Streamlining CSIDH: Cost-Effective Strategies for Group Actions Evaluation

Ahmed Zawia(✉) and M. Anwar Hasan

University of Waterloo, Waterloo, ON, Canada
{azawia,ahasan}@uwaterloo.ca

Abstract. In the realm of post-quantum cryptography, the Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) has drawn considerable attention since its proposal at Asiacrypt 2018. This paper introduces a new batching strategy for computing multiple group actions in CSIDH, which is essential for constructing cryptographic schemes such as zero-knowledge proofs and signature schemes. Two variants of the batching strategy are presented in this work, suited to different security requirements and application contexts. In the first scenario, we focus on situations where group actions are public, aiming to reduce CSIDH's computational cost. Using our strategy, we show that computational costs can be significantly reduced, making it a viable option for schemes in which efficiency is critical. This variant reduces the computational requirements of group actions by roughly up to 14% when compared to non-batched action evaluation. The second variant is towards constant-time group actions, and it reduces computational costs while maintaining resilience to side-channel timing attacks. This article also introduces a new constant-time implementation of CSIDH that, when combined with the second variant, reduces the computation costs of secret action sets by roughly up to 8% compared to individual computation using state-of-the-art constant-time algorithms, while the new constant time alone reduces computation by approximately up to 4%.

Keywords: Post-quantum cryptography · Isogeny-based cryptography · CSIDH · Constant-time algorithms · Digital signatures · Zero-knowledge proofs

1 Introduction

With the advent of quantum computing, attention has turned to post-quantum cryptography, a quantum resistant solution. A viable candidate in this arena is the Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) protocol, first proposed by Castryck et al. in Asiacrypt 2018 [10]. While not a contender for post-quantum standardization by National Institute of Standards and Technology, CSIDH has received significant attention within the post-quantum cryptography research community. Building upon the hardness of computing isogenies between supersingular elliptic curves, CSIDH offers a promising foundation for various cryptographic constructions, including non-interactive key-exchanges, digital signatures, zero-knowledge proofs, and oblivious transfers.

However, a critical challenge in the practical deployment of CSIDH-based schemes is the efficiency of computing group actions (via an isogeny). Current implementations, while considered secure, suffer from high computational cost and latency, hindering their applicability in real-world scenarios, as explored in [8]. These efficiency concerns are not merely theoretical but have practical implications, especially when CSIDH is employed in constructions such as signature schemes [1,7,14–16] or zero-knowledge proofs [1,6,15,20]. As an example, a CSIDH-based Sigma protocol, which has a challenge space of three and thus a soundness error of $1/3$, requires repeating its actions in the commit-challenge-response format $\lceil \lambda / \log 3 \rceil$ times to obtain λ bits of security. In these settings, the computational overhead introduced by CSIDH's group actions becomes magnified, potentially limiting their scalability and wider adoption.

So, the question we address in this work is how to strategically compute a set of group actions to minimize the overall computational cost. In particular, this question is addressed in two contexts: one involves computing a set of private group actions (e.g., for secret commitments), which must be implemented in a constant-time manner to prevent information leakage, and the other involves computing a set of public group actions (e.g., for validating responses), which does not require security restrictions. Through our work, we aim to enhance the practical usability of CSIDH-based constructions, making them viable for wider adoption.

Contribution. In this work, we propose a new batching strategy to address the above-mentioned question by reducing the number of field arithmetic operations required. An *action system*, as used in this paper, refers to a collection of cryptographic group actions performed on *a set element*. This article proposes a practical approach to computing action systems applying two variants of our batching strategy. Our first variant, described in Sect. 3, reduces computational costs of public action set by roughly up to 14% compared to element-wise group action evaluation. As described in Sect. 4, we introduce a new constant-time implementation of CSIDH and a second variant that, when combined, reduce computation costs of secret action set by roughly up to 8% compared to the current state-of-the-art constant-time CSIDH. On its own, the new constant-time implementation of CSIDH reduces computation by about 4%. Lastly, the source code for a proof-of-concept implementation of our work is available on https://github.com/ahzawia/Streamlining-CSIDH.

