \ell_i^{s_i} \right] Q_{sj+1} \right\rangle,$$
$$\ker(\varphi^+_{tj}) = \left\langle \left[M_t \Big/ \left(\prod_{i=1}^m \ell_i'^{t_i} \right) \right] \varphi^+_{sj}(P_{tj}) \right\rangle, \ \ker(\varphi^-_{tj}) = \left\langle \left[\prod_{i=1}^m \ell_i'^{t_i} \right] \varphi^-_{sj}(Q_{tj+1}) \right\rangle.$$

Similarly, the following tuple

$$\left(E'_j, \varphi^-_{tj} \circ \varphi^-_{sj}(P_{sj+1}), \varphi^+_{tj} \circ \varphi^+_{sj}(Q_{sj}), \varphi^-_{tj} \circ \varphi^-_{sj}(P_{tj+1}), \varphi^+_{tj} \circ \varphi^+_{sj}(Q_{tj})\right), \ j = 1, \cdots, r$$

is the full kernel representation of the orientation on E'_j induced by the isogeny $\varphi^+_{tj} \circ \varphi^+_{sj} : E_j \to E'_j$ associated to the action $\mathfrak{a} * \mathfrak{b}$.

2.2 SIMD Architecture

The *Single Instruction Multiple Data* (SIMD) instruction set architecture enables the parallel processing of packed data elements. This execution model significantly enhances computational throughput, particularly for compute intensive workloads like finite field arithmetic in cryptography [OAL18, CFGR22, NS22].

The protocol presented in this paper is highly suitable for SIMD parallelization because it consists of multiple identical and mutually independent computational modules. Rather than processing sequentially, our approach executes these independent modules concurrently across wide vector lanes.

To evaluate the parallel potential of our approach, we use the Intel AVX-512 architecture as a concrete example of a SIMD execution engine. Specifically, we instantiate our implementation using two distinct instruction sets: the AVX-512 Foundation (AVX-512F) for standard arithmetic operations, and the AVX-512 Integer Fused Multiply-Add (AVX-512IFMA) extension, which optimizes large integer multiplication using a redundant radix-2^{52} representation [CFG+21]. The specific microarchitectural details and the target platform used for our evaluation are deferred to Appendix A.1.

3 Evaluating the Class Group Action in Parallel

This section provides a detailed explanation of parallel evaluation strategies for the class group action of CSIDH-LDO. While the parallelization methodologies discussed herein offer general insights applicable to various SIMD architectures, we instantiate them with concrete and highly optimized implementations utilizing AVX-512F and AVX-512IFMA instructions. All reported CPU cycle counts in this section were measured on an 11th Gen Intel Core i7-1185G7 processor with Intel Turbo Boost and Hyper-Threading disabled.

As mentioned in Section 2.1, the evaluation of one group action consists of the following blocks: (1) Arithmetic on $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$; (2) (X, Z)-only scalar multiplication; (3) Isogeny construction and evaluation with Vélu's formula: kernel points computation (`KPS`), codomain curve construction (`xISOG`) and isogeny evaluation (`xEVAL`); (4) Isogenies φ_j^+ and φ_j^- between E_j and E_{j+1} for $j = 1, 2, \ldots, r$; (5) Correction for an isomorphism between $\varphi_r^+(E_r)$ and $\varphi_r^-(E_1)$.

The parameters of our NIST level 1 implementations, following the setup in [Hou26], are provided in Appendix A.3. We instantiate the protocol with two parameter settings that differ only in the choice of r. These include:

- p: the characteristic of the finite field,
- r: the number of oriented elliptic curves (we evaluate $r = 7$ and $r = 13$ for 2048-bit and 4096-bit discriminants, achieving "aggressive" and "conservative" NIST level 1, respectively),
- B: the number of class group action evaluations required to achieve the desired key space size,
- M_s: the product of the degrees of the isogenies constructed with kernels on the original curve,
- M_t: the product of the degrees of the isogenies constructed with kernels mapped from its quadratic twist.

At first glance, instantiating the protocol with an odd number of curves (e.g., $r = 7$ or $r = 13$) might appear suboptimal for AVX-512 architectures, as they typically reach peak

efficiency when workloads perfectly align with 8 or 16 vector lanes. A naive coarse-grained parallelization across the r independent actions would inevitably waste vector speedup by padding the execution with dummy operations to fill the remaining lanes.

To overcome this inherent structural misalignment, our implementations are completely *dummy-free*. Rather than relying solely on high-level parallelism across the r actions, we deeply exploit *internal parallelism* within the underlying field arithmetic and isogeny computations (e.g., Blocks 1–3). We explicitly implemented both the naive approach (with dummy operations) and our dummy-free approach. Our experimental results confirm that the proposed dummy-free strategy achieves higher performance, demonstrating the practicality of our parameter choices.

3.1 Finite Field Arithmetic

In this section, we present the selections of the radix and the arithmetic over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$.

3.1.1 Suitable Radix Selection

For AVX-512F, the radix of the element in $\mathbb{F}_p$ is limited to 2^{32}, while for IFMA, it does not exceed 2^{52}. These limits ensure that no data is lost when performing multiplication instructions between registers.

AVX-512F. For our 255-bit p, an element in $\mathbb{F}_p$ is divided into 9 limbs. Although $8 \times 64 = 256 > 255$, exactly 8 limbs are not feasible in practice since a radix of 2^{32} would cause the result of multiplying two registers to reach a magnitude of 64 bits, and subsequent additions could lead to data overflow. Therefore, we choose a radix-2^{29} representation since it is the smallest possible that can be used with 9 limbs. This allows us to perform efficient finite field multiplication and utilize redundant representations.

IFMA. For the same 255-bit p, an $\mathbb{F}_p$-element is divided into 5 limbs. Since 52 bits is much smaller than 64 bits, the integer multiplication remains unaffected.

3.1.2 Parallel Field Operations

In our implementation, we primarily adopt an (8×1)-way[1] field operation strategy, performing 8 operations in parallel, with each operation involving one 64-bit lane of the vector. The layout of the radix representation is shown in Figure 1. This configuration optimally utilizes the vector width and can be smoothly consumed at higher algorithmic levels.

To optimize modular arithmetic, we leverage the redundant representations (radix-2^{29} and radix-2^{52}) to omit the final subtraction in Montgomery reduction [Mon85]. By ensuring $4p < R$, we use $2p$ instead of p as the modulus for addition and subtraction, keeping all $\mathbb{F}_p$-elements in the range $[0, 2p)$.

For multiplications and squarings in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, we tailor the scanning strategies and the application of Karatsuba's algorithm to the specific limb counts. Standard arithmetic operations and detailed bounds analysis are deferred to Appendix A.2. Tables 1 and 2 present the latency of our finite field implementations under different parallelization strategies.

[1] In our notation, a $(y \times z)$-way $\mathbb{F}_p$ implementation performs y operations in parallel, each utilizing z vector lanes. For $\mathbb{F}_{p^2}$, an $(x \times y \times z)$-way implementation executes x $\mathbb{F}_{p^2}$-operations in parallel, each performing y concurrent $\mathbb{F}_p$-operations across z lanes.

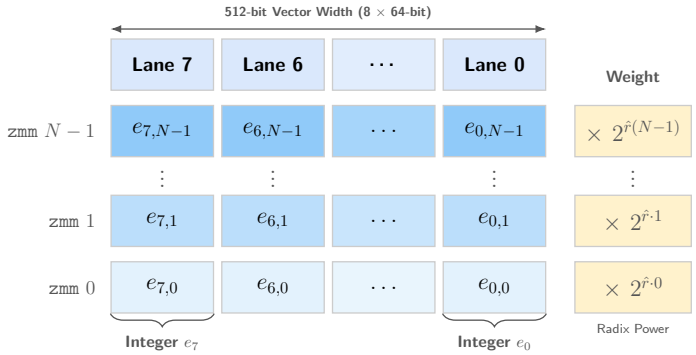


Figure 1: Layout of $e_i = \sum_{k=0}^{N-1} e_{i,k} \cdot 2^{\hat{r} \cdot k}$ in $\mathbb{F}_p$ for 8×1-way $\mathbb{F}_p$-operations, $i \in \{0, 1, \ldots, 7\}$.

Table 1: Performance comparison of $\mathbb{F}_p$ multiplication and squaring implementations.

ISA	Int. Mul/Sqr	Reduction	Struct.	#Inst.	Multiplication		Squaring	
					Lat.†	Sp.	Lat.†	Sp.
x64	Schoolbook	Schoolbook	Separated	1	73.0	1.00×	73.0	1.00×
AVX-512F	Karatsuba	Prod-Scan.	Separated	8	251.0	2.33×	-	-
	Prod-Scan.	Prod-Scan.	Interleaved	8	233.0	2.51×	204.0	2.86×
IFMA	Prod-Scan.	Op-Scan.	Interleaved	8	114.0	5.12×	95.0	6.15×

† The latency, reported in CPU cycles, is the median of 100 data points, where each point is the average cycle count over 10,000,000 executions.

Table 2: Performance comparison of $\mathbb{F}_{p^2}$ multiplication and squaring implementations.

ISA	Implementation	#Instances	Multiplication		Squaring	
			Latency†	Speedup	Latency†	Speedup
x64	1-way	1	324.0	1.00×	204.0	1.00×
AVX-512F	$(2 \times 4 \times 1)$-way	2	353.0	1.84×	353.0	1.16×
	$(4 \times 2 \times 1)$-way	4	426.5	3.04×	302.0	2.70×
	$(8 \times 1 \times 1)$-way	8	739.5	3.51×	524.0	3.11×
IFMA	$(2 \times 4 \times 1)$-way	2	180.0	3.60×	180.0	2.27×
	$(4 \times 2 \times 1)$-way	4	196.0	6.61×	136.0	6.00×
	$(8 \times 1 \times 1)$-way	8	321.0	8.07×	261.0	6.25×

† The latency, reported in CPU cycles, is the median of 100 data points, where each point is the average cycle count over 10,000,000 executions.

3.2 Scalar Multiplication

This section focuses on the optimization of the scalar multiplications $\texttt{Multiply}(R_0[1], m)$ and $\texttt{Multiply}(R_1[1], \ell_i)$ in Algorithm 1, where m is the product of ℓ_i.

Differential Addition Chain. For efficient constant-time execution, we precompute a differential addition chain (see [Mon92, CVCCD+19, BBC+21, CCJR22, CCSCD+24, BCL25]) for ℓ_i and store it for use in the relevant scalar multiplications. This optimization benefits both $\texttt{Multiply}(R_1[1], \ell_i)$ and $\texttt{Multiply}(R_0[1], m)$, since m defined as the product of ℓ_i. Compared to the Montgomery ladder, the differential addition chains method achieves an average optimization of approximately 30% for the scalar multiplication by ℓ_i, as shown in Figure 2.

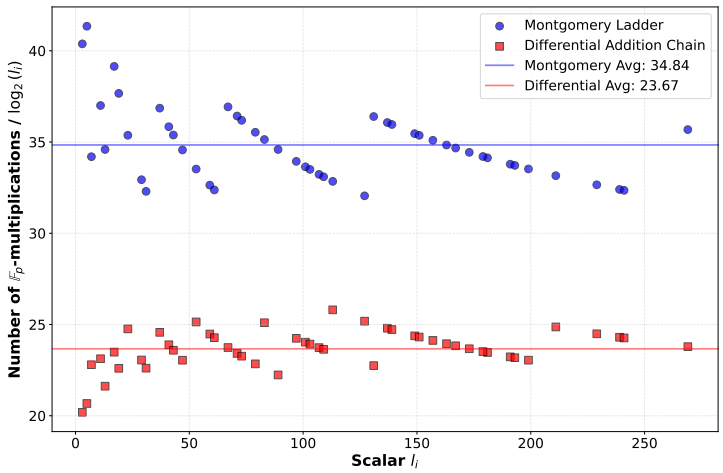


Figure 2: Cost of Scalar Multiplication by ℓ_i: Differential Addition Chain vs. Montgomery Ladder. Here, one $\mathbb{F}_{p^2}$-multiplication (resp. squaring) is measured as 3 (resp. 2) $\mathbb{F}_p$-multiplications.

2-Way Parallelization at $\mathbb{F}_{p^2}$-Level. The x-only differential addition chain for the scalar multiplication $\texttt{Multiply}(E, P, N)$ on Montgomery curves consists of $\texttt{xDBL}(E, P)$ and $\texttt{xADD}(E, P, Q, P - Q)$. These two algorithms with 2-way parallelization, as described in Algorithms 13 and 14, facilitate the construction of $\texttt{Multiply}(E, P, N)$.

3.3 Isogeny Construction and Evaluation

The algorithm $\texttt{Isogeny}(E, \mathcal{L}, K, N)$ contained in Algorithm 1 consists of three parts: (1) `KPS` for computing the kernel points, (2) `xISOG` for constructing the codomain curve, (3) `xEVAL` for evaluating the isogeny on points.

By employing the twist trick described in [Hou26, Section 2.1], the full kernel representation of the orientation can be written as $\{(E_j, P_{sj}, Q_{sj}, P_{tj}, Q_{tj}) \mid j = 1, \cdots, r\}$. Here, P_{sj} and Q_{sj} are points on the curve E_j of order M_s, while P_{tj} and Q_{tj} are points mapped from the quadratic twist E_j^t of order M_t. Each `Isogeny` involves one `KPS`, one `xISOG`, and four `xEVAL`s evaluating the isogeny on $P_{sj}, Q_{sj}, P_{tj}, Q_{tj}$, respectively.

3.3.1 Vélu's Formula and √élu

Vélu's formula (`Isogeny_tra`) and √élu (`Isogeny_srt`) are two well-known methods for isogeny construction and evaluation. When computing an isogeny of a large degree ℓ, √élu is theoretically more efficient, offering a complexity of $\tilde{\mathcal{O}}(\sqrt{\ell})$ compared to the $\mathcal{O}(\ell)$ complexity of Vélu's formula.

We profile the cost of both methods for varying isogeny degrees on twisted Edwards curves. We assume that one $\mathbb{F}_{p^2}$-multiplication (resp. squaring) costs 3 (resp. 2) $\mathbb{F}_p$-multiplications. Because applying the twist trick means a single `Isogeny` computation requires 1 `KPS`, 1 `xISOG`, and 4 `xEVAL`s (rather than just 1 `xEVAL`), the efficiency crossover threshold shifts notably higher than in common scenarios. As illustrated in Figure 3, the theoretical $\mathbb{F}_p$-multiplication count of `Isogeny_srt` only drops below that of `Isogeny_tra` when $\ell > 211$. At the exact crossover point $\ell = 211$, `Isogeny_tra` requires $7,414$ multiplications, which is strictly less than the $7,422$ required by `Isogeny_srt`.

In our specific implementation, only four small primes ($229, 239, 241$, and 269) exceed this threshold. Even for these, the theoretical reduction in multiplications is marginal: for example, at $\ell = 229$, `Isogeny_srt` saves merely 355 multiplications ($7,689$ vs. $8,044$).

In our experiments, this minor theoretical arithmetic advantage is entirely offset by the algorithmic overhead of `Isogeny_srt`. Consequently, measured in actual CPU cycles, `Isogeny_tra` consistently outperforms `Isogeny_srt` across all tested degrees regardless of the target ISA.

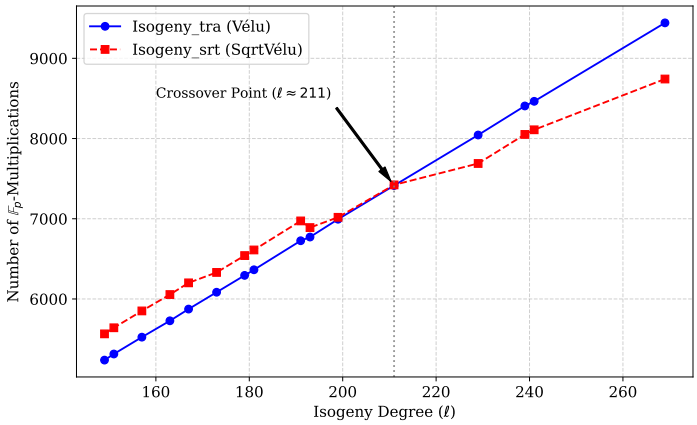


Figure 3: Comparison of theoretical $\mathbb{F}_p$-multiplication costs between Vélu's formula (`Isogeny_tra`) and $\sqrt{}$élu (`Isogeny_srt`) for a single isogeny step (1`KPS`+1`xISOG`+4`xEVAL`s). The cost model assumes 1 $\mathbb{F}_{p^2}$-multiplication (resp. squaring) costs 3 (resp. 2) $\mathbb{F}_p$-multiplications.

3.3.2 Parallelization for `Isogeny`

Instead of parallelizing the `Isogeny` function internally (i.e., at $\mathbb{F}_{p^2}$-level), another approach is to use external parallelization by executing multiple `Isogeny` computations concurrently. This higher-level parallelization avoids the need to manage parallelism at $\mathbb{F}_{p^2}$-level.