Prior Work. Since the CSIDH proposal in 2018, various suggestions have been made to improve the efficiency of CSIDH operations, specifically the class group action. Most of these proposals focus on enhancing certain computations within the scheme, such as improving isogeny and scalar multiplication through various methods [9,12,18]. Additionally, many improvements aim to secure CSIDH implementations against side-channel attacks, as in [2,8,11,17,19]. However, most techniques designed for constant time implementation increase the cost of CSIDH actions. Although many techniques have been proposed to reduce computational cost, they typically exploit the properties of a single isogeny evaluation. To the best of our knowledge, no work has focused on exploiting the properties of evaluating a set of CSIDH group actions. We note that the method proposed in [13] computes group actions in a Single Instruction/Multiple Data fashion on Intel processors, thereby increases execution throughput.

2 Background and Preliminaries

This section introduces relevant concepts and notations used in this work.

Notations. We use the calligraphic font to denote a finite set (e.g., $\mathcal{K}$). The size of a set $\mathcal{K}$ is represented by $|\mathcal{K}|$. We denote the process of uniformly sampling a random element e from $\mathcal{K}$ by $e \leftarrow_{\$} \mathcal{K}$, whereas the deterministic choice of an element e from $\mathcal{K}$ is referred to by $e \leftarrow \mathcal{K}$. The process of executing an algorithm Alg on a uniformly random distribution is denoted by $a \leftarrow_{\$} \mathsf{Alg}$, where a is the output. On the other hand, the deterministic process of executing an algorithm Alg is denoted by $a \leftarrow \mathsf{Alg}$. The set $\{1, \ldots, n\}$ is denoted by $[n]$ for a positive integer n. The process of sampling the vector $(r_i)_{i \in [n]}$ of size n for a uniform distribution on $\mathcal{K}^n$ is denoted by $\boldsymbol{r} \leftarrow_{\$} \mathcal{K}^n$ such that $r_i \leftarrow_{\$} \mathcal{K}$ for all $i \in [n]$. The 'cmov' is a (conditional move) function that takes three inputs: A, B, and C. It returns A if the condition C is true; otherwise, it returns B.

2.1 CSIDH

Castryck et al. [5] proposed CSIDH as an efficient commutative group action for constructing a post-quantum cryptographic scheme. CSIDH was initially introduced as a form of the Diffie-Hellman key exchange, leveraging isogeny evaluations between elliptic curves to provide protection against quantum attacks.

Ideal Class Group Action. Let $\mathcal{O}$ be an order of an imaginary quadratic field $\mathbb{Q}(\sqrt{p})$, where p is a large prime. The ideal class group $\mathbf{cl}(\mathcal{O})$ is an abelian group of equivalence classes of invertible fractional ideals $\mathcal{O}$-ideals under multiplication (see [23, Theorem 5.5]). Let $\mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$ be the set of supersingular elliptic curves over a prime finite field $\mathbb{F}_p$ of characteristic p with $\mathcal{O}$. By theory of complex multiplication, the action of $\mathbf{cl}(\mathcal{O})$ on the set $\mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$ defines a group action (i.e., $\star : \mathbf{cl}(\mathcal{O}) \times \mathcal{E}_{\mathbb{F}_p}(\mathcal{O}) \rightarrow \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$ that satisfies a set of properties). Relevant details can be found in [21] and [10, Theorem 7].

Evaluation of an Action. CSIDH operates by performing a group action, computed by an isogeny evaluation, on supersingular elliptic curves in $\mathcal{E}_{\mathbb{F}_p}(\mathcal{O} = \mathbb{Z}[\sqrt{-p}])$, where p is congruent to -1 modulo a small odd prime for all small odd primes in a set $\mathcal{L}$. Essentially, for a given input curve E, the basic operation of CSIDH involves the secret $\mathfrak{a} \in \mathbf{cl}(\mathcal{O})$ acting on E, resulting in $E' = \mathfrak{a} \star E$.

The main efficiency advantage of CSIDH is the use of supersingular curves with an order of $\#E(\mathbb{F}_p) = p + 1$. This allows for an easy construction of $\mathbb{F}_p$-rational subgroups of $\mathrm{E}(\mathbb{F}_p)$ with a small prime order (points of order ℓ_i exist over $\mathbb{F}_p$) by selecting $p = 4\ell_1\ell_2 \cdots \ell_n - 1$ and $\forall \ell_i \in \mathcal{L}$. These small primes

Algorithm 1: CSIDH's original algorithm [10] for class group action

```
Input: A $\in \mathbb{F}_p$ such that $E_A : y^2 = x^3 + Ax^2 + x$ is supersingular, and an
       integer exponent vector $(e_1, \ldots, e_n)$
Output: $B$ such that $E_B : y^2 = x^3 + Bx^2 + x$ is $[\ell_1^{e_1} \cdot \cdots \cdot \ell_n^{e_n}] * E_A$
1  $B \leftarrow A$
2  while some $e_i \neq 0$ do
3      Sample a random $x \in \mathbb{F}_p$
4      $s \leftarrow +1$ if $x^3 + Bx^2 + x$ is square in $\mathbb{F}_p$, else $s \leftarrow -1$
5      $S \leftarrow \{i : e_i \neq 0, \mathrm{sign}(e_i) = s\}$
6      if $S \neq \varnothing$ then
7          $k \leftarrow \prod_{i \in S} \ell_i$
8          $Q \leftarrow [(p+1)/k]T$, where $T$ is the projective point with $x$-coordinate $x$
9          for $i \in S$ do
10             $P \leftarrow [k/\ell_k]Q$          // Point to be used as kernel generator
11             if $P \neq \infty$ then
12                 $(B, Q) \leftarrow \mathrm{Isogeny}(B, P, Q)$
13                 $(k, e_i) \leftarrow (k/\ell_i, e_i - s)$
14 return $B$
```

are known as Elkies primes, where the ideal $\ell_i\mathcal{O}$ splits into $\mathfrak{l}_i = (\ell_i, \pi - 1)$ and $\hat{\mathfrak{l}}_i = (\ell_i, \pi+1)$ for every $\ell_i \in \mathcal{L}$ (i.e., $\ell_i\mathcal{O} = \mathfrak{l}_i\hat{\mathfrak{l}}_i, \forall \ell_i \in \mathcal{L}$) and π being the Frobenius endomorphism. This means that for every $E \in \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$, two actions, namely $\mathfrak{l}_i$ or $\hat{\mathfrak{l}}_i$, can be applied for each $\ell_i \in \mathcal{L}$. Each of these actions can be computed via an isogeny constructed using Vélu's formulas [24]. Specifically, we compute $\mathfrak{l}_i$ via an isogeny with a kernel point $P \in E[\ell_i] \cap \mathsf{ker}(\pi - 1])$, representing a single step in one direction. Similarly, $\hat{\mathfrak{l}}_i$ is computed using $P \in E[\ell_i] \cap \mathsf{ker}(\pi + [1])$ for the opposite direction. Given a sufficiently large set $\mathcal{L}$, we hope that elements of $\mathbf{cl}(\mathcal{O})$ can be expressed as a product of small integral powers of elements of $\mathcal{L}$ (i.e., $\prod_{i=1}^{n} \mathfrak{l}_i^{e_i}$ for $e_i \in \mathbb{Z}$), allowing for efficient computation of the class-group action. Hence, CSIDH action is defined by a vector of integers $(e_i)_{[n]}$ uniformly sampled from a set $\mathcal{K}^n$ (i.e., $e_i \in \mathcal{K} = [-b, b]$).

The supersingular elliptic curve in CSIDH is defined by its Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$, with the Montgomery coefficient A mod p. Let $[l]$ be the multiplication-by l map. Algorithm 1 illustrates the original CSIDH class group action evaluation.

2.2 Constant-Time Implementation of CSIDH

In CSIDH, the secret action $\mathfrak{a}$ is represented by a vector of integers $(e_1, \ldots, e_n)$. For all $i \in [n]$, computing $\mathfrak{l}_i^{e_i}$ requires evaluating a degree-ℓ_i isogeny e_i times, which causes the key to directly influence the execution time of Algorithm 1 (i.e., a *variable time* algorithm). To eliminate this dependency, several constant-time methods and efficient implementations have been proposed [2,8,11,17,19].

Constant Time Algorithm. A constant-time algorithm has an execution time that does not depend on the input, meaning the time required to complete the algorithm remains constant regardless of the input. Hence, a constant time version of Algorithm 1 has an execution time independent of the secret vector $(e_1, \ldots, e_n)$. In our context, one of the most important concepts for achieving constant time is the use of a dummy isogeny.

Dummy Isogenies. To achieve a constant running time, this technique involves performing a fixed number of isogenies of each degree $\ell_i \in \mathcal{L}$ and only updating the results dictated by the private key (i.e., when $e_i \neq 0$). However, if used and unused isogenies are computed in the same loop, additional multiplications are necessary to push the torsion point for the next isogeny evaluation. Meyer et al. [17] discussed the use of tailored dummy isogenies, in which instead of computing an ℓ_i-isogeny and pushing the point P through, $[\ell_i]P$ is computed using the kernel computation of a dummy isogeny with two extra differential additions.