Regarding internal parallelization of the `Isogeny` function, given the limited performance gain, we do not pursue internal parallelization of the `Isogeny_srt` function. Considering `Isogeny_tra`, we discuss 2-way and 4-way parallelism. A natural idea is that, since four `xEVAL_tra` calls are required, we can parallelize them externally, i.e., compute 2 or 4 image points simultaneously. For `KPS_tra`, `xISOG_tra`, and the associated `Multiply`, we employ parallelization at the $\mathbb{F}_{p^2}$-level.

2-Way Parallelization. We have two strategies for achieving 2-way parallelism. The first is to externally pair the `xEVAL_tra` calls, denoted as $(2 \times 1 \times 1 \times 1)$-way, and use internal $\mathbb{F}_{p^2}$-level parallelism for other functions (Algorithms 17 and 18). The second is to apply internal $\mathbb{F}_{p^2}$-level parallelism directly within `xEVAL_tra` (Algorithm 19), denoted as $(1 \times 2 \times 1 \times 1)$-way.

In practice, the internal $(1 \times 2 \times 1 \times 1)$-way approach is more efficient. To maximize vector register utilization during benchmarking, we process four independent curves concurrently, extending the implementations to $(4 \times 1 \times 2 \times 1 \times 1)$-way and $(4 \times 2 \times 1 \times 1 \times 1)$-way. Furthermore, the internal approach successfully pairs almost all $\mathbb{F}_{p^2}$-operations and eliminates the data rearrangement overhead caused by storage format mismatches between the preceding $(1 \times 2 \times 1 \times 1)$-way `KPS_tra` and the external $(2 \times 1 \times 1 \times 1)$-way `xEVAL_tra`.

4-Way and 8-Way Parallelization. We can scale up by combining both levels of parallelization. For 4-way parallelism `Isogeny_tra`, we externally process four `xEVAL_tra`

Table 3: Performance comparison of $(4 \times 2 \times 1 \times 1 \times 1)$-way and $(4 \times 1 \times 2 \times 1 \times 1)$-way `xEVAL_tra`, including data rearrangement overhead after `KPS_tra`.

Implementation	#Inst.	AVX-512F		IFMA	
		Latency†	Speedup	Latency†	Speedup
$(4 \times 1 \times 2 \times 1 \times 1)$-way	4	187,089.0	1.00×	84,617.0	1.00×
$(4 \times 2 \times 1 \times 1 \times 1)$-way	8	386,370.0	0.97×	174,001.0	0.97×

† The latency, reported in CPU clock cycles, is the median of 50 data points, where each point is the average cycle count over 10,000 executions.

calls while handling the remaining functions using 2-way $\mathbb{F}_{p^2}$-level and 2-way $\mathbb{F}_p$-level parallelization.

In our full implementation, the `Isogeny_tra` computation requires 8-way parallelism. For functions other than `xEVAL_tra`, we employ $(1{\times}2{\times}4{\times}1)$-way parallelization (comprising 1-way functional parallelism and $(2 \times 4 \times 1)$-way $\mathbb{F}_{p^2}$-operations). For `xEVAL_tra`, we use $(4 \times 2 \times 1 \times 1)$-way parallelization, evaluating 4 points simultaneously to maximize the efficiency of the $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-operations. While evaluating only a single point externally would prevent data rearrangement, processing fewer than 4 points forces the use of a significantly less efficient $\mathbb{F}_{p^2}$ parallelization strategy, making the rearrangement tradeoff entirely worthwhile.

3.4 Class Group Action

Denote by `action_home_base` the procedure that computes $(E'_j, P'_{sj}, Q'_{sj}, P'_{tj}, Q'_{tj})$ from $(E_j, P_{sj}, Q_{sj}, P_{tj}, Q_{tj})$ and $(E_{j+1}, P_{sj+1}, Q_{sj+1}, P_{tj+1}, Q_{tj+1})$. A single class group action (noted as `group_action`) consists of r such `action_home_base` instances (corresponding to r pairs φ_j^+ and φ_j^- for $j = 1, 2, \ldots, r$), followed by one correction for an isomorphism between $\varphi_r^+(E_r)$ and $\varphi_r^-(E_1)$. Noting that r `action_home_base` operations are mutually independent, we can compute them in parallel.

3.4.1 Parallelization Strategies at a Higher Level

In our implementation, the maximum capacity of a single SIMD vector is 8 concurrent instances of `action_home_base` (established in Section 2.2). This hardware constraint creates a structural misalignment when evaluating $r = 13$ or $r = 7$ instances. Specifically, computing $r = 13$ instances requires multiple execution passes, while computing $r = 7$ instances in a single 8-way pass inherently leaves one lane underutilized. To address this mismatch, we evaluate two parallelization strategies

- **Strategy A (Dummy-based external parallelism):** Rely exclusively on the maximum 8-way external parallelism. For $r = 13$, this requires an $(8+8)$-way parallelism, that is, executing two 8-way parallel implementations at `action_home_base`-level sequentially; for $r = 7$, it requires a single 8-way execution. Consequently, 3 lanes (for $r = 13$) or 1 lane (for $r = 7$) must perform redundant dummy computations to fill the 512-bit registers, inevitably wasting SIMD potential.
- **Strategy B (Dummy-free internal parallelism):** Mix different parallelization widths to precisely match r without any redundant operations. For $r = 13$, we execute an $(8 + 4 + 1)$-way parallelism, that is, executing one 8-way, one 4-way, and one 1-way parallel implementation sequentially at `action_home_base`-level; for $r = 7$, we execute a $(4 + 2 + 1)$-way parallelism in a similar sequential manner.

Strategy A is straightforward and avoids data rearrangement, as all internal operations remain 1-way. However, the unavoidable dummy operations fundamentally limit its

efficiency. In contrast, Strategy B eliminates dummy computations entirely. To achieve this, Strategy B deeply exploits *internal parallelism* at `xEVAL_tra`-level or $\mathbb{F}_{p^2}$-level to keep the 512-bit registers fully occupied. Although mixing different parallel widths introduces overhead from data rearrangement, our experiments show that the zero-waste execution in Strategy B yields superior overall efficiency.

To formalize our internal parallelism designs, we define a $(v \times w \times x \times y \times z)$-way parallel `group_action` implementation as follows:

- v: number of parallel `group_action` instances.
- w: number of parallel functions (e.g., `KPS`, `xISOG`, `xEVAL` or `Multiply`) per instance.
- x: number of parallel $\mathbb{F}_{p^2}$-operations per function.
- y: number of parallel $\mathbb{F}_p$-operations per $\mathbb{F}_{p^2}$-operation.
- z: number of 64-bit lanes occupied per $\mathbb{F}_p$-operation within the vector.

For both AVX-512F and IFMA, the product $v \cdot w \cdot x \cdot y \cdot z$ must strictly equal 8. We detail the implementations for the various v-way components below.

8-Way Parallelization. Used in both strategies, this is efficiently implemented via an $(8 \times 1 \times 1 \times 1 \times 1)$-way configuration. It avoids data rearrangement and achieves optimal parallel efficiency per instance.

4-Way Parallelization. With $v = 4$, and knowing $z = 1$ (due to our (8×1)-way $\mathbb{F}_p$-arithmetic) and $w = 1$ (due to data dependencies in most functions), we must set either x or y to 2. For `xEVAL_tra`, setting $w = 2$ is theoretically possible but less efficient than parallelizing at the $\mathbb{F}_{p^2}$-level (Section 3.3.2). Thus, we compare two primary configurations:

(1) **$(4 \times 1 \times 2 \times 1 \times 1)$-way**: Applies 2-way parallelization at the $\mathbb{F}_{p^2}$-level. This offers the most efficient $\mathbb{F}_{p^2}$-arithmetic but requires data shuffling.

(2) **$(4 \times 1 \times 1 \times 2 \times 1)$-way**: Employs a $(4 \times 2 \times 1)$-way approach for $\mathbb{F}_{p^2}$-operations. It avoids data rearrangement but has slightly slower arithmetic throughput.

As shown in Table 4, the $(4 \times 1 \times 2 \times 1 \times 1)$-way parallelism is more efficient.

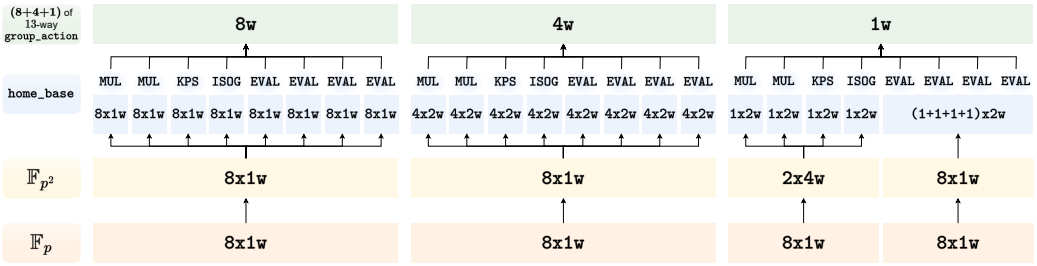
Table 4: Performance comparison of $(4 \times 1 \times 2 \times 1 \times 1)$-way and $(4 \times 1 \times 1 \times 2 \times 1)$-way `home_base` implementations.

ISA	Implementation	Latency†	#Instances	Speedup
AVX-512F	$(4 \times 1 \times 2 \times 1 \times 1)$-way	37.10×10^6	4	1.00×
	$(4 \times 1 \times 1 \times 2 \times 1)$-way	41.53×10^6	4	0.89×
IFMA	$(4 \times 1 \times 2 \times 1 \times 1)$-way	17.18×10^6	4	1.00×
	$(4 \times 1 \times 1 \times 2 \times 1)$-way	19.90×10^6	4	0.86×

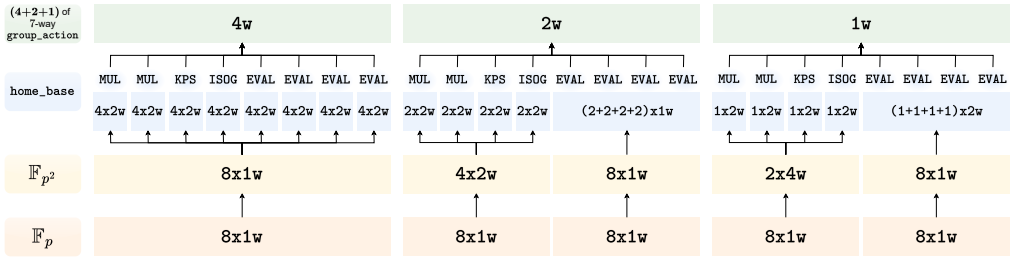
† The latency, reported in CPU cycles, is the median of 50 data points, where each point is the average cycle count over 100 executions.

2-Way Parallelization. For the 2-way component in $r = 7$, we must set $v = 2$. Following a similar analysis of the arithmetic hierarchy, we adopt two distinct configurations depending on the function to fully pack the SIMD lanes. For `xEVAL_tra`, we use a $(2 \times 4 \times 1 \times 1 \times 1)$-way configuration to process the 4 points simultaneously. For the remaining functions, we adopt a $(2 \times 1 \times 2 \times 2 \times 1)$-way configuration, which balances the throughput of field arithmetic with the necessary data rearrangement costs.

1-Way Parallelization. For $v = 1$, the optimal configuration for most functions is $(1 \times 1 \times 2 \times 4 \times 1)$. For `xEVAL_tra`, we adopt the $(1 \times 4 \times 2 \times 1 \times 1)$ configuration. This allows us to use the efficient $(8 \times 1 \times 1)$-way $\mathbb{F}_{p^2}$-arithmetic, although it introduces additional data rearrangement overhead.



(i) Strategy B ($r = 13$): $(8 + 4 + 1)$-way parallel implementation.



(ii) Strategy B ($r = 7$): $(4 + 2 + 1)$-way parallel implementation.

Figure 4: An overview of the structure of `group_action` implementation for Strategy B. Top: $r = 13$, $(8 + 4 + 1)$-way parallel. Bottom: $r = 7$, $(4 + 2 + 1)$-way parallel. In both configurations, the two `MUL`s represent scalar multiplications by ℓ_i and m, while the four `EVAL`s represent isogeny evaluations for P_{sj}, Q_{sj}, P_{tj} and Q_{tj}.

3.4.2 Correction for an Isomorphism

After computing the `action_home_base` between E_r and E_1, a correction via an isomorphism must be applied to the resulting points. This is necessary because $\varphi_r^+(E_r)$ and $\varphi_r^-(E_1)$ are isomorphic, but not identical. In contrast, when using Vélu (or $\sqrt{}$élu) formula for the isogeny computations, we have $\varphi_r^+(E_j) = \varphi_r^-(E_{j+1})$ for all $1 \le j \le r-1$.

For clarity, let us denote $\varphi_r^+(E_r)$ by $E_A : y^2 = x^3 + Ax^2 + x$ and $\varphi_r^-(E_1)$ by $E_{A'} : y^2 = x^3 + A'x^2 + x$. The isomorphism between E_A and $E_{A'}$ is then given by

$$\psi \colon E_A \to E_{A'}, \quad (x, y) \mapsto \left(\frac{x - \beta}{\alpha^2}, \frac{y - s(x - \beta) - t}{\alpha^3} \right),$$

where $\alpha^2 = \sqrt{3\beta^2 + 2A\beta + 1}$ and $\beta \neq 0$ is chosen such that $(\beta, 0)$ is a 2-torsion point on E_A. Additionally, $\frac{A+3\beta}{\alpha^2} = A'$ should hold. The kernel representation (with the twist trick considered) of E'_r is $\left(\varphi_r^+(E_r),\ \psi^{-1} \circ \varphi_r^-(P_{s1}),\ \varphi_r^+(Q_{sr}),\ \psi^{-1} \circ \varphi_r^-(P_{t1}),\ \varphi_r^-(Q_{tr})\right)$.

Remark 1. In our implementation, to meet the constant-time requirement and reduce the number of square root computations, we swap the positions of A and A' when $A' = 0$ to ensure that $A' \neq 0$. This guarantees that the value of α^2 can always be computed using $(A + 3\beta)/A' = \alpha^2$. Additionally, there are two possible choices for β that satisfy the condition that $(\beta, 0)$ is a 2-torsion point. Therefore, we further screen the β that can correctly construct the isomorphism by verifying whether $\alpha^4 = 3\beta^2 + 2A\beta + 1$ holds.

It is worth noting that when $A = 0$ or $A' = 0$, β can be chosen as either i or $-i$ to construct the mapping. However, the mapping that ensures the adjusted P_{sr} and P_{tr} satisfy $\langle P_{sr}, Q_{sr}\rangle = E_r[M_{sr}]$ and $\langle P_{tr}, Q_{tr}\rangle = E_r[M_{tr}]$ must be constructed with $\beta = i$. This only requires that in the square root computation (which is a constant-time deterministic algorithm), the square root of -1 is always taken as i.

We implement a 4-way parallelization at $\mathbb{F}_{p^2}$-level for the correction (Algorithm 2).

Algorithm 2 Construct the isomorphism between $E_A : y^2 = x^3 + Ax^2 + x$ and $E_{A'} : y^2 = x^3 + A'x^2 + x$ via α and β defined above, with 4-way parallelization at $\mathbb{F}_{p^2}$ layer.

Require: Points $\varphi_r^-(P_{s1}) = (X_{Ps} : Z_{Ps})$ and $\varphi_r^-(P_{t1}) = (X_{Pt} : Z_{Pt})$ on $E_A/\langle\pm 1\rangle$, coefficients α and β.

Ensure: Points $\psi^{-1} \circ \varphi_r^-(P_{s1})$ and $\psi^{-1} \circ \varphi_r^-(P_{t1})$ on $E_{A'}/\langle\pm 1\rangle$.

$s_0 \leftarrow Z_{Ps} \times Z_{\alpha^2}$ $\quad t_0 \leftarrow Z_{Pt} \times Z_{\alpha^2}$ $\quad v_0 \leftarrow Z_{Ps} \times Z_\beta$ $\quad w_0 \leftarrow Z_{Pt} \times Z_\beta$

$s_2 \leftarrow X_{Ps} \times X_{\alpha^2}$ $\quad t_2 \leftarrow X_{Pt} \times X_{\alpha^2}$ $\quad Z_{Ps_1} \leftarrow v_0 \times X_{\alpha^2}$ $\quad Z_{Pt_1} \leftarrow w_0 \times X_{\alpha^2}$

$s_2 \leftarrow s_2 \times Z_\beta$ $\quad t_2 \leftarrow t_2 \times Z_\beta$ $\quad v_2 \leftarrow Z_{Ps} \times X_\beta$ $\quad w_2 \leftarrow Z_{Pt} \times X_\beta$

$s_1 \leftarrow s_0 \times X_\beta$ $\quad t_1 \leftarrow t_0 \times X_\beta$ $\quad v_1 \leftarrow X_{Ps} \times Z_\beta$ $\quad w_1 \leftarrow X_{Pt} \times Z_\beta$

$X_{Ps_0} \leftarrow s_1 + s_2$ $\quad X_{Pt_0} \leftarrow t_1 + t_2$ $\quad v_1 \leftarrow v_1 - v_2$ $\quad w_1 \leftarrow w_1 - w_2$

$X_{Ps_1} \leftarrow v_1 \times Z_{\alpha^2}$ $\quad X_{Pt_1} \leftarrow w_1 \times Z_{\alpha^2}$ $\quad Z_{Ps_0} \leftarrow s_0 \times Z_\beta$ $\quad Z_{Pt_0} \leftarrow t_0 \times Z_\beta$

$\texttt{CSWAP}(P_{s0}, P_{s1}, \text{Aprime_is_zero})$ ▷ Swap P_{s0} and P_{s1} if $A' = 0$

$\texttt{CSWAP}(P_{t0}, P_{t1}, \text{Aprime_is_zero})$ ▷ Swap P_{t0} and P_{t1} if $A' = 0$

return P_{s0}, P_{t0}

4 Public-Key Compression

In this section, we discuss the public-key compression for CSIDH-LDO. The compression framework is first presented. We then introduce the optimization details for the building blocks of our public-key compression procedure, including the torsion basis generation, the pairing and discrete logarithm computations. Moreover, we propose two effective approaches to a more compact and efficient compression.

4.1 The Framework of the Public-Key Compression

Let the notation be the same as Section 2.1. We first consider the standard case in which the twist trick is not utilized. Alice and Bob need to transmit their public keys $\mathcal{E}_A = (E_{A_j}, P_{A_j}, Q_{A_j})$ and $\mathcal{E}_B = (E_{B_j}, P_{B_j}, Q_{B_j})$, $j = 1, \cdots, r$, to each other after the key generation. We fix one party as an example, and define the public key to be (E_j, P_j, Q_j), $j = 1, \cdots, r$, to simplify the notation. As mentioned in [Hou26, Section 5.1], every Q_{j+1} can be recovered by P_j as follows.

(1) Find an M-torsion point R_1 such that $\langle P_1, R_1\rangle = E_1[M]$.

(2) Compute the image $\varphi_1(R_1)$ with $\ker(\varphi_1) = \langle P_1\rangle$. We can take $Q_2 = \varphi_1(R_1)$, which generates $\ker(\widehat{\varphi_1})$, and then successively obtain $Q_{j+1} = \varphi_j(Q_j)$, $j = 2, \cdots, r$. Note that Q_{r+1} and Q_1 differ by a correction of an isomorphism.

Additionally, once we obtain E_1, the remaining curves E_j, $j = 2, \cdots, r$ can be subsequently derived by evaluating the isogenies $\varphi_1, \cdots, \varphi_{r-1}$ with $\langle P_1\rangle, \cdots, \langle P_{r-1}\rangle$ as kernels, respectively. Therefore, it is sufficient for one party to send $\mathsf{pk} = (E_1, P_1, \cdots, P_r)$ to the other, and vice versa.

One can make further compression by encoding P_j via its coordinates $a_j, b_j \in \mathbb{Z}/M\mathbb{Z}$ of $P_j,\ j = 1, \cdots, r$, with respect to a basis $\langle U_j, V_j \rangle = E_j[M]$. This is equivalent to $P_j = [a_j]U_j + [b_j]V_j$. To derive a_j and $b_j,\ j = 1, \cdots, r$, one can solve a 2-dimensional elliptic curve discrete logarithm. However, it is relatively expensive as the arithmetic on elliptic curves is required. To overcome this bottleneck, we can make use of the bilinear pairings to reduce this problem to solving the discrete logarithm over a finite field. The procedure is presented as follows.

(1) Compute three Weil pairings $f_j = e_{W,M}(U_j, V_j)$ $g_j = e_{W,M}(P_j, U_j)$ and $h_j = e_{W,M}(P_j, V_j)$.

(2) Solve two discrete logarithms $a_j = \log_{f_j} h_j$ and $b_j = -\log_{f_j} g_j$.

Following the above procedure, the bit size of the public key

$$\mathsf{pk} = (E_1, (a_1, b_1), \cdots, (a_r, b_r))$$

can be reduced to about $(2r + 2)\log_2(p)$.

As mentioned in [Hou26, Section 6.2], the twist trick can be utilized to further enhance the performance of CSIDH-LDO. Under these circumstances, it follows from Section 2.1 that one needs to transmit $\mathcal{E} = \{(E_j, P_{sj}, Q_{sj}, P_{tj}, Q_{tj}) \mid j = 1, \cdots, r\}$ to the other, where $P_{sj},\ Q_{sj} \in E_j[M_s],\ M_s \mid p+1$ and $P_{tj},\ Q_{tj} \in E_j[M_t],\ M_t \mid p-1$. The points P_{tj} and Q_{tj} are mapped from P'_{tj} and Q'_{tj} on the quadratic twist E_j^t by a twisting isomorphism over $\mathbb{F}_{p^4}$. The (M_sM_t)-isogeny $\varphi_j,\ j = 1, \cdots, r$ is split into $\varphi_j = \varphi_{tj} \circ \varphi_{sj}$, with $\ker(\varphi_{sj}) = \langle P_{sj} \rangle$, $\ker(\widehat{\varphi_{sj-1}}) = \langle \widehat{\varphi_{tj-1}}(Q_{sj}) \rangle$, $\ker(\varphi_{tj}) = \langle \varphi_{sj}(P_{tj}) \rangle$ and $\ker(\widehat{\varphi_{tj-1}}) = \langle Q_{tj} \rangle$.

Similar to the standard setting, Q_{sj+1} and $Q_{tj+1},\ j = 1, \cdots, r$ need not be transmitted as they can be respectively recovered by:

(1) Finding R_{s1} and R_{t1} such that $\langle P_{s1}, R_{s1} \rangle = E_1[M_s]$ and $\langle P_{t1}, R_{t1} \rangle = E_1[M_t]$, respectively;

(2) Computing $\varphi_{t1} \circ \varphi_{s1}(R_{s1})$ and $\varphi_{t1} \circ \varphi_{s1}(R_{t1})$, then it holds that $\varphi_{t1} \circ \varphi_{s1}(R_{s1}) = Q_{s2}$ and $\varphi_{t1} \circ \varphi_{s1}(R_{t1}) = Q_{t2}$. The points $Q_{sj+1} = \varphi_{tj} \circ \varphi_{sj}(Q_{sj})$ and $Q_{tj+1} = \varphi_{tj} \circ \varphi_{sj}(Q_{tj}),\ j = 2, \cdots, r$ can be successively derived.

In what follows, we briefly justify why this approach allows us to recover Q_{sj} and Q_{tj}. It can be deduced that $\widehat{\varphi_{s1}} \circ \widehat{\varphi_{t1}} \circ \varphi_{t1} \circ \varphi_{s1}(R_{s1}) = [M_sM_t]R_{s1} = \mathcal{O}_{E_1}$ and $\widehat{\varphi_{t1}} \circ \varphi_{t1} \circ \varphi_{s1}(R_{t1}) = \varphi_{s1}([M_t]R_{t1}) = \mathcal{O}_{\varphi_{s1}(E_1)}$. Consequently, the points $\widehat{\varphi_{t1}} \circ \varphi_{t1} \circ \varphi_{s1}(R_{s1})$ and $\varphi_{t1} \circ \varphi_{s1}(R_{t1})$ generate $\ker(\widehat{\varphi_{s1}})$ and $\ker(\widehat{\varphi_{t1}})$, respectively.

The coefficients of $E_2, \cdots, E_r$ can also be recovered using E_1 together with $P_{sj}, P_{tj},\ j = 1, \cdots, r$. It remains to compress P_{sj} and P_{tj}. The procedure is similar to the situation in which the twist trick is not employed. One can generate two basis $\langle U_{sj}, V_{sj} \rangle = E_j[M_s]$ and $\langle U_{tj}, V_{tj} \rangle = E_j[M_t]$, and separately compute the coordinates $a_{sj}, b_{sj} \in \mathbb{Z}/M_s\mathbb{Z}$ and $a_{tj}, b_{tj} \in \mathbb{Z}/M_t\mathbb{Z}$ for P_{sj} and P_{tj} by executing several pairing computations and solving the corresponding discrete logarithms. Finally, the public key is compressed as

$$\mathsf{pk} = (E_1, (a_{sj}, b_{sj}), (a_{tj}, b_{tj})),\ j = 1, \cdots, r$$

with bit-length as $(4r + 2)\log_2(p)$.

4.2 Optimized Methods for a More Compact Compression

In this subsection, we present optimized approaches that yield a more compact and efficient compression than the method described in Section 4.1. We investigate both the standard setting where the twist trick is not applied, together with the scenario in which the

twist trick is leveraged, achieving a compressed public key of around $(r+2)\log_2(p)$ and $(2r+2)\log_2(p)$ bits, respectively.

It is worth noting that the recoveries for the curves $E_2, \ldots, E_r$ and the points $Q_1, \ldots, Q_r$ ($Q_{s1}, \cdots, Q_{sr}$, $Q_{t1}, \cdots, Q_{tr}$ if the twist trick is leveraged) are identical to those discussed in Section 4.1. We focus exclusively on the compression of $P_j \in E_j[M],\ M \mid p+1$ ($P_{sj} \in E[M_s]$ and $P_{tj} \in E[M_t],\ M_s \mid p+1,\ M_t \mid p-1$ if the twist trick is used).

Before presenting our methods, we first state two core propositions in this subsection, which generalize [LLC+24, Proposition 3] to our situation. In our setting, M is of the form $M = \prod_{i=1}^{n} \ell_i^{e_i}$ with $e_i \geq 1$, rather than $M = \prod_{i=1}^{n} \ell_i$, and we also treat the Weil pairing case. These propositions can be used to construct a point Q of order M such that P, Q forms a basis of $E[M]$ for a given point P of order M, which is one of the main procedures for encoding $P_j \in E_j[M]$, $P_{sj} \in E_j[M_s]$ or $P_{tj} \in E_j[M_t],\ j = 1, \cdots, r$.

Proposition 1. *Let $P \in E(\mathbb{F}_{p^2})$ denote a point of order M, where $M = \prod_{i=1}^{n} \ell_i^{e_i},\ e_i \geq 1$ such that $\gcd\left(\ell_i^{e_i}, \ell_j^{e_j}\right) = 1, i \neq j$ and $\gcd\left(M, \frac{p^2-1}{M}\right) = 1$. Assume that $Q \in E(\mathbb{F}_{p^2})$, then the multiplicative orders of the (reduced) Tate pairing*

$$ord(e_{T,M}(P,Q)) = \prod_{\substack{i \in \{1,\cdots,n\}, \\ e_{T,\ell_i^{e_i}}\left(P^{(i)},Q\right) \neq 1}} \ell_i^{m_i},$$

where $0 < m_i \leq e_i$ is the multiplicative order of $e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right)$, and $P^{(i)} = \left[\frac{M}{\ell_i^{e_i}}\right] P \in E[\ell_i^{e_i}]$. Additionally, if $m_i = e_i$ for all i, then $\langle P, Q' \rangle = E[M]$, where $Q' = \left[\frac{p+1}{M}\right] Q$.

Proof. Let $s_i = \left(\frac{M}{\ell_i^{e_i}}\right)^{-1} \bmod \ell_i^{e_i}$ as $\gcd\left(\ell_i^{e_i}, \ell_j^{e_j}\right) = 1, i \neq j$. By the Chinese Remainder Theorem it holds that $P = \sum_{i=1}^{n} [s_i] P^{(i)}$, and

$$e_{T,M}(P,Q) = e_{T,M}\left(\sum_{i=1}^{n} [s_i] P^{(i)}, Q\right) = \prod_{i=1}^{n} e_{T,M}\left(P^{(i)}, Q\right)^{s_i}. \tag{1}$$

It follows from [Bla05, Theorem IX.9] that

$$e_{T,M}\left(P^{(i)}, Q\right) = e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right) \in \mu_{\ell_i^{e_i}}. \tag{2}$$

As $\gcd\left(s_i, \ell_i^{e_i}\right) = 1$, by Eq. (2) we obtain

$$\operatorname{ord}\left(e_{T,M}\left(P^{(i)}, Q\right)^{s_i}\right) = \operatorname{ord}\left(e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right)^{s_i}\right) = \operatorname{ord}\left(e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right)\right). \tag{3}$$

Moreover, since $e_{T,\ell_i^{e_i}}(P^{(i)}, Q) \in \mu_{\ell_i^{e_i}}$ and $\gcd\left(\ell_i^{e_i}, \ell_j^{e_j}\right) = 1, i \neq j$, from Eqs. (1) and (3) we deduce

$$\operatorname{ord}(e_{T,M}(P,Q)) = \prod_{i=1}^{n} \operatorname{ord}\left(e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right)\right). \tag{4}$$

Let $Q^{(i)} = \left[\frac{p+1}{\ell_i^{e_i}}\right] Q \in E[\ell_i^{e_i}]$. Then it yields that

$$e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q\right) = e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)^{\frac{p+1}{\ell_i^{e_i}}} \in \mu_{\ell_i^{e_i}}. \tag{5}$$

By $\gcd\left(\ell_i^{e_i}, \frac{M}{\ell_i^{e_i}}\right) = 1$ and $\gcd\left(M, \frac{p^2-1}{M}\right) = 1$ we obtain $\gcd\left(\ell_i^{e_i}, \frac{p^2-1}{\ell_i^{e_i}}\right) = 1$. Using Eqs. (4) and (5) yields

$$\operatorname{ord}(e_{T,M}(P,Q)) = \prod_{i=1}^{n} \operatorname{ord}\left(e_{T,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)\right). \tag{6}$$

We now consider $e_{T,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)$. There are following two situations: (1) $Q^{(i)}\in\left\langle P^{(i)}\right\rangle$, then $e_{T,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)=1$; (2) $Q^{(i)}\notin\left\langle P^{(i)}\right\rangle$, then $e_{T,\ell_i^{e_i}}(P^{(i)},Q^{(i)})\neq 1$ and the order of $e_{T,\ell_i^{e_i}}(P^{(i)},Q^{(i)})$ equals to $\ell_i^{m_i}$, where $0<m_i\leq e_i$.

Additionally, from Eq. (5) and $\gcd\left(\ell_i^{e_i},\frac{p^2-1}{\ell_i^{e_i}}\right)=1$ it follows that $e_{T,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)\neq 1$ if and only if $e_{T,\ell_i^{e_i}}\left(P^{(i)},Q\right)\neq 1$. Based on the above analysis, by Eq. (6) we have

$$\mathrm{ord}(e_{T,M}(P,Q))=\prod_{i=1}^{n}\mathrm{ord}\left(e_{T,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)\right)=\prod_{\substack{i\in\{1,\cdots,n\},\\ e_{T,\ell_i^{e_i}}\left(P^{(i)},Q\right)\neq 1}}\ell_i^{m_i}.$$

It is worth noting that $\gcd\left(\ell_i^{e_i},\frac{p^2-1}{\ell_i^{e_i}}\right)=1$ ensures that the final exponentiation $\frac{p^2-1}{\ell_i^{e_i}}$ does not reduce the order of each Tate pairing $e_{T,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)$, $i\in\{1,\cdots,n\}$, hence it is possible that $m_i=e_i$ for each i.

If $m_i=e_i$ for $i=1,\cdots,n$, then $\mathrm{ord}(e_{T,M}(P,Q))=M$. By $\gcd\left(M,\frac{p^2-1}{M}\right)=1$ we deduce $\mathrm{ord}\left(e_{T,M}\left(P,Q'\right)\right)=\mathrm{ord}\left(e_{T,M}(P,Q)^{\frac{p+1}{M}}\right)=M$. As $P,Q'\in E[M]$, it follows from $\mathrm{ord}(e_{T,M}(P,Q'))=M$ and Eq. (6) that $e_{T,M}(P^{(i)},Q^{(i)})$ generates $\mu_{\ell_i^{e_i}}$, and thus $\left\langle P^{(i)},Q^{(i)}\right\rangle=\left\langle P^{(i)},\left[\frac{M}{\ell_i^{e_i}}\right]Q'\right\rangle=E[\ell_i^{e_i}]$ for each i. Furthermore, it should be noted that $E[M]\cong\bigoplus_{i=1}^{n}E[\ell_i^{e_i}]$ by the structure theorem for finitely generated abelian groups. Therefore, we have $\langle P,Q'\rangle=E[M]$, which ends the proof. □

Note that if the embedding degree with respect to M (the smallest integer k such that $M\mid p^k-1$) for $e_{T,M}(P,Q)$ in Proposition 1 is one, then M should satisfy $\gcd\left(M,\frac{p-1}{M}\right)=1$. As for the Weil pairing, one can derive a similar result summarized in Proposition 2.