Although the dummy isogenies technique (or dummy computation) helps obscure the magnitude of the secret, it is not entirely sufficient. Certain parts of Algorithm 1 are closely linked to the secret key, such as the number of iterations for computing the scalar in Lines 8 and 10, which can reveal the sign of the secret keys. To address this issue, Meyer et al. [17] suggest changing the secret key space by considering only positive elements (i.e., changing $\mathcal{K} = [-b, b]$ to $\mathcal{K} = [0, 2b]$), which increases the number of dummy isogenies. To enhance efficiency, Onuki et al. [19] proposed using two torsion points (instead of one) in the for-loop and pushing both. This approach reduces the magnitude of the key space (i.e., from $2b$ to b), significantly reducing the number of dummy computations.

Unfortunately, the constant-time version of Algorithm 1 performs worse in terms of execution time compared to the variable-time version. Hence, different techniques to speed up the computations have been proposed, such as splitting isogenies into multiple batches [17], multiplicative strategy [18]. A notable work, CTIDH [2], has introduced a highly optimized constant-time algorithm that reduces the number of field operations, resulting in a faster execution time compared to its counterpart.

CTIDH. CTIDH, introduced by Banegas et al., improves upon previous constant time CSIDH implementation by batching key spaces to achieve constant-time performance.

Batching Primes. In CTIDH, primes ℓ_i are grouped into B batches, each containing consecutive primes, defined by a vector $N = (N_1, \ldots, N_B)$ where $N_i > 0$ and $\sum_{i=1}^{B} N_i = n$. The set of n primes $\{\ell_1, \ldots, \ell_n\}$ are distributed and relabeled among these batches as: $(\ell_1, \ldots, \ell_{N_1}) := (\ell_{1,1}, \ldots, \ell_{1,N_1})$, $(\ell_{N_1+1}, \ldots, \ell_{N_1+N_2}) := (\ell_{2,1}, \ldots, \ell_{2,N_2})$ and so on.

Batching Isogenies. Secret key elements e_k are also relabeled as $e_{i,j}$ representing j^{th} secret element in the i^{th} batch. For the i^{th} batch $(\ell_{i,1}, \ldots, \ell_{i,N_i})$, secret elements $e_{i,j}$ are sampled such that $\sum_{j=1}^{N_i} |e_{i,j}| \leq m_i$, where m_i is batch bound. For $N \in \mathbb{Z}_{>0}^B$ and $m \in \mathbb{Z}_{\geq 0}^B$, the CTIDH key space is defined as follows:

$$\mathcal{K}_{N,m} := \{(e_1, \ldots, e_n) \in \mathbb{Z}^n : \sum_{j=1}^{N_i} |e_{i,j}| \leq m_i \text{ for } 1 \leq i \leq B\}.$$

Constant Time Techniques. CTIDH addresses side-channel timing attacks by computing each isogeny in a batch at the cost of the highest degree isogeny in that batch. The key insight is that isogeny formulas exhibit a "Matryoshka-doll" structure, where, for i^{th} batch, an $\ell_{i,j}$-isogeny can be computed at the cost of an ℓ_{i,N_i}-isogeny ($\ell_{i,j} \leq \ell_{i,N_i}$) by performing additional dummy operations. Consequently, every isogeny computation within the batch incurs the same computational cost, concealing the actual isogeny degree. Therefore, to maintain constant time, we compute m_i isogenies from the batch $(\ell_{i,1}, \ldots, \ell_{i,N_i})$ for all $i \in [B]$. Additional operations, such as constant-time scalar multiplication and kernel point generation success rate, are adjusted to maintain constant-time behavior. These adjustments ensure that no timing information leaks about the secret key. Despite the computational overhead, CTIDH improves the performance of constant-time CSIDH by reducing the number of required isogenies through secret key space modification. Consequently, CTIDH achieves a larger key space with fewer isogenies compared to CSIDH. For instance, a key space of 2^{256} in CSIDH-512 requires 438 isogenies [8], while CTIDH needs only 208. Algorithm 2 outlines the constant-time group action of CTIDH. Similar to Algorithm 1, Algorithm 2 employs the following subroutines:

– $\mathsf{UniformRandomPoints}$ takes as an input a curve coefficient A of $E_A \in \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$, and returns a pair of projective points (T_0, T_1) of an order in $\#E(\mathbb{F}_p)$, where $(T_0, T_1) \in (E[p+1] \cap \mathsf{ker}(\pi - 1]), E[p+1] \cap \mathsf{ker}(\pi + 1]))$.
– $\mathsf{PointAccept}$ takes as input a kernel point P, batch number I, and an index $J \in [N_I]$. It returns 0 if P is a point at infinity and 1 otherwise, with a distribution based only on I.
– $\mathsf{MatryoshkaIsogeny}$ is a constant time isogeny subroutine, with running time depends only on I.