Proposition 2. *Let $P\in E(\mathbb{F}_{p^2})$ denote a point of order M, where $M=\prod_{i=1}^{n}\ell_i^{e_i}$, $e_i\geq 1$ such that $\gcd\left(\ell_i^{e_i},\ell_j^{e_j}\right)=1, i\neq j$. Assume that $Q\in E[M]$, then the multiplicative orders of the Weil pairing*

$$ord(e_{W,M}(P,Q))=\prod_{\substack{i\in\{1,\cdots,n\},\\ e_{W,\ell_i^{e_i}}\left(P^{(i)},Q\right)\neq 1}}\ell_i^{m_i},$$

where $0<m_i\leq e_i$ is the multiplicative order of $e_{W,\ell_i^{e_i}}\left(P^{(i)},Q\right)$, and $P^{(i)}\in E[\ell_i^{e_i}]$. Additionally, if $m_i=e_i$ for all i, then $\langle P,Q\rangle=E[M]$.

Proof. With the same notation as Proposition 1, and let $P^{(i)}=\left[\frac{M}{\ell_i^{e_i}}\right]P\in E[\ell_i^{e_i}]$, $s_i=\left(\frac{M}{\ell_i^{e_i}}\right)^{-1}\bmod\ell_i^{e_i}$. Then $P=\sum_{i=1}^{n}[s_i]P^{(i)}$, and

$$e_{W,M}(P,Q)=\prod_{i=1}^{n}e_{W,M}\left(P^{(i)},Q\right)^{s_i}. \tag{7}$$

From [Mil86, Lemma 16.2] we have

$$e_{W,M}(P,Q)^{\frac{M}{\ell_i^{e_i}}}=e_{W,\ell_i^{e_i}}\left(\left[\frac{M}{\ell_i^{e_i}}\right]P,\left[\frac{M}{\ell_i^{e_i}}\right]Q\right)=e_{W,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right), \tag{8}$$

where $Q^{(i)}=\left[\frac{M}{\ell_i^{e_i}}\right]Q$. By the bilinearity, it holds that $e_{W,M}(P,Q)^{\frac{M}{\ell_i^{e_i}}}=e_{W,M}\left(P^{(i)},Q\right)$. Hence, from Eq. (8) it follows that

$$e_{W,\ell_i^{e_i}}\left(P^{(i)},Q^{(i)}\right)=e_{W,M}\left(P^{(i)},Q\right). \tag{9}$$

As $\gcd(s_i, \ell_i^{e_i}) = 1$, by Eq. (9) we obtain

$$\mathrm{ord}\left(e_{W,M}\left(P^{(i)}, Q\right)^{s_i}\right) = \mathrm{ord}\left(e_{W,M}\left(P^{(i)}, Q\right)\right) = \mathrm{ord}\left(e_{W,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)\right). \quad (10)$$

As $\gcd\left(\ell_i^{e_i}, \ell_j^{e_j}\right) = 1, i \neq j$, by Eqs. (7) and (10) we obtain $\mathrm{ord}(e_{W,M}(P,Q)) = \prod_{i=1}^{n} \mathrm{ord}\left(e_{W,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)\right)$. The remainder of this proof is almost the same as that of Proposition 1 by replacing the Tate pairing $e_{T,M}(\cdot,\cdot)$ by the Weil pairing $e_{W,M}(\cdot,\cdot)$, and discussing the order of $e_{W,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)$. The difference is that the condition $\gcd\left(M, \frac{p^2-1}{M}\right)$ is no longer needed since the Weil pairing does not execute the final exponentiation, which means that the order of $e_{W,\ell_i^{e_i}}\left(P^{(i)}, Q^{(i)}\right)$ can not be reduced.

□

Remark 2. In most scenarios, Proposition 1 is preferred since the Tate pairing is faster than the Weil pairing. However, if $\gcd\left(M, \frac{p^2-1}{M}\right) \neq 1$ (or $\gcd\left(M, \frac{p-1}{M}\right) \neq 1$ if the embedding degree with respect to M is 1), using Proposition 1 is vulnerable to checking the point order correctly due to the pairing degeneracy caused by the final exponentiation. For instance, assume that $\gcd\left(M, \frac{p^2-1}{M}\right) = \ell \in \mathbb{Z}$. Given two points Q and $\widetilde{Q}$ of orders M and M/ℓ, respectively, the multiplicative orders of both Tate pairings $e_{T,M}(P,Q) = f_{M,P}(Q)^{\frac{p^2-1}{M}}$ and $e_{T,M}(P,\widetilde{Q}) = f_{M,P}(\widetilde{Q})^{\frac{p^2-1}{M}}$, $P \in E[M]$ are M/ℓ, which prevents the point-generation algorithms (See Algorithms 3 and 5 for more details) from identifying points of full order correctly. To figure out this problem, we should apply Proposition 2 and the Weil pairing instead to prevent the order degeneration.

In the CSIDH-LDO setting, the parameters satisfy that $\gcd\left(M, \frac{p^2-1}{M}\right) = 1$ when the twist trick is not exploited, and $\gcd\left(M_s, \frac{p^2-1}{M_s}\right) = 1$, $\gcd\left(M_t, \frac{p-1}{M_t}\right) \neq 1$ when the twist trick is applied. Consequently, we provide two effective encoding methods for P_j, $j = 1, \cdots, r$, using the **Tate pairing** in Section 4.2.1. For the compression of P_{sj}, $j = 1, \cdots, r$, we can use the approaches in Section 4.2.1, while we should use the **Weil pairing** with embedding degree 1 for encoding P_{tj}, $j = 1, \cdots, r$. See Section 4.2.2 for more details.

4.2.1 Two Effective Approaches for Encoding P_j: The Standard Setting

We now consider the situation where the twist trick is not employed, and provide two effective methods for the compression and decompression of points $P_j \in E[M]$, $M = \prod_{i=1}^{n} \ell_i^{e_i} \mid p+1$, $j = 1, \cdots, r$. Our attention is restricted to $P := P_1$ on $E := E_1$; the procedures for $P_2, \cdots, P_r$ are the same.

Method I: employing the Chinese Remainder Theorem. We now introduce the first method, which compresses the public key to a bit-length of about $(r+2)\log_2(p) + rn$, where n represents the number of different prime factors of M. The fundamental components for the compression of P consist of three main steps: (1) Generating a torsion basis $\langle U, V\rangle = E[M]$; (2) Computing two (reduced) Tate pairings $e_{T,M}(P,U)$ and $e_{T,M}(P,V)$; (3) Solving a discrete logarithm utilizing the Chinese Remainder Theorem.

Torsion basis generation. We first consider the generation of a torsion basis $\langle U, V\rangle = E[M]$. To generate U, we employ the generalized naive approach proposed in [LLC+24]. Specifically, we sample a point $U \in \mathbb{F}_{p^2}$ using Elligator 2 [BHKL13] and multiply it by $(p+1)/M$, then determine the indices i for which $\mathrm{ord}\left([M/\ell_i^{e_i}]U\right) < \ell_i^{e_i}$ and store them in the set I_U. If $\mathrm{ord}\left([M/\ell_i^{e_i}]U\right) < \ell_i^{e_i}$, then $\ell_i^{e_i} \nmid \mathrm{ord}(U)$. We need to sample another point U' and check whether $\mathrm{ord}([(p+1)/\ell_i^{e_i}]U') = \ell_i^{e_i}$ for each $i \in I_U$ when $I_U \neq \emptyset$. If so, we update

U as $U + [(p+1)/\ell_i^{e_i}]U'$ and and remove i from I_U. By repeating the above process until $I_U = \emptyset$, we derive U of order M. The concrete procedure is presented in Algorithm 20.

As for the generation of V, we can use Proposition 1. The details are as follows. After sampling a point $V \in E(\mathbb{F}_{p^2})$, we verify the multiplicative order of the reduced Tate pairing $e_{T,\ell_i^{e_i}}([M/\ell_i^{e_i}]\,U, V)$ and store the indices i such that ord $\left(e_{T,\ell_i^{e_i}}([M/\ell_i^{e_i}]\,U, V)\right) < \ell_i^{e_i}$ in I_V. Similarly, we sample another $V' \in E(\mathbb{F}_p)$ and determine the multiplicative order of $e_{T,\ell_i^{e_i}}([M/\ell_i^{e_i}]\,U, V)$ for each $i \in I_V$ if $I_V \neq \emptyset$. We update V as $V + [M/\ell_i^{e_i}]\,V'$, and delete i from I_V if ord $\left(e_{T,\ell_i^{e_i}}([M/\ell_i^{e_i}]\,U, V')\right) = \ell_i^{e_i}$. Once I_V is empty, the above repeated procedure is complete. Multiplying the result by $\frac{p+1}{M}$ yields the final point V such that $\langle U, V\rangle = E[M]$. Algorithm 3 describes the detailed procedure for generating V from U.

Algorithm 3 GenerateV: Generate a point V of order $M = \prod_{i=1}^{n} \ell_i^{e_i}$ such that $\langle U, V\rangle = E[M]$

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I = \{1, \cdots, n\}$, the cofactor $(p+1)/M$, a point $U \in E(\mathbb{F}_{p^2})$ of order M;
Ensure: A point $V \in E(\mathbb{F}_{p^2})$ such that $\langle U, V\rangle = E[M]$.

```
Generate a point V ∈ E(F_{p^2}) using Elligator 2
{f_i | i ∈ I} ← { e_{T,M}(U,V)^{M/ℓ_i^{e_i}} | i ∈ I }, I_V ← [ ]
for i ∈ I do
    if ord(f_i) < ℓ_i^{e_i} then
        I_V ← I_V ∪ {i}
    end if
end for
U ← [M/∏_{i∈I_V} ℓ_i^{e_i}]U
while I_V ≠ ∅ do
    Generate a point V' ∈ E(F_{p^2}) using Elligator 2
    L ← ∏_{i∈I_V} ℓ_i^{e_i}, V' ← [M/L]V', {f'_i | i ∈ I_V} ← { e_L(U,V')^{L/ℓ_i^{e_i}} | i ∈ I_V }
    for i ∈ I_V do
        if ord(f'_i) = ℓ_i^{e_i} then
            I_V ← I_V \ {i}, V ← V + [L/ℓ_i^{e_i}]V', U ← [ℓ_i^{e_i}]U
        end if
    end for
end while
V ← [(p+1)/M]V
return V
```

Remark 3. To ensure correct decompression, the compressing party should transmit the random seeds required in the Elligator 2 algorithm, so that both parties generate the same basis U and V. We can precompute a table as a public parameter and send the index of the corresponding seed in this table. This additional transmission overhead is negligible compared to the coordinates of P to the basis U, V. The compressing party can either transmit the index sets I_U and I_V to reduce the computational cost on the decompression side or omit them to save bandwidth, reflecting an inherent trade-off.

Pairing computation and discrete logarithm. After generating the basis U, V, the point P can be represented as $P = [a]U + [b]V$. Then the compressing party computes two Tate pairings: $g = e_{T,M}(P, U)$ and $h = e_{T,M}(P, V)$.

It is worth noting that we cannot directly solve the discrete logarithm $-\log_h g$ or $-\log_g h$ to respectively obtain $a^{-1}b$ or ab^{-1}, since both a and b may be non-invertible in $\mathbb{Z}/M\mathbb{Z}$, where $M = \prod_{i=1}^{n} \ell_i^{e_i}$. In the following, we discuss how to figure out this problem.

It follows from $P = [a]U + [b]V,\ a, b \in \mathbb{Z}/M\mathbb{Z}$ that $\left[\frac{M}{\ell_i^{e_i}}\right] P = \left[a^{(i)}\right] U^{(i)} + \left[b^{(i)}\right] V^{(i)}$ for every $\ell_i,\ i = 1, \cdots, n$, where $U^{(i)} = \left[\frac{M}{\ell_i^{e_i}}\right] U,\ V^{(i)} = \left[\frac{M}{\ell_i^{e_i}}\right] V,\ a^{(i)} \equiv a \bmod \ell_i^{e_i}$ and $b^{(i)} \equiv b \bmod \ell_i^{e_i}$. Then one of $a^{(i)}$ and $b^{(i)}$ must be invertible in $\mathbb{Z}/\ell_i^{e_i}\mathbb{Z}$. Otherwise, it yields that

$$\gcd\left(a^{(i)}, \ell_i^{e_i}\right) = \ell_i^{m_i},\ \gcd\left(b^{(i)}, \ell_i^{e_i}\right) = \ell_i^{n_i},\ m_i, n_i \in \{1, \cdots, e_i\}$$

as ℓ_i is prime, which implies $\operatorname{ord}\left(\left[a^{(i)}\right] U^{(i)} + \left[b^{(i)}\right] V^{(i)}\right) < \ell_i^{e_i}$. This contradicts P being a full M-torsion point.

Additionally, by $\langle U, V\rangle = E[M]$ we have $e_{T,M}(U, V)$ is a generator of the group of M-th roots of unity μ_M. Thus, the pairing value $f^{(i)} = f^{\frac{M}{\ell_i^{e_i}}} = e_{T,M}(U, V)^{\frac{M}{\ell_i^{e_i}}}$ is a generator of $\mu_{\ell_i^{e_i}}$. Therefore, if $a^{(i)} \in \mathbb{Z}/\ell_i^{e_i}\mathbb{Z}$ is invertible, then $h^{(i)} = h^{\frac{M}{\ell_i^{e_i}}} = e_{T,M}(P, V)^{\frac{M}{\ell_i^{e_i}}} = e_{T,M}(U, V)^{a^{(i)} \cdot \frac{M}{\ell_i^{e_i}}}$ is also a generator of $\mu_{\ell_i^{e_i}}$, and we can solve the discrete logarithm $\log_{h^{(i)}} g^{(i)}$ to derive $\left(a^{(i)}\right)^{-1} \cdot b^{(i)} \equiv a^{-1} \cdot b \bmod \ell_i^{e_i}$, where $g^{(i)} = e_{T,M}(P, U)^{\frac{M}{\ell_i^{e_i}}} = e_{T,M}(U, V)^{-b^{(i)} \cdot \frac{M}{\ell_i^{e_i}}}$. Similarly, if $b^{(i)}$ is invertible, one can obtain $\left(b^{(i)}\right)^{-1} \cdot a^{(i)}$ by solving the discrete logarithm $\log_{g^{(i)}} h^{(i)}$. For each $i \in \{1, \cdots, n\}$, we compute the following tuple

$$\left(S^{(i)},\ label^{(i)}\right) = \begin{cases} \left(\left(a^{(i)}\right)^{-1} \cdot b^{(i)} \bmod \ell_i^{e_i}, 0\right), & a^{(i)} \text{ is invertible,} \\ \left(\left(b^{(i)}\right)^{-1} \cdot a^{(i)} \bmod \ell_i^{e_i}, 1\right), & b^{(i)} \text{ is invertible.} \end{cases} \tag{11}$$

By the Chinese Remainder Theorem, one can derive $S \in \mathbb{Z}/M\mathbb{Z}$ such that $S \equiv S^{(i)} \bmod \ell_i^{e_i},\ i = 1, \cdots, n$. Algorithm 4 illustrates the whole procedure for the compression of $P := P_1$, where CRT denotes the Chinese Remainder Theorem.

Algorithm 4 CompressP_MethodI: Compress P using Method I

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I = \{1, \cdots, n\}$, the cofactor $(p+1)/M$, a point P of order M;
Ensure: A coordinate $S \in \mathbb{Z}/M\mathbb{Z}$ and $label \in \mathbb{Z}$.

```
U ← GenerateU(E, I, (p+1)/M))                ▷ Algorithm 20
V ← GenerateV(E, I, (p+1)/M, U)              ▷ Algorithm 3
g ← e_{T,M}(P, U), h ← e_{T,M}(P, V)
{g^(i) | i ∈ I} ← {g^{M/ℓ_i^{e_i}} | i ∈ I}, {h^(i) | i ∈ I} ← {h^{M/ℓ_i^{e_i}} | i ∈ I}
for i ∈ I do
    if ord(h^(i)) = ℓ_i^{e_i} then
        S^(i) ← log_{h^(i)} g^(i), label^(i) ← 0
    else
        S^(i) ← log_{g^(i)} h^(i), label^(i) ← 1
    end if
end for
S ← CRT(S^(1), ··· , S^(n)), label ← (label^(1), ··· , label^(n))_2
return S, label
```

Similarly, the corresponding elements $S_2, \cdots, S_r \in \mathbb{Z}/M\mathbb{Z}$ for the compression of $P_2, \cdots, P_r$, respectively, can also be obtained by executing the above procedures. Finally, the public key becomes

$$\mathsf{pk} = \left(E_1, (S_1, label_1), \cdots, (S_r, label_r)\right),$$

where $label_j,\ j = 1, \cdots, r$ is a positive integer such that the binary expansion $label_j = \left(label_j^{(1)}, label_j^{(2)}, \cdots, label_j^{(n)}\right)_2$. The size of the public key pk is around $(r+2)\log_2(p)+rn$, which is shorter than that in Section 4.1 (with bit-length $(2r+2)\log_2(p)$). Furthermore, Method I reduces the computational cost, saving one pairing computation and one discrete logarithm operation during the encoding of each P_j.