Further, in this work, we define $\mathsf{xMULConstTime}$ for constant-time scalar multiplication, taking torsion point P, scalar r, and batch number I, and returning $[r]P$ in time dependent only on I. In Algorithm 2, it is used in Lines 9, 14, and 20. Conversely, xMUL is a variable-time scalar multiplication algorithm used in Line 8.

3 Public Action Set Evaluation

In this section, we propose a strategy for evaluating a set of public CSIDH actions, aiming to optimize the evaluation of the set. Below, we will use either a vector of integers $\boldsymbol{e}^{(i)}$ of size n or a Gothic font to denote the i^{th} ideal $\mathfrak{e}_i$ from a set.

Algorithm 2: CTIDH constant-time group action.

```
**Parameters**: $(B \in \mathbb{Z}_{>0}) \leq n, N \in \mathbb{Z}^B_{>0}, m \in \mathbb{Z}^B_{\geq 0}$
**Require**: $e \in \mathcal{K}_{N,m}, E_A \in \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$
**Ensure**: $A$ with $E'_A = (\prod_i \ell_i^{e_i}) \star E_A$
$(\mu_1, \ldots, \mu_B) \leftarrow (m_1, \ldots, m_B)$;
while $\mu_1, \ldots, \mu_B \neq (0, \ldots, 0)$ do
    Let $I := \{1 \leq i \leq B : \mu_i > 0\}$ of size $k$, sorted in ascending order;
    for $1 \leq i \leq k$ do
        $J_i \leftarrow$ some $j$ such that $e_{i,j} \neq 0$ if such a $j$ exists, else $j \leftarrow_\$ [N_{I_i}]$;
        $\epsilon_i \leftarrow \mathrm{sign}(e_{I_i,J_i})$;    // 1 if $e_{I_i,J_i} > 0$; 0 if $e_{I_i,J_i} = 0$; o.w. -1;
    $(T_0, T_1) \leftarrow \mathrm{UniformRandomPoints}(A)$;
    $(T_0, T_1) \leftarrow (([r]T_0, [r]T_1)$ where $r = 4\prod_{i \notin I}\prod_{j=1}^{N_i} \ell_{i,j}$;
    $(T_0, T_1) \leftarrow (([r']T_0, [r']T_1)$ where $r' = \prod_{i \in I}\prod_{j=1, j \neq J_i}^{N_i} \ell_{i,j}$;
    $r' \leftarrow \prod_{i \in I} \ell_{i,J_i}$;
    for $j = k$ to 1 do
        $s \leftarrow \mathrm{SignBit}(\epsilon_j)$;    // 1 if $\epsilon_j < 0$; o.w. 0;
        $r' \leftarrow r'/\ell_{I_j,J_j}$;
        $P \leftarrow [r']T_s$;
        $f_j \leftarrow \mathrm{PointAccept}(P, I_j, J_j)$;
        if $f_j = 1$ then
            $(A', (T'_0, T'_1)) \leftarrow \mathsf{MatryoshkaIsogeny}(A, P, (T_0, T_1), I_j, J_j)$;
            if $\epsilon_j \neq 0$ then
                $(A, T_0, T_1) \leftarrow (A', T'_0, T'_1)$;
        $(T_0, T_1) \leftarrow ([\ell_{I_j,J_j}]T_0, [\ell_{I_j,J_j}]T_1)$;
    for $1 \leq i \leq k$ do
        $(\mu_{I_i}, e_{I_i,J_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i,J_i} - f_i\epsilon_i)$;
return $A$;
```

Action System. An execution set is a set of independent CSIDH-actions $\{\boldsymbol{e}^{(j)}\}_{j\in[c]}$ that need to be evaluated to run an application (e.g., a digital signature), where $c \in \mathbb{N}$ is the execution set size. The action system is a subset of the execution set. In particular, an action system is a set of actions $\{\boldsymbol{e}^{(j)}\}_{j\in[c']}$ with the same starting point E_A, where $c' \leq c$ is the system size. The difficulty of an action $\mathfrak{e}_i$ is the absolute sum of all $\boldsymbol{e}^{(i)}$'s elements, i.e., $\sum_{k=1}^{n} |\boldsymbol{e}_k^{(i)}|$.