The procedure for the decompression. We now explain how to derive the kernels $\langle P_1\rangle, \cdots, \langle P_r\rangle$ from the compressed key $((S_1, label_1), \cdots, (S_r, label_r))$. Similarly, we only consider the decompression of $P := P_1$ with respect to $(S, label) := (S_1, label_1)$ for simplicity. As mentioned before, the decompressing party can generate the same basis U, V as the compressing party.

After receiving $(S, label)$, one can determine

$$\widetilde{S^{(i)}} = \begin{cases} 1, & label^{(i)} = 0, \\ S \bmod \ell_i^{e_i}, & label^{(i)} = 1, \end{cases} \quad \overline{S^{(i)}} = \begin{cases} S \bmod \ell_i^{e_i}, & label^{(i)} = 0, \\ 1, & label^{(i)} = 1. \end{cases} \tag{12}$$

By the Chinese Remainder Theorem, one can obtain $\widetilde{S}$ and $\overline{S}$ such that $\widetilde{S} \equiv \widetilde{S^{(i)}} \bmod \ell_i^{e_i}$ and $\overline{S} \equiv \overline{S^{(i)}} \bmod \ell_i^{e_i}$, respectively. In the following, we provide the following lemma to claim that $\left[\widetilde{S}\right]U + \left[\overline{S}\right]V$ and $P = [a]U + [b]V$ generate the same subgroup.

Lemma 1. *Let the notations be denoted as above. Assume that $\widetilde{S},\ \overline{S} \in \mathbb{Z}/M\mathbb{Z}$ satisfy Eq. (12), then $\left\langle \left[\widetilde{S}\right]U + \left[\overline{S}\right]V \right\rangle = \langle P\rangle$.*

Proof. Let $K = \left[\widetilde{S}\right]U + \left[\overline{S}\right]V$. By $\gcd\left(\ell_i^{e_i}, \ell_j^{e_j}\right) = 1$ for any $i \neq j$, it is equivalent to showing that

$$\left\langle \left[\frac{M}{\ell_i^{e_i}}\right] K \right\rangle = \left\langle \left[a^{(i)}\right] U^{(i)} + \left[b^{(i)}\right] V^{(i)} \right\rangle = \left\langle \left[\frac{M}{\ell_i^{e_i}}\right] P \right\rangle$$

for all $i \in \{1, \cdots, n\}$. If $a^{(i)}$ is invertible in $\mathbb{Z}/\ell_i^{e_i}\mathbb{Z}$, then from Eq. (11) it follows that $label^{(i)} = 0$ and $\gcd\left(\left(a^{(i)}\right)^{-1}, \ell_i^{e_i}\right) = 1$. Additionally, by Eqs. (11) and (12) we obtain

$$\left[\frac{M}{\ell_i^{e_i}}\right] K = \left[\left(a^{(i)}\right)^{-1}\right] \left(\left[a^{(i)}\right] U^{(i)} + \left[b^{(i)}\right] V^{(i)}\right) \in \left\langle \left[\frac{M}{\ell_i^{e_i}}\right] P \right\rangle.$$

As $\gcd\left(\left(a^{(i)}\right)^{-1}, \ell_i^{e_i}\right) = 1$, we have $\operatorname{ord}\left(\left[\frac{M}{\ell_i^{e_i}}\right] K\right) = \operatorname{ord}\left(\left[\frac{M}{\ell_i^{e_i}}\right] P\right) = \ell_i^{e_i}$, which implies $\left\langle \left[\frac{M}{\ell_i^{e_i}}\right] K \right\rangle = \left\langle \left[\frac{M}{\ell_i^{e_i}}\right] P \right\rangle$. A similar proof goes for the case where $b^{(i)}$ is invertible in $\mathbb{Z}/\ell_i^{e_i}\mathbb{Z}$. By the arbitrary of i, we end the proof of this lemma.

□

Method II: Efficient Joint Torsion Basis Generation and Pairing Computation. In the following, we propose another method that compresses the public key to a bit-length of $(r+2)\log_2(p)$. Compared to Method I, this approach saves the scalar multiplications and simultaneously accomplishes part of the pairing computations in the torsion basis generation. Moreover, it avoids the transmission of extra label information in the discrete logarithm phase. Similarly, we concentrate on the compression and decompression of $P := P_1$ on $E := E_1$.

The main motivation behind this approach is to eliminate the redundancies in the discrete logarithm phase. For Method I, neither $e_{T,M}(P, U)$ nor $e_{T,M}(P, V)$ is the generator of μ_M, resulting in a more complex discrete logarithm solving and extra transmission of

the label information. Inspired by this, we make one of U and V form a basis with P, then either $e_{T,M}(P,U)$ or $e_{T,M}(P,V)$ is the generator of μ_M. Thus, the discrete logarithm can be recovered using one of these two pairing values as the base.

Assume that U is constructed such that $\langle P,U\rangle = E[M]$. Recall from Proposition 1 and Algorithm 3 that it is necessary to determine the multiplicative orders of certain Tate pairings associated with P and U. If $e_{T,M}(P,U)$ can be derived simultaneously, the overall computational cost can be further reduced. The framework of Method II is as follows: (1) Generating U such that $\langle P,U\rangle = E[M]$ and deriving $g = e_{T,M}(P,U)$; (2) Generating V such that $\langle U,V\rangle = E[M]$; (3) Computing the reduced Tate pairing $h = e_{T,M}(P,V)$; (4) Solving the discrete logarithm $a^{-1}b \equiv \log_g h \bmod M$.

Torsion basis generation and pairing computation. We now present the detailed computational procedure for the torsion basis generation and pairing computation. For the generation of U, we modify Algorithm 3 to simultaneously obtain the reduced Tate pairing $e_{T,M}(P,U)$. Algorithm 5 illustrates the process of generating U such that $\langle P,U\rangle = E[M]$, and computing $e_{T,M}(P,U)$ at the same time.

Algorithm 5 GenerateU&TatePairingU: Generate a point U of order $M = \prod_{i=1}^{n} \ell_i^{e_i}$ such that $\langle P,U\rangle = E[M]$ and derive the reduced Tate pairing $e_{T,M}(P,U)$

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I = \{1,\cdots,n\}$, the cofactor $(p+1)/M$, a point $P \in E(\mathbb{F}_{p^2})$ of order M;

Ensure: A point $U \in E(\mathbb{F}_{p^2})$ such that $\langle U,P\rangle = E[M]$ and the reduced Tate pairing $e_{T,M}(P,U)$.

Generate a point $U \in E(\mathbb{F}_{p^2})$ using Elligator 2
$e \leftarrow e_{T,M}(P,U),\ I_U \leftarrow [\,],\ \{f_i \mid i \in I\} \leftarrow \left\{e^{M/\ell_i^{e_i}} \mid i \in I\right\}$ ▷ Initialize e
for $i \in I$ **do**
  if $\mathrm{ord}(f_i) < \ell_i^{e_i}$ **then**
    $I_U \leftarrow I_U \cup \{i\}$
  end if
end for
$P \leftarrow \left[M/\prod_{i\in I_U} \ell_i^{e_i}\right] P$
while $I_U \neq \emptyset$ **do**
  Generate a point $U' \in E(\mathbb{F}_{p^2})$ using Elligator 2
  $L \leftarrow \prod_{i\in I_U} \ell_i^{e_i},\ e' \leftarrow e_{T,L}(P,U')$
  $\{f_i' \mid i \in I_U\} \leftarrow \left\{(e')^{L/\ell_i^{e_i}} \mid i \in I_U\right\},\ U' \leftarrow [M/L]\,U'$
  for $i \in I_U$ **do**
    if $\mathrm{ord}(f_i') = \ell_i^{e_i}$ **then**
      $I_U \leftarrow I_U \setminus \{i\},\ U \leftarrow U + [L/\ell_i^{e_i}]\,U',\ P \leftarrow [\ell_i]\,P,\ e \leftarrow e \cdot f_i'$ ▷ Update e
    end if
  end for
end while
$U \leftarrow [(p+1)/M]\,U$
return U, e ▷ $e = e_{T,M}(P,U)$

The following proposition claims that Algorithm 5 returns the correct value of the reduced Tate pairing $e_{T,M}(P,U)$.

Proposition 3. *Algorithm 5 correctly computes the reduced Tate pairing $e_{T,M}(P,U)$.*

Proof. We claim that the returned value e equals $e_{T,M}(P,U)$. We prove this by demonstrating that Algorithm 5 incrementally constructs the point U and accumulates the corresponding pairing factors, maintaining the bilinearity property at each step.

To clarify the iterative process, let m denote the total number of iterations of the `while` loop (Lines 9-18 of Algorithm 5). If $m = 0$, then we already obtain U and the initialized pairing value e in Line 2 is exactly $e_{T,M}(P, U)$. We now consider the case where $m > 0$. For the k-th iteration ($1 \le k \le m$), let $I^{(k)}$ denote the active index set at the start of this iteration, and define the order $L_k = \prod_{i \in I^{(k)}} \ell_i^{e_i}$ as computed in Line 11 of Algorithm 5. Furthermore, let $S^{(k)} \subseteq I^{(k)}$ be the subset of indices successfully resolved and removed from the active set $I^{(k)}$ during this k-th iteration. Thus, it holds that $I^{(k+1)} = I^{(k)} \setminus S^{(k)}$.

Let U_0 denote the initial point established before the `while` loop (the point constructed in Line 1), and define U'_k to be the point generated in Line 10 of Algorithm 5 during the k-th iteration. The final point U is constructed by adding incremental updates (scaled from U'_k in Lines 12 and 15) to U_0 whenever an index is resolved (Line 15 of Algorithm 5) as

$$U = U_0 + \sum_{k=1}^{m} \sum_{i \in S^{(k)}} \left[\frac{M}{L_k} \cdot \frac{L_k}{\ell_i^{e_i}} \right] U'_k = U_0 + \sum_{k=1}^{m} \sum_{i \in S^{(k)}} \left[\frac{M}{\ell_i^{e_i}} \right] U'_k.$$

By the bilinearity of the reduced Tate pairing, the target pairing $e_{T,M}(P, U)$ can be factored identically to the point additions

$$e_{T,M}(P, U) = e_{T,M}(P, U_0) \cdot \prod_{k=1}^{m} \prod_{i \in S^{(k)}} e_{T,M}\left(P, \left[\frac{M}{\ell_i^{e_i}} \right] U'_k \right).$$

It remains to precisely map each term in this factorization to the internal computations of Algorithm 5. The initial pairing $e_{T,M}(P, U_0)$ corresponds exactly to the value assigned to e in Line 2. We now demonstrate that the update factor f'_i computed in Lines 12 strictly equals the incremental pairing factor $e_{T,M}\left(P, \left[\frac{M}{\ell_i^{e_i}}\right] U'_k\right)$.

Initially, before the `while` loop, Line 8 updates the base point P as $\left[\frac{M}{L_1}\right] P = \left[\frac{M}{\prod_{i \in I^{(1)}} \ell_i^{e_i}}\right] P$. Within every $(k-1)$-th iteration ($k \ge 2$) of the `while` loop, each time an index i is resolved, Line 15 updates this point as $P \leftarrow [\ell_i^{e_i}]P$. Consequently, at the end of this iteration, the accumulated base point is exactly $\left[\frac{M}{\prod_{i \in I^{(k-1)} \setminus S^{(k-1)}} \ell_i^{e_i}}\right] P = \left[\frac{M}{L_k}\right] P$, which serves as the starting base point for the k-th iteration. Define $P^{(k)} = \left[\frac{M}{L_k}\right] P$.

Line 11 computes the reduced Tate pairing of order L_k over $P^{(k)}$ and U'_k

$$f' = e_{T,L_k}(P^{(k)}, U'_k) = e_{T,L_k}\left(\left[\frac{M}{L_k} \right] P, U'_k \right)$$

during the k-th iteration. Subsequently, Line 12 handles the exponentiation required for index i by raising f' to the power of $\frac{L_k}{\ell_i^{e_i}}$. By the bilinearity of the reduced Tate pairing and Eq. (2), we obtain

$$f'_i = (f')^{\frac{L_k}{\ell_i^{e_i}}} = e_{T,L_k}\left(\left[\frac{M}{L_k} \right] P, U'_k \right)^{\frac{L_k}{\ell_i^{e_i}}} = e_{T,M}\left(\left[\frac{M}{L_k} \right] P, U'_k \right)^{\frac{L_k}{\ell_i^{e_i}}} = e_{T,M}\left(P, \left[\frac{M}{\ell_i^{e_i}} \right] U'_k \right).$$

This completes the logical link: the mathematical multiplier f'_i explicitly computed by the algorithm is exactly the corresponding component in our bilinear expansion of $e_{T,M}(P, U)$. Because e iteratively accumulates these f'_i factors (Line 15) onto the base value $e_{T,M}(P, U_0)$, the final returned e matches $e_{T,M}(P, U)$ in its entirety. We complete the proof of this proposition. □

Remark 4. In Lines 2, 11 and 12 of Algorithm 5, the final exponentiation of the reduced Tate pairing and the powers of elements in μ_{p+1} can be accelerated using the techniques of Lucas sequences mentioned in [LLC+24].

Compared to Method I, Algorithm 5 exhibits the following advantages: (1) Requiring fewer scalar multiplications than the naive approach (Algorithm 20 in Appendix A.7) for the generation of U; (2) The pairing $e_{T,M}(P,U)$ can be computed concurrently with the generation of the torsion point U, requiring only a few additional operations; (3) The generated torsion point U on the first oriented curve E_1 can be reused to recover Q_2 as mentioned in Section 4.1, thereby avoiding regenerating a torsion point R_1 that forms a basis with $P_1 \in E_1[M,\sigma]$; (4) The pairing $e_{T,M}(P,U)$ is a generator of μ_M and can be used as the base in the phase of solving discrete logarithm, simplifying the process of the discrete logarithm, and preventing the transmission of the extra labels.

After deriving U and $e_{T,M}(P,U)$, the compressing party can apply Algorithm 3 to generate V such that $\langle U,V\rangle = E[M]$, and compute the reduce Tate pairing $e_{T,M}(P,V)$. Then P can be represented as $P=[a]U+[b]V, a,b \in \mathbb{Z}/M\mathbb{Z}$.

Discrete logarithm. Let $g = e_{T,M}(P,U)$ and $h = e_{T,M}(P,V)$. Based on the above analysis, g is a generator of μ_M i.e., a is invertible in $\mathbb{Z}/M\mathbb{Z}$. Unlike Method I, we can directly choose g as the base and compute the discrete logarithm $S = \log_g h \equiv a^{-1}b \bmod M$ using the Pohlig-Hellman algorithm, avoiding the transmission of extra labels. Consequently, the point $P := P_1$ can be encoded by $S \in \mathbb{Z}/M\mathbb{Z}$.

Similarly, the coordinates $S_2,\cdots,S_r \in \mathbb{Z}/M\mathbb{Z}$ for compressing $P_2,\cdots,P_r$, can be similarly derived by executing the above procedures. Thus, the public key is

$$\mathsf{pk} = (E_1, S, \cdots, S_r)\,,$$

with a bit-length of about $(r+2)\log_2(p)$, which is smaller than that of Method I. Algorithm 6 summarizes the whole process for compressing P by Method II.

Due to its superior efficiency and lower transmission memory footprint compared to Method I, Method II is considered for implementation. The performance of our implementation on x64 and AVX-512 is presented in Section 6.

Algorithm 6 CompressP_MethodII: Compress P using Method II

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I=\{1,\cdots,n\}$, the cofactor $(p+1)/M$, a point P of order M;
Ensure: A coordinate $S \in \mathbb{Z}/M\mathbb{Z}$.
1: $U, g \leftarrow$ GenerateU&TatePairingU$(E, I, (p+1)/M, P)$ ▷ Algorithm 5
2: $V \leftarrow$ GenerateV$(E, I, (p+1)/M, U)$ ▷ Algorithm 3
3: $h \leftarrow e_{T,M}(P,V)$
4: $S \leftarrow \log_g h$
5: **return** S

The procedure of decompression. Consider the decompression of $P := P_1$ with respect to S for simplicity. The decompressing party generates the same basis U,V as the compressing party using the index sets I_U and I_V from the generations of U and V, and then computes $K = U + [S]V = U + [a^{-1}b \bmod M]V$, which generates the same kernel as $P=[a]U+[b]V$. The points $P_2,\cdots,P_r$ can also be decompressed in the same manner.

4.2.2 The Encoding for P_{sj} and P_{tj}: The Situation Where the Twist Trick is Employed

We now consider the case where the twist trick is utilized. It follows from Remark 2 that it satisfies $\gcd\left(\frac{p^2-1}{M_s}, M_s\right) = 1$ and $\gcd\left(\frac{p-1}{M_t}, M_t\right) \neq 1$ in the CSIDH-LDO setting. By Proposition 1, we can use the reduced Tate pairing for the torsion basis generation within the compression of $P_{sj},\ j=1,\cdots,r$. Thus, Methods I and II discussed in Section 4.2.1 can be directly employed, with a public-key size of $(r+2)\log_2(p)+rn$ and $(r+2)\log_2(p)$, respectively.