Basic Idea. Our approach is centered around exploiting group actions' commutative property (i.e., $\mathfrak{ab} \star E = \mathfrak{a} \star (\mathfrak{b} \star E) = \mathfrak{b} \star (\mathfrak{a} \star E)$). By carefully arranging multiple actions in a specific order (i.e., a strategy), we take advantage of commutativity, leading to instances where certain group actions overlap during the evaluation process. This overlapping becomes crucial, allowing us to identify and eliminate redundant overlapping actions within the action system, reducing the number of operations. In other words, the crucial aspect of strategically constructing an action system $\{\boldsymbol{e}^{(j)}, \boldsymbol{e}^{(i)}\}$ is to ensure that $\boldsymbol{e}^{(j)}$ and $\boldsymbol{e}^{(i)}$ have a significant amount of overlap. This optimization strategy is expected to result in around $\sim 20\%$ reduction in computational cost. Additionally, it is necessary to recalculate the strategy for each new execution set during the evaluation process, which unfortunately introduces additional computational overhead. It is important to understand that the final outcome will depend on the method used to derive the strategy. To illustrate, for $\boldsymbol{e}^{(j)}$ and $\boldsymbol{e}^{(i)}$, we are looking for a common action $\boldsymbol{q}^{(ji)}$ such that $\boldsymbol{e}^{(j)} = \boldsymbol{q}^{(ji)} + \boldsymbol{r}^{(j)}$ and $\boldsymbol{e}^{(i)} = \boldsymbol{q}^{(ji)} + \boldsymbol{r}^{(i)}$. Hence, to obtain $\boldsymbol{e}^{(j)} \star E$ and $\boldsymbol{e}^{(i)} \star E$, we compute $E' = \boldsymbol{q}^{(ji)} \star E$ and then $\boldsymbol{r}^{(j)} \star E'$ and $\boldsymbol{r}^{(i)} \star E'$. The question to be answered here is how to find a common action $\boldsymbol{q}^{(ji)}$ efficiently, which can be formulated as follows.

Problem 1. Given $\{\boldsymbol{e}^{(j)}\}_{j\in[c]}$, find distinct pairs $\boldsymbol{e}^{(j)}$ and $\boldsymbol{e}^{(i)}$ with a *minimum distance*, e.g., with $\boldsymbol{q}^{(ji)}$ that minimizes the difficulty of $\boldsymbol{r}^{(j)}$ and $\boldsymbol{r}^{(i)}$.

To answer this question, one straightforward method involves finding a pair $(\boldsymbol{e}^{(j)}, \boldsymbol{e}^{(i)})$ with $\boldsymbol{q}^{(ji)}$ that has the most common values. However, such a method might not be the most efficient. Therefore, in this study, we propose using a more practical metric, referred to as M, to measure the proximity between $\boldsymbol{e}^{(j)}$ and $\boldsymbol{e}^{(i)}$. We then identify the pair with the highest M-value. Thus, the problem is defined as follows: *Find distinct* $\boldsymbol{e}^{(j)}$ *for every* $\boldsymbol{e}^{(i)}$ *in* $\{\boldsymbol{e}^{(j)}\}_{j\in[c]}$ *that maximizes* M. Here, the distance metric M quantifies the number of intersecting positive and negative (non-zero) elements in the pair, where a higher M indicating a smaller distance and greater similarity. The main advantage of this method is that it only involves determining the number of 1's in the output of an XOR of two n-bit operands, which has a low computational cost. We then select the pairs $(\boldsymbol{e}^{(j)}, \boldsymbol{e}^{(i)})$ with the highest M-value, where $i, j \in [c]$ and $i \neq j$. Given this approach, it is easy to see that the complexity is at most equivalent to $\frac{(c-1)c}{2}(1+n)$ integer additions, which constitute an overhead but for parameters used in our experiments it is negligibly small compared to other operations required for a group action. Finally, our empirical results indicate that our method offers a reasonable approximation to $\boldsymbol{q}^{(ji)}$.

Analysis of Our Strategy. Here, we discuss how we analyze the computational cost of variable-time CSIDH implementations