As for encoding $P_{tj} \in E[M_t],\ j = 1, \cdots, r$, we should apply Proposition 2 and use the Weil pairing in the torsion basis generation phase according to Remark 2. We focus on compressing P_t using Method II as it is faster and more compact than Method I. The corresponding framework is as follows.

(1) Generate $U_{tj},\ j = 1, \cdots, r$ such that $\langle P_{tj}, U_{tj} \rangle = E_j[M_t]$ and simultaneously compute the Weil pairing $g_{tj} = e_{W,M_t}(P_{tj}, U_{tj})$;

(2) Generate $V_{tj},\ j = 1, \cdots, r$ such that $\langle U_{tj}, V_{tj} \rangle = E_j[M_t]$ and then compute the Weil pairing $h_{tj} = e_{W,M_t}(P_{tj}, V_{tj})$;

(3) Solve the discrete logarithm $S_{tj} = \log_{g_{tj}} h_{tj},\ j = 1, \cdots, r$.

Hence, the total bit-length of the public key $\mathsf{pk} = (E_1, S_{s1}, \cdots, S_{sr}, S_{t1}, \cdots, S_{tr})$ compressed by Method II is about $(2r + 2)\log_2(p)$ when using the twist trick, which is smaller than that in Section 4.1 (with a bit-length $(4r + 2)\log_2(p)$). Moreover, for the encoding of each P_{sj} and P_{tj}, one pairing and one discrete logarithm computation can be saved compared to the approach described in Section 4.1.

The related algorithms for the compression of P_{tj} using Method II are analogous to Algorithms 5 and 6, with the only difference being the use of the Weil pairings. We only present the procedure for generating $U_t := U_{t1}$ such that $\langle P_t, U_t \rangle = E[M_t]$, $P_t := P_{t1}$, $E := E_1$ and computing the Weil pairing $e_{W,M_t}(P_t, U_t)$ for brevity, which is summarized in Algorithm 7.

Algorithm 7 GenerateUt&WeilPairingUt: Generate a point U_t of order $M_t = \prod_{i=1}^{n} {\ell'_i}^{e'_i}$ such that $\langle P_t, U_t \rangle = E[M_t]$ and derive the Weil pairing $e_{W,M_t}(P_t, U_t)$

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I = \{1, \cdots, n\}$, the cofactor $(p-1)/M_t$, a point P_t of order M_t;
Ensure: A point U_t such that $\langle U_t, P_t \rangle = E[M_t]$ and the Weil pairing $e_{W,M_t}(P_t, U_t)$.

Generate a point U using Elligator 2
$U \leftarrow [(p-1)/M_t]\,U,\ e \leftarrow e_{W,M_t}(P_t, U),\ \{f_i \mid i \in I\} \leftarrow \left\{ f^{M_t/{\ell'_i}^{e'_i}} \mid i \in I \right\},\ I_U \leftarrow [\,]$
for $i \in I$ **do**
  if $\mathrm{ord}(f_i) < {\ell'_i}^{e'_i}$ **then**
    $I_U \leftarrow I_U \cup \{i\}$
  end if
end for
$P \leftarrow \left[M_t / \prod_{i \in I_U} {\ell'_i}^{e'_i} \right] P_t$
while $I_U \neq \emptyset$ **do**
  Generate a point U' using Elligator 2
  $L \leftarrow \prod_{i \in I_U} {\ell'_i}^{e'_i},\ U' \leftarrow [(p-1)/L]U',\ \{f'_i \mid i \in I_U\} \leftarrow \left\{ e_{W,L}(P, U')^{L/{\ell'_i}^{e'_i}} \mid i \in I_U \right\}$
  for $i \in I_U$ **do**
    if $\mathrm{ord}(f'_i) = {\ell'_i}^{e'_i}$ **then**
      $I_U \leftarrow I_U \setminus \{i\},\ U \leftarrow U + \left[L/{\ell'_i}^{e'_i} \right] U',\ P \leftarrow [\ell'_i]\,P,\ e \leftarrow e \cdot f'_i$ ▷ Update e
    end if
  end for
end while
return U, e ▷ $e = e_{W,M_t}(P_t, U_t)$

Proposition 4. *Algorithm 7 correctly computes the Weil pairing* $e_{W,M}(P_t, U_t)$.

Proof. The proof is similar to Proposition 3 by replacing the Tate pairing with the Weil pairing, invoking the corresponding properties (Eq. (8) in Proposition 2). □

Remark 5. Thanks to the work of Pope *et al.* [PRR+25], we can employ **cubical arithmetic**, which offers improved efficiency over Miller's algorithm for pairing computations in isogeny-based cryptography, to evaluate all the required Tate and Weil pairings within the compression procedure. For brevity, we omit a detailed discussion of this optimization and instead incorporate it into our implementation.

5 Security Considerations

This section evaluates the security of our constant-time implementation and the compressed CSIDH-LDO. We first discuss the practical implementation details, and then provide the theoretical security analysis of the compressed CSIDH-LDO.

5.1 Constant-Time Execution and Implementation Details

This subsection details the implementation strategies used to ensure the practical security of our scheme. Specifically, we discuss our approach to constant-time execution, the use of the GMP library for public-key operations, and memory security during key generation.

Constant-Time Execution, Batching, and Memory Access. Our implementations for x64, AVX-512F, and AVX-512IFMA architectures are constant-time with respect to secret data, protecting against timing and branch prediction attacks. All underlying field arithmetic executes in constant time, and secret-dependent branches are replaced with constant-time logical operations or vector masking. For our high-performance SIMD batching, parallel instances operate on isolated vector lanes using data-independent instructions, strictly preventing cross-lane leakage. During scalar multiplication via precomputed differential addition chains, the operation sequence and table lookups are entirely independent of the private key. Consequently, memory access patterns do not depend on secret data, preventing cache-timing vulnerabilities.

Public Key Compression and GMP Usage. Public-key compression and decompression utilize the GNU Multiple Precision Arithmetic Library (GMP). While GMP operations and their internal table lookups are variable-time, they operate exclusively on public keys. Since no secret information is processed during these phases, this introduces no timing side-channel leakage.

Key Generation and Memory Security. Private keys are generated using system randomness via `/dev/urandom`. To protect against potential memory-dumping attacks, dynamically allocated private keys are securely zeroed out before deallocation. In summary, by adhering to constant-time principles for all secret data and restricting variable-time operations to public inputs, our implementation maintains the security assumptions of the underlying primitives without introducing side-channel vulnerabilities.

5.2 The Security of Compressed CSIDH-LDO

In this subsection, we explain that our compression methods do not affect the security of CSIDH-LDO. For simplicity, we only consider the standard case. The deduction for the situation where the twist trick is leveraged is similar. The security of CSIDH-LDO is based on the hardness of the following vectorization problem:

Problem 1. *Given two full kernel representations* $\{(E_j, P_j, Q_j) \mid j = 1, \cdots, r\}$ *and* $(E'_j, P'_j, Q'_j),\ j = 1, \cdots, r$, *compute a vector* $(s_1, \cdots, s_n),\ 1 \leq s_i \leq Be_i, B \in \mathbb{Z}$ *such that* $E'_j = [\prod_{i=1}^{n} [\ell_i^{s_i}] * E_j$.

The most significant quantum attack of Problem 1 is Kuperberg's algorithm, which achieves subexponential time complexity in the class number $\#\mathrm{Cl}(\mathcal{O}) \approx |\mathrm{Disc}(\mathcal{O})|^{\frac{1}{2}}$ [Hou26]. For classical attacks, the complexity is primarily determined by the characteristic p. In our compressed CSIDH-LDO protocol, the parameters (including the discriminant $\mathrm{Disc}(\mathcal{O})$, the characteristic p), as well as the key generation and key exchange procedures, are adopted from [Hou26], with the only modification being the encoding of the public key. In the original CSIDH-LDO, the transmitted public key is $(E_j, P_j, Q_j),\ j = 1, \cdots, r$. In contrast, in our compressed protocol, the public key is given by $(E_1, (S_1, label_1) \cdots, (S_r, label_r))$ or $(E_1, S_1 \cdots S_r)$, depending on whether Method I or II is used. Consequently, the security of the compressed CSIDH-LDO protocol reduces to the hardness of Problem 2.

Problem 2. *Given two full kernel representations* $\{(E_j, P_j, Q_j) \mid j = 1, \cdots, r\}$ *and* $\{E'_1, (S_1, label_1), \cdots, (S_r, label_r)\}$ *(or* $\{E'_1, S_1, , \cdots, S_r\}$ *using Method II), compute a vector* $(s_1, \cdots, s_n),\ 1 \leq s_i \leq Be_i,\ B \in \mathbb{Z}$ *such that* $E'_j = [\prod_{i=1}^{n} [_i^{s_i}] * E_j$.

Since our compressed CSIDH-LDO does not change the discriminant of the quadratic order $\mathcal{O}$, the characteristic p, nor does it leak any information about the private key, it is secure if Problems 1 and 2 are equivalent. We now establish the following proposition.

Proposition 5. *Problems 1 and 2 are equivalent.*

Proof. The kernel representation $\{(E'_j, P'_j, Q'_j),\ j = 1, \cdots, r\}$ can be compressed to $\{E'_1, (S_1, label_1), \cdots, (S_r, label_r)\}$ or $\{E'_1, S_1, , \cdots, S_r\}$ using Methods I and II, respectively. On the other hand, from Lemma 1 and the discussion in Section 4.2.1 it follows that $P_j,\ j = 1, \cdots, r$ can be decompressed from $\{(S_1, label_1), \cdots, (S_r, label_r)\}$ or $\{S_1, , \cdots, S_r\}$ applying Method I or II. Additionally, Q_j and E_j can be successively recovered from P_j and E_1 by executing the group actions according to Section 4.1. Hence, there exists a mutual reduction between Problems 1 and 2. This ends the proof of this proposition. □

6 Experimental Results

In this section, we present the experimental results of our implementation, which combines algorithmic optimizations with parallel execution. Additionally, the results for public-key compression of CSIDH-LDO are also provided.

6.1 Experimental Setup

To evaluate the performance gains achieved by our vectorization techniques, we benchmarked our software on an 11th Gen Intel Core i7-1185G7 processor (Tiger Lake) clocked at 3.00 GHz. All source code was compiled using GCC version 11.4.0. To ensure consistent and reproducible timing results, both Intel Turbo Boost and Hyper-Threading technologies were disabled during the performance measurements.

Our implementation builds upon a proof-of-concept SageMath code presented in [Hou26]. Thanks to the open-source code in [CFG+21] and [CFGR22] available at https://github.com/hchengv/avxcsidh and https://github.com/hchengv/avxsike, respectively, we further establish a fast parallel implementation of the finite field arithmetic carefully adapted and modified to suit our setting. To evaluate the practical efficiency of our proposed parallelization strategies, we developed and benchmarked three distinct software implementations:

(1) **Baseline (x64 Assembly):** A scalar C implementation where the underlying finite field arithmetic is optimized using standard x64 assembly. This serves as the serial reference point for our benchmarks.

(2) **AVX-512F:** A parallel implementation leveraging the AVX-512 Foundation instruction set. This version utilizes 512-bit vector registers for SIMD parallelism but relies on standard integer arithmetic instructions (e.g., `vpmuludq`) without the IFMA extension.

(3) **AVX-512IFMA:** A highly optimized parallel implementation incorporating the AVX-512 Integer Fused Multiply-Add (IFMA) extension. This version exploits the specialized `vpmadd52` instructions to significantly accelerate the large integer arithmetic within the finite field operations.

Each implementation is instantiated with two parameter configurations targeting NIST level 1 security (as detailed in Appendix A.3): a conservative choice with $r = 13$ (yielding a 4096-bit discriminant) and an aggressive choice with $r = 7$ (yielding a 2048-bit discriminant).

6.2 Efficiency Comparison across Instruction Set Architectures

In our implementation, the complete key exchange protocol requires each party (Alice and Bob) to perform two rounds of `group_action` executions, with each round comprising $B = 35$ calls: one for public-key generation and the other for shared secret derivation. Each `group_action`, in turn, consists of $r = 7$ or $r = 13$ `action_home_base` operations followed by one isomorphism correction, whose overhead is negligible.

Table 5: Efficiency of `action_home_base` on x64, AVX-512F and IFMA

ISA	Implementation	Latency†	#Inst.	Speedup
x64	1-way	32.69×10^6	1	1.00×
AVX-512F	$(1\times1\times2\times4\times1)$-way‡	14.86×10^6	1	2.20×
	$(2\times1\times2\times2\times1)$-way§	20.56×10^6	2	3.18×
	$(4\times1\times2\times1\times1)$-way	37.10×10^6	4	3.52×
	$(8\times1\times1\times1\times1)$-way	72.82×10^6	8	3.59×
IFMA	$(1\times1\times2\times4\times1)$-way‡	7.65×10^6	1	4.27×
	$(2\times1\times2\times2\times1)$-way§	9.95×10^6	2	6.57×
	$(4\times1\times2\times1\times1)$-way	17.18×10^6	4	7.61×
	$(8\times1\times1\times1\times1)$-way	33.34×10^6	8	7.84×

† The latency, reported in CPU clock cycles, is the median of 50 data points, where each point is the average cycle count over 100 executions.
§ The $(2\times1\times2\times2\times1)$-way implementation is with $(4\times1\times1\times1)$-way `xEVAL_tra`.
‡ The $(1\times1\times2\times4\times1)$-way implementation is with $(4\times2\times1\times1)$-way `xEVAL_tra`.

We first analyze the efficiency of `action_home_base`. As shown in Table 5, 8-way parallelism achieves peak efficiency on both AVX-512F and IFMA by completely avoiding the complexities of internal function- and arithmetic-level parallelism. The 4-way approach performs nearly on par with the 8-way. Its core operations within `action_home_base`, including the scalar multiplication and isogeny computation using V'elu's formula, are well-suited to 2-way internal parallelism at $\mathbb{F}_{p^2}$-level (see Appendix A.6), fully leveraging concurrent computation.

The 2-way and 1-way implementations exhibit lower performance. Except for the `xEVAL_tra` function, they rely on $(4\times2\times1)$-way and $(2\times4\times1)$-way parallelization schemes for $\mathbb{F}_{p^2}$-operations, respectively. While these internal configurations are generally less efficient (see Table 2), the 2-way approach outperforms the 1-way due to the relative superiority of the $(4\times2\times1)$ scheme. Nevertheless, the 1-way scheme remains unavoidable in our settings because $r = 13$ and $r = 7$ are odd.

Two parallelization strategies for `group_action`, the dummy-based ($(8+8)$- or 8-way) and dummy-free ($(8+4+1)$- or $(4+2+1)$-way) approaches, are introduced in Section 3.4.1. The overall efficiency of `group_action` closely aligns with that of `action_home_base`.

Table 6: Efficiency of a single `group_action` on x64, AVX-512F and IFMA for $r = 13$ and $r = 7$ ($B = 35$).

ISA	$r = 13$			$r = 7$		
	Impl.	**Latency**[†]	**Sp.**	**Impl.**	**Latency**[†]	**Sp.**
x64	1-way	$14{,}856.59\times10^6$	1.00×	1-way	$7{,}999.43\times10^6$	1.00×
AVX-512F	$(8+4+1)$-way	$4{,}370.48\times10^6$	**3.40×**	$(4+2+1)$-way	$2{,}541.10\times10^6$	**3.15×**
	$(8+8)$-way	$5{,}114.65\times10^6$	2.90×	8-way	$2{,}549.17\times10^6$	3.14×
IFMA	$(8+4+1)$-way	$2{,}033.91\times10^6$	**7.30×**	$(4+2+1)$-way	$1{,}208.86\times10^6$	6.62×
	$(8+8)$-way	$2{,}325.51\times10^6$	6.39×	8-way	$1{,}162.05\times10^6$	**6.88×**

[†] The latency, reported in CPU cycles, is the median of 20 data points, where each point is the average cycle count over 50 executions.

As shown in Table 6, for $r = 13$, the $(8+4+1)$ strategy is notably more efficient than the $(8+8)$ strategy, achieving speedups of 7.30× on IFMA and 3.40× on AVX-512F. Although the pure 8-way `action_home_base` offers the highest efficiency, the $(8+8)$ execution requires computing three redundant instances. This overhead significantly outweighs the benefits of avoiding internal parallelism.

For $r = 7$, the $(4+2+1)$ and 8-way strategies show comparable overall performance. Since the 8-way approach only executes a single redundant instance, this minor overhead is effectively offset by the inherent inefficiencies of the internal parallelism required in the $(4+2+1)$ approach. However, their relative superiority is architecture-dependent: the $(4+2+1)$ strategy is slightly faster on AVX-512F but slower on IFMA. This divergence occurs because the $(4+2+1)$ approach relies on less efficient $\mathbb{F}_{p^2}$-arithmetic. As Table 2 shows, the performance drop from the optimal $(8\times1\times1)$-way $\mathbb{F}_{p^2}$-arithmetic is much worse on IFMA than on AVX-512F.

6.3 Comparison with State-of-the-Art Protocols

Table 7 compares the efficiency of CSIDH-LDO against state-of-the-art protocols. On the x64 architecture, CSIDH-LDO outperforms dCSIDH, consuming 27.5% and 77.7% of the CPU cycles required by dCSIDH for the 4096-bit and 2048-bit discriminants, respectively. On the x64 baseline, dCTIDH-2048 evaluates approximately 11.8% faster than the sequential CSIDH-LDO ($r = 7$). However, when transitioning to SIMD environments, CSIDH-LDO demonstrates a high degree of scalability. For context, Cheng *et al.* [CFG+21] introduced low-latency implementations of CSIDH-512 utilizing IFMA instructions. Their optimized OAYT-style and dummy-free-style group actions achieved parallel speedups of 1.54× and 1.71×, respectively, over their x64 baselines. In contrast, our parallelized CSIDH-LDO scales up to 3.40× (AVX-512F) and 7.30× (IFMA), demonstrating the structural suitability of our optimization approaches for modern vectorized platforms.

6.4 Performance of the Public-Key Compression

In this subsection, we provide a concrete comparison of the transmitted public-key sizes, measured in KiB (kibibytes), together with the computational performance in terms of CPU clock cycles, for both the original and the compressed CSIDH-LDO with the twist trick applied. Figure 5 illustrates the public-key sizes for the uncompressed and compressed CSIDH-LDO at "conservative" NIST level 1 under Methods I and II. On the x-axis, "Ps" and "Pt" represent the aggregated sizes of $\{P_{sj}\}_{j=1}^{r}$ and $\{P_{tj}\}_{j=1}^{r}$, respectively. The label

Table 7: Efficiency comparison of isogeny-based protocols at different security levels.

Scheme	Source	ISA	Latency*	Ratio†
Conservative NIST level 1 (4096-bit discriminant)				
dCSIDH-4096	[CCSCD+24]	x64	215,965.70×10^6	0.28×
CSIDH-LDO ($r = 13$)	This work	x64	59,426.36×10^6	1.00×
CSIDH-LDO ($r = 13$)	This work	AVX-512F§	17,474.78×10^6	3.40×
CSIDH-LDO ($r = 13$)	This work	IFMA§	8,137.16×10^6	7.30×
Aggressive NIST level 1 (2048-bit discriminant)				
dCSIDH-2048	[CCSCD+24]	x64	41,178.80×10^6	0.78×
dCTIDH-2048-194	[CHMR25]	x64	28,230.60×10^6	1.13×
CSIDH-LDO ($r = 7$)	This work	x64	31,997.71×10^6	1.00×
CSIDH-LDO ($r = 7$)	This work	AVX-512F§	10,163.15×10^6	3.15×
CSIDH-LDO ($r = 7$)	This work	IFMA§	4,653.40×10^6	6.88×

* Latency (avg. CPU cycles over 100 runs) covers the full protocol (two key generation processes and two final group actions for the shared key), excluding public key validation.
† The ratio is calculated relative to the x64 implementation of this work within the same security level.
§ AVX-512F and IFMA use optimal configurations: $(8+4+1)$-way for $r = 13$; for $r = 7$, $(4+2+1)$-way (AVX-512F) and 8-way (IFMA).

"Total" denotes the overall public-key size, which consists of "Ps", "Pt", and all remaining public-key components. These include the coefficients of $\{E_j\}_{j=1}^r$, the points $\{Q_{sj}\}_{j=1}^r$ and $\{Q_{tj}\}_{j=1}^r$ in the uncompressed CSIDH-LDO setting, and, in the compressed variant, only the coefficient of E_1, as described in Section 4.1. As shown in Figure 5, Methods I

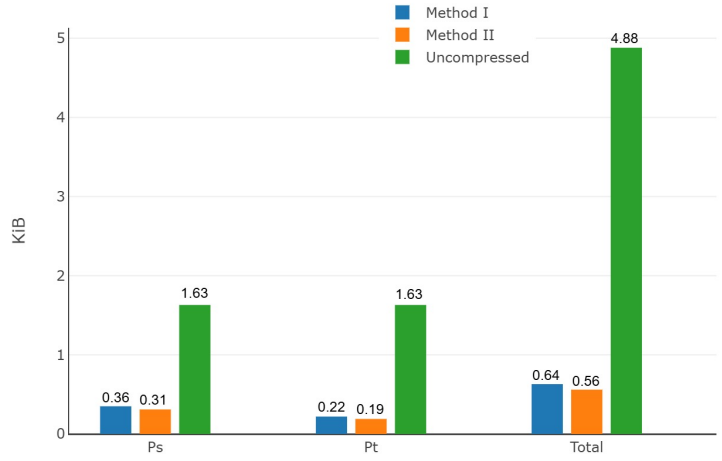


Figure 5: The public-key sizes (in KiB) of the original and compressed CSIDH-LDO at "conservative" NIST level 1 using Methods I and II.

and II achieve substantial reductions compared to the original protocol. Specifically, the total public-key sizes for the compressed protocols using Methods I and II amount to only 7.9% and 6.9% of the uncompressed ones, respectively.

Table 8 illustrates the efficiency performance for the public-key compression and decompression of CSIDH-LDO. As mentioned in Section 4.2.1, we consider implementing Method II in C on platforms x64 and AVX512 for CSIDH-LDO at the "aggressive" ($r = 7$)

Table 8: Efficiency of the compressed CSIDH-LDO protocol on x64, AVX-512F and IFMA for $r = 13$ and $r = 7$.

ISA	Operations	$r = 13$		$r = 7$	
		Latency[†]	**Ratio**[‡]	**Latency**[†]	**Ratio**[‡]
x64	`compressed CSIDH-LDO`	$61{,}043.57\times10^6$	100.00%	$32{,}880.98\times10^6$	100.00%
	2× `public-key compression`	625.10×10^6	1.02%	342.49×10^6	1.04%
	2× `public-key decompression`	992.11×10^6	1.63%	540.78×10^6	1.64%
AVX-512F	`compressed CSIDH-LDO`[§]	$18{,}535.21\times10^6$	100.00%	$10{,}719.63\times10^6$	100.00%
	2× `public-key compression`	532.30×10^6	2.87%	277.61×10^6	2.59%
	2× `public-key decompression`	521.00×10^6	2.81%	277.64×10^6	2.59%
IFMA	`compressed CSIDH-LDO`[§]	$8{,}946.61\times10^6$	100.00%	$5{,}083.67\times10^6$	100.00%
	2× `public-key compression`	473.83×10^6	5.30%	251.31×10^6	4.94%
	2× `public-key decompression`	337.16×10^6	3.77%	184.17×10^6	3.62%

[†] Reported in average CPU clock cycles over 100 executions, the latency of `compressed CSIDH-LDO` covers the full protocol: 4 group actions (including 2 for key generation) and 2 public-key compressions & decompressions.
[‡] The ratio is calculated as "latency of the operation/latency of `compressed CSIDH-LDO`" within the same r.
[§] For AVX-512F and IFMA, public-key compression & decompression are optimized using their respective architectures. Furthermore, the underlying group actions adopt the most efficient configurations: $(8 + 4 + 1)$-way for $r = 13$; for $r = 7$, $(4 + 2 + 1)$-way (AVX-512F) and 8-way (IFMA).

and "conservative" ($r = 13$) NIST level 1. Since the public-key compression does not involve any information related to the private key, this phase is not implemented in constant time. The experimental results show that the public-key compression and decompression implemented by x64, AVX-512F, and IFMA account for only 2.65% (resp. 2.68%), 5.68% (resp. 5.18%), and 9.07% (resp. 8.56%) of the compressed CSIDH-LDO protocol at the "conservative" (resp. "aggressive") NIST level 1. In conclusion, from Figure 5 and Table 8 it follows that our effective compression approach achieves a significant reduction in public-key size, incurring only a marginal increase in computational overhead compared to the original version.

7 Conclusions

In this paper, we presented optimized, constant-time implementations and a public-key compression framework for the CSIDH-LDO protocol. To evaluate its performance on modern processors equipped with vector extensions, we targeted two NIST level 1 parameter sets ($r = 13$ and $r = 7$). By addressing the structural mismatch between the odd number of group actions and 8-lane SIMD registers, we designed and compared parallel scheduling strategies. Our dummy-free approach utilizes internal parallelism to fully saturate 512-bit vectors, avoiding the redundant computations inherent to dummy-based methods.

Our benchmarking results confirm the parallel scalability of this approach. Compared to our x64 baseline, the parallel implementations achieve speedups of up to 3.40× (AVX-512F) and 7.30× (AVX-512IFMA). In horizontal comparisons on the x64 architecture, CSIDH-LDO requires fewer CPU cycles than dCSIDH, though it evaluates approximately 11.8% slower than dCTIDH-2048. However, when transitioning to SIMD environments, our implementations exhibit significantly higher parallel speedups than prior vectorized CSIDH-like protocols, demonstrating the structural suitability of our approach for modern vectorized platforms.

Furthermore, we proposed public-key compression techniques that reduce the key size to approximately 7.9% and 6.9% of the uncompressed format, effectively mitigating bandwidth requirements. Finally, we provided a dedicated security analysis detailing the constant-time execution mechanisms, memory access safety, and the cryptographic soundness of

the compressed protocol. These optimizations and analyses contribute to the practical evaluation and deployment of isogeny-based cryptography with large discriminants.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (No. 2025YFA1017100), the National Natural Science Foundation of China (No. 12441107), and Guangdong Provincial Key Laboratory of Information Security Technology (No. 2023B1212060026). We deeply appreciate the anonymous reviewers for their insightful comments and suggestions. We also thank Jipeng Zhang for valuable guidance on the security considerations.

A Appendix

This appendix provides supplementary material for the paper, including the specific protocol parameters, related algorithms, and the detailed finite field arithmetic configurations.

A.1 Microarchitectural Details of AVX-512

To evaluate the SIMD parallelization discussed in Section 2.2, we utilize the Intel Advanced Vector Extensions 512 (AVX-512). This architecture expands the vector registers to 512 bits and increases the number of registers to 32 (denoted as `zmm0` through `zmm31`), which helps reduce register pressure [Int23, FHLD19].

AVX-512 Foundation (AVX-512F). This baseline instruction set provides support for parallel arithmetic on 64-bit integers. Instructions such as `vpaddq`, `vpsubq`, and `vpternlogq` perform simultaneous operations across independent lanes. However, standard AVX-512F faces limitations in integer multiplication. Instructions like `vpmuludq` typically multiply 32-bit operands to produce 64-bit results. Constructing wide-integer multiplication (e.g., 64×64-bit) solely using AVX-512F requires complex instruction sequences to manage operand sizes and partial products, leading to suboptimal latency [CFGR22, KG19].

AVX-512 Integer Fused Multiply-Add (AVX-512IFMA). To accelerate large integer multiplication, the AVX-512IFMA extension provides specialized instructions (`vpmadd52luq` and `vpmadd52huq`). These instructions perform fused multiply-add operations on packed unsigned 52-bit integers within 64-bit lanes. For source vectors A and B, and an accumulator C, the operations compute a 104-bit product and add either the lower or upper 52 bits to the destination vector:

$$C[i] \leftarrow C[i] + (A[i] \times B[i])[51:0],\ C[i] \leftarrow C[i] + (A[i] \times B[i])[103:52]$$

where i denotes the index of the 64-bit lane ($0 \le i < 8$). This mechanism significantly improves performance by fusing multiplication and accumulation [GK16].

Target Platform. Our implementations are evaluated on an 11th Generation Intel Core processor (Intel Core i7-1185G7 @ 3.00GHz, Willow Cove microarchitecture). This platform provides full hardware support for both AVX-512F and AVX-512IFMA.

A.2 Detailed Finite Field Arithmetic

Addition and Subtraction Bounds. Montgomery reduction of $a \times b \in [0, pR)$ produces a result in $[0, 2p)$ when $4p < R$. In our implementations, $4p < 2^{257} < R = 2^{29\times 9} = 2^{261}$ for AVX-512F, and $4p < 2^{257} < R = 2^{52\times 5} = 2^{260}$ for IFMA. Thus, operations modulo $2p$ are directly computed as $a + b$ and $a - b + 2p$ (or $4p$), where $a, b \in [0, 2p)$. The Montgomery reduction input range is relaxed to $[0, 64p^2)$ and $[0, 32p^2)$ for AVX-512F and IFMA respectively.

Multiplication Strategies. For AVX-512F (9 limbs), we interleave product-scanning integer arithmetic and Montgomery reduction. For IFMA (5 limbs), we pair product-scanning integer arithmetic with operand-scanning Montgomery reduction. In standard $\mathbb{F}_p$ multiplication, Karatsuba is slower than product-scanning due to the small limb counts. However, for $\mathbb{F}_{p^2}$ multiplication using AVX-512F, employing Karatsuba becomes more efficient because immediate Montgomery reduction is not possible. To maintain compatibility with Montgomery reduction in $\mathbb{F}_{p^2}$ Karatsuba implementations, the intermediate variables ($\bar{s}$ and $\bar{t}$) are strictly bounded by pR.

A.3 Parameters

These are the parameters for the key exchange protocol proposed by Houben [Hou26]. We evaluate two parameter configurations achieving NIST level 1 security: a conservative choice and an aggressive choice. Both configurations share the same underlying field, isogeny degrees, and bound B, but differ in the number of oriented elliptic curves r. The shared parameters are:

$$
\begin{aligned}
p &= 37912568609339855107791674627735323252528738550915436620927168502646994567159,\\
M_s &= 144159291472354504918928513734475353795046106918890003859 15,\\
M_t &= 30534654588481117810928107395993 9823,\\
B &= 35.
\end{aligned}
$$

The specific choices for r are: $r = 13$ (conservative) and $r = 7$ (aggressive).

A.4 Parallelization in $\mathbb{F}_{p^2}$

Algorithms 8-12 are multiplications and squarings in $\mathbb{F}_{p^2}$ with different parallelization approaches.

Algorithm 8 $\mathbb{F}_{p^2}$-multiplication with 1-way parallelization at $\mathbb{F}_p$-level.

Require: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Ensure: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1]i$.

1: $t_1 \leftarrow a_0 + a_1$
2: $t_2 \leftarrow b_0 + b_1$
3: $\bar{t}_1 \leftarrow a_0 \times b_0$
4: $\bar{t}_2 \leftarrow a_1 \times b_1$
5: $\bar{t}_3 \leftarrow t_1 \times t_2$
6: $\bar{t}_3 \leftarrow \bar{t}_3 - \bar{t}_1 - \bar{t}_2$
7: $\bar{t}_1 \leftarrow \bar{t}_1 - \bar{t}_2$
8: $r_0 \leftarrow \bar{t}_1 \bmod p$
9: $r_1 \leftarrow \bar{t}_3 \bmod p$
10: **return** $r = r_0 + r_1 i$

Algorithm 9 $\mathbb{F}_{p^2}$-squaring with 1-way parallelization at $\mathbb{F}_p$-level.

Require: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Ensure: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0 a_1 i$.

1: $t_1 \leftarrow a_0 + a_1$
2: $t_2 \leftarrow a_0 - a_1$
3: $t_3 \leftarrow a_0 + a_0$
4: $r_0 \leftarrow t_1 \times t_2$
5: $r_1 \leftarrow t_3 \times a_1$
6: **return** $r = r_0 + r_1 i$

Algorithm 10 $\mathbb{F}_{p^2}$-multiplication with 2-way parallelization at $\mathbb{F}_p$-level.

Require: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Ensure: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1: $\bar{t}_1 \leftarrow a_0 \times b_0$ $\quad$ $\bar{s}_1 \leftarrow a_0 \times b_1$
2: $\bar{t}_2 \leftarrow a_1 \times b_0$ $\quad$ $\bar{s}_2 \leftarrow a_1 \times b_1$
3: $\bar{t}_3 \leftarrow \bar{t}_1 - \bar{s}_2$ $\quad$ $\bar{s}_3 \leftarrow \bar{t}_2 + \bar{s}_1$
4: $r_0 \leftarrow \bar{t}_3 \bmod p$ $\quad$ $r_1 \leftarrow \bar{s}_3 \bmod p$
5: **return** $r = r_0 + r_1 i$

Algorithm 11 $\mathbb{F}_{p^2}$-squaring with 2-way parallelization at $\mathbb{F}_p$-level.

Require: $\mathbb{F}_{p^2}$ element $a = a_0 + a_1 i$.
Ensure: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 + a_1)(a_0 - a_1) + 2a_0 a_1 i$.

1: $t_1 \leftarrow a_0 + a_1$ $\quad$ $s_1 \leftarrow a_0 + a_0$
2: $t_2 \leftarrow a_0 - a_1$
3: $r_0 \leftarrow t_1 \times t_2$ $\quad$ $r_1 \leftarrow s_1 \times a_1$
4: **return** $r = r_0 + r_1 i$

Algorithm 12 $\mathbb{F}_{p^2}$-multiplication/squaring with 4-way parallelization at $\mathbb{F}_p$-level.

Require: $\mathbb{F}_{p^2}$ elements $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$.
Ensure: $\mathbb{F}_{p^2}$ element $r = r_0 + r_1 i = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.

1: $t_1 \leftarrow a_0 \times b_0$ $\quad$ $s_1 \leftarrow a_0 \times b_1$ $\quad$ $t_2 \leftarrow a_1 \times b_0$ $\quad$ $s_2 \leftarrow a_1 \times b_1$
2: $r_0 \leftarrow t_1 - s_2$ $\quad$ $r_1 \leftarrow t_2 + s_1$
3: **return** $r = r_0 + r_1 i$

A.5 2-Way Parallelization of `xDBL` and `xADD` at $\mathbb{F}_{p^2}$-Level

Algorithms 13 and 14 are employed in scalar multiplication for point doubling and differential addition, respectively.

Algorithm 13 `xDBL`: XZ-coordinate point doubling with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: $P = (X_P : Z_P), (A_{24} : C_{24})$
Ensure: $Q = [2]P = (X_Q : Z_Q)$

1: $t_0 \leftarrow X_P + Z_P$ $\quad$ $s_0 \leftarrow X_P - Z_P$
2: $t_1 \leftarrow t_0^2$ $\quad$ $s_1 \leftarrow s_0^2$
3: $t_0 \leftarrow t_1 + s_1$ $\quad$ $s_0 \leftarrow t_1 - s_1$
4: $t_2 \leftarrow C_{24} \times s_1$ $\quad$ $s_2 \leftarrow A_{24} \times s_0$
5: $t_3 \leftarrow t_2 + s_2$
6: $X_Q \leftarrow t_2 \times t_1$ $\quad$ $Z_Q \leftarrow t_3 \times s_0$
7: **return** $Q = (X_Q : Z_Q)$

Algorithm 14 `xADD`: XZ-coordinate point differential addition with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Points $P = (X_P : Z_P)$, $Q = (X_Q : Z_Q)$, and $P - Q = (X_{PQ} : Z_{PQ})$ in XZ-coordinates.
Ensure: $R = (X_R : Z_R) = P + Q$.

1: $s_0 \leftarrow X_P + Z_P$ $\quad$ $t_0 \leftarrow X_P - Z_P$
2: $s_1 \leftarrow X_Q - Z_Q$ $\quad$ $t_1 \leftarrow X_Q + Z_Q$
3: $s_2 \leftarrow s_0 \times s_1$ $\quad$ $t_2 \leftarrow t_0 \times t_1$
4: $s_0 \leftarrow s_2 + t_2$ $\quad$ $t_0 \leftarrow s_2 - t_2$
5: $s_1 \leftarrow s_0^2$ $\quad$ $t_1 \leftarrow t_0^2$
6: $X_R \leftarrow s_1 \times Z_{PQ}$ $\quad$ $Z_R \leftarrow t_1 \times X_{PQ}$
7: **return** $R = (X_R : Z_R)$

A.6 2-Way Parallelization of `KPS_tra`, `xISOG_tra` and `xEVAL_tra` at $\mathbb{F}_{p^2}$-Level

Algorithms 15 and 16 perform point doubling and differential addition on a twisted Edwards curve, used in Algorithm 17 for computing kernel points during isogeny computation.

Algorithm 15 `yDBL`: YT-coordinate point doubling with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Point $P = (X_P : Z_P)$ on a Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$ with parameter $(A+2)/4 = (A_{24} : C_{24})$.
Ensure: $Q = (Y_Q : T_Q)$ in YT-coordinates, which is $[2]P$ on the corresponding twisted Edwards curve.

1: $s_0 \leftarrow X_P^2$ $t_0 \leftarrow Z_P^2$
2: $s_2 \leftarrow s_0$ $t_2 \leftarrow t_0 - s_0$
3: $s_3 \leftarrow t_0$ $t_3 \leftarrow t_2$
4: $s_0 \leftarrow s_2 \times C_{24}$ $t_0 \leftarrow t_2 \times A_{24}$
5: $s_2 \leftarrow s_0$ $t_2 \leftarrow s_0 + t_0$
6: $s_0 \leftarrow s_2 \times s_3$ $t_0 \leftarrow t_2 \times t_3$
7: $X_Q \leftarrow s_0 - t_0$ $Z_Q \leftarrow s_0 + t_0$
8: **return** $Q = (X_Q : Z_Q)$

Algorithm 16 `yADD`: YT-coordinate point differential addition with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Points $P = (Y_P : T_P)$, $Q = (Y_Q : T_Q)$ and $P - Q = (Y_{PQ} : T_{PQ})$ on a twisted Edwards curve.
Ensure: The point $R = P + Q = (Y_R : T_R)$.

1: $t_0 \leftarrow T_{PQ} + Y_{PQ}$ $s_0 \leftarrow T_{PQ} - Y_{PQ}$
2: $t_1 \leftarrow Y_P \times T_Q$ $s_1 \leftarrow Y_Q \times T_P$
3: $t_2 \leftarrow t_1 - s_1$ $s_2 \leftarrow t_1 + s_1$
4: $t_2 \leftarrow t_2^2$ $s_2 \leftarrow s_2^2$
5: $t_1 \leftarrow t_0 \times t_2$ $s_1 \leftarrow s_0 \times s_2$
6: $Y_R \leftarrow s_1 - t_1$ $T_R \leftarrow s_1 + t_1$
7: **return** $R = (Y_R : T_R)$

Algorithm 17 `KPS_tra`: Kernel points computation for isogeny using Vélu's formula with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Isogeny degree ℓ_i, point $P = (X_P : Z_P)$ on a Montgomery curve $E_A : y^2 = x^3 + Ax^2 + x$ with parameter $(A+2)/4 = (A_{24} : C_{24})$.
Ensure: Array K containing the computed kernel points $(Y_K : T_K)$ on the corresponding twisted Edwards curve.

1: $Y_{K[0]} \leftarrow X_P - Z_P$ $T_{K[0]} \leftarrow X_P + Z_P$
2: $K[1] \leftarrow \texttt{yDBL}(K[0], A_{24}, C_{24})$ ▷ Algorithm 15
3: **for** $j = 2$ to $(\ell_i - 1)/2 - 1$ **do**
4: $\quad K[j] \leftarrow \texttt{yADD}(K[j-1], K[0], K[j-2])$ ▷ Algorithm 16
5: **end for**
6: **return** K

Algorithm 18 `xISOG_tra`: Isogeny codomain curve construction using Vélu's formula with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Isogeny degree ℓ_i, array K containing the computed kernel points, the curve parameter $(A+2)/4 = (A_{24} : C_{24})$ of $E_A : y^2 = x^3 + Ax^2 + x$.
Ensure: The codomain curve parameter $(A'+2)/4 = (A'_{24} : C'_{24})$ of $E_{A'} : y^2 = x^3 + A'x^2 + x$, the codomain curve of E_A.

1: $t_0 \leftarrow A_{24} - C_{24}$ $t_1 \leftarrow A_{24}$ ▷ Montgomery to Twisted Edwards
2: $t_0 \leftarrow t_0^\ell$ $t_1 \leftarrow t_1^\ell$
3: $s_0 \leftarrow \prod_{j=0}^{(\ell_i-1)/2-1} Y_{K[j]}$ $s_1 \leftarrow \prod_{j=0}^{(\ell_i-1)/2-1} T_{K[j]}$
4: $s_0 \leftarrow s_0^8$ $s_1 \leftarrow s_1^8$
5: $t_0 \leftarrow s_0 \times t_0$ $t_1 \leftarrow s_1 \times t_1$
6: $C'_{24} \leftarrow t_1 - t_0$ $A'_{24} \leftarrow t_1$ ▷ Twisted Edwards to Montgomery
7: **return** (A'_{24}, C'_{24})

Algorithm 19 `xEVAL_tra`: Isogeny evaluation using Vélu's formula with 2-way parallelization at $\mathbb{F}_{p^2}$-level

Require: Isogeny degree ℓ_i, point $P = (X_P, Z_P)$, array K containing the computed kernel points.
Ensure: The isogeny image point $Q = (X_Q, Z_Q)$.

$s_0 \leftarrow X_P + Z_P$ $\quad$ $s_1 \leftarrow X_P - Z_P$
$t_0 \leftarrow X_{K[0]} \times s_0$ $\quad$ $t_1 \leftarrow Z_{K[0]} \times s_1$
$r_0 \leftarrow t_0 + t_1$ $\quad$ $r_1 \leftarrow t_0 - t_1$
for $j = 1$ to $(\ell_i - 1)/2 - 1$ **do**
  $t_0 \leftarrow X_{K[j]} \times s_0$ $\quad$ $t_1 \leftarrow Z_{K[j]} \times s_1$
  $u_0 \leftarrow t_0 + t_1$ $\quad$ $u_1 \leftarrow t_0 - t_1$
  $r_0 \leftarrow r_0 \times u_0$ $\quad$ $r_1 \leftarrow r_1 \times u_1$
end for
$r_0 \leftarrow r_0^2$ $\quad$ $r_1 \leftarrow r_1^2$
$X_Q \leftarrow r_0 \times X_P$ $\quad$ $Z_Q \leftarrow r_1 \times Z_P$
return $Q = (X_Q, Z_Q)$

A.7 The Naive Approach for Generating a Point U of Order $M = \prod_{i=1}^{n} \ell_i^{e_i}$

Algorithm 20 illustrates the generation of a point U of order $M = \prod_{i=1}^{n} \ell_i^{e_i}$.

Algorithm 20 GenerateU: Generate a point U of order $M = \prod_{i=1}^{n} \ell_i^{e_i}$

Require: A supersingular curve $E/\mathbb{F}_{p^2}$, the index set $I = \{1, \cdots, n\}$, the cofactor $(p+1)/M$;
Ensure: A point $U \in E(\mathbb{F}_{p^2})$ of order M.

Generate a point $U \in E(\mathbb{F}_{p^2})$ using Elligator 2
$U \leftarrow [(p+1)/M]U,\ I_U \leftarrow [\,],\ \{U_i \mid i \in I\} \leftarrow \{[M/\ell_i^{e_i}]\,U \mid i \in I\}$
for $i \in I$ **do**
  if $\mathrm{ord}(U_i) < \ell_i^{e_i}$ **then**
    $I_U \leftarrow I_U \cup \{i\}$
  end if
end for
while $I_U \neq \emptyset$ **do**
  Generate a point $U' \in E(\mathbb{F}_{p^2})$ using Elligator 2
  $U' \leftarrow \left[(p+1)/\prod_{i \in I_U} \ell_i^{e_i}\right] U',\ \{U_i' \mid i \in I_U\} \leftarrow \left\{\left[(\prod_{i \in I_U} \ell_i^{e_i})/\ell_i^{e_i}\right] U' \mid i \in I_U\right\}$
  for $i \in I_U$ **do**
    if $\mathrm{ord}(U_i') = \ell_i^{e_i}$ **then**
      $U \leftarrow U + U_i',\ I_U \leftarrow I_U \setminus \{i\}$
    end if
  end for
end while
return U

References

[ABE+24] Bill Allombert, Jean-François Biasse, Jonathan Komada Eriksen, Péter Kutas, Chris Leonardi, Aurel Page, Renate Scheidler, and Márton Tot Bagi. PEARL-SCALLOP: parameter extension applicable in real-life SCALLOP. *IACR Cryptol. ePrint Arch.*, page 1744, 2024.

[BBC+21] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. Ctidh: faster constant-time csidh. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):351–387, Aug. 2021.

[BCL25] Daniel J. Bernstein, Jolijn Cottaar, and Tanja Lange. Searching for differential addition chains. *Research in Number Theory*, 11(2):45, 2025.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013.

[Bla05] *Advances in Elliptic Curve Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2005.

[CCJR22] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: sublinear vélu quantum-resistant isogeny action with low exponents. *J. Cryptogr. Eng.*, 12(3):349–368, 2022.

[CCSCD+24] Fabio Campos, Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. Optimizations and practicality of high-security CSIDH. *Communications in Cryptology*, 1(1), 2024.

[CDV20] Wouter Castryck, Thomas Decru, and Frederik Vercauteren. Radical isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 493–519, Cham, 2020. Springer International Publishing.

[CFG+21] Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan, and Peter B. Rønne. Batching csidh group actions using avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):618–649, Aug. 2021.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized sike for avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):41–68, Feb. 2022.

[CGT+21] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-Throughput Elliptic Curve Cryptography Using AVX2 Vector Instructions. In Orr Dunkelman, Michael J. Jacobson, Jr., and Colin O'Flynn, editors, *Selected Areas in Cryptography*, pages 698–719, Cham, 2021. Springer International Publishing.

[CHMR25] Fabio Campos, Andreas Hellenbrand, Michael Meyer, and Krijn Reijnders. dCTIDH: Fast & deterministic CTIDH. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):516–541, June 2025.

[CIL+26] Wouter Castryck, Riccardo Invernizzi, Gioella Lorenzon, Jonas Meers, and Frederik Vercauteren. Orient express: Using frobenius to express oriented isogenies. In Daniel Escudero and Ivan Damgård, editors, *Progress in Cryptology – LATINCRYPT 2025*, pages 145–173, Cham, 2026. Springer Nature Switzerland.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. Csidh: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 395–427, Cham, 2018. Springer International Publishing.

[CLP24] Mingjie Chen, Antonin Leroux, and Lorenz Panny. Scallop-hd: Group action from 2-dimensional isogenies. In Qiang Tang and Vanessa Teague, editors, *Public-Key Cryptography – PKC 2024*, pages 190–216, Cham, 2024. Springer Nature Switzerland.

[Cou06] Jean Marc Couveignes. Hard homogeneous spaces. *IACR Cryptol. ePrint Arch.*, page 291, 2006.

[CVCCD+19] Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and faster side-channel protections for csidh. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 173–193, Cham, 2019. Springer International Publishing.

[FFK+23] Luca De Feo, Tako Boris Fouotsa, Péter Kutas, Antonin Leroux, Simon-Philipp Merz, Lorenz Panny, and Benjamin Wesolowski. Scallop: Scaling the csi-fish. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography – PKC 2023*, pages 345–375, Cham, 2023. Springer Nature Switzerland.

[FHLD19] Armando Faz-Hernández, Julio López, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, 2019.

[GK16] Shay Gueron and Vlad Krasnov. Accelerating big integer arithmetic using Intel IFMA extensions. In *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, pages 32–38. IEEE, 2016.

[HLKA20] Aaron Hutchinson, Jason LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of csidh: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 481–501, Cham, 2020. Springer International Publishing.

[Hou26] Marc Houben. Efficient Post-quantum Commutative Group Actions from Orientations of Large Discriminant. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025*, pages 141–173, Singapore, 2026. Springer Nature Singapore.

[Int23] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2023. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[KG19] Dusan Kostic and Shay Gueron. Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 215–218. IEEE, 2019.

[Kup05] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM Journal on Computing*, 35(1):170–188, 2005.

[LLC+24] Kaizhan Lin, Jianming Lin, Shiping Cai, Weize Wang, and Chang-An Zhao. Compressed M-SIDH: an instance of compressed sidh-like schemes with isogenies of highly composite degrees. *Des. Codes Cryptogr.*, 92(6):1823–1843, 2024.

[LLWZ23] Kaizhan Lin, Jianming Lin, Weize Wang, and Chang-An Zhao. Faster public-key compression of SIDH with less memory. *IEEE Trans. Computers*, 72(9):2668–2676, 2023.

[MCR19] Michael Meyer, Fabio Campos, and Steffen Reith. On lions and elligators: An efficient constant-time implementation of csidh. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 307–325, Cham, 2019. Springer International Publishing.

[Mil86] J. S. Milne. *Abelian Varieties*, pages 103–150. Springer New York, New York, NY, 1986.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mon92] Peter L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via lucas chains. *Available at https://cr.yp.to/bib/1992/montgomery-lucas.pdf*, 1992.

[MR18] Michael Meyer and Steffen Reith. A faster way to the csidh. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology – INDOCRYPT 2018*, pages 137–152, Cham, 2018. Springer International Publishing.

[NS22] Kaushik Nath and Palash Sarkar. Efficient 4-way vectorizations of the montgomery ladder. *IEEE Transactions on Computers*, 71(3):712–723, 2022.

[OAL18] Gabriell Orisaka, Diego F Aranha, and Julio López. Finite field arithmetic using AVX-512 for isogeny-based cryptography. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 49–56. SBC, 2018.

[PRR+25] Giacomo Pope, Krijn Reijnders, Damien Robert, Alessandro Sferlazza, and Benjamin Smith. Simpler and faster pairings from the montgomery ladder. *IACR Communications in Cryptology*, 2(2), 2025.

[RS06] Alexander Rostovtsev and Anton Stolbunov. Public-key cryptosystem based on isogenies. *IACR Cryptol. ePrint Arch.*, page 145, 2006.

[ZHZ+24] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6381–6398, Philadelphia, PA, August 2024. USENIX Association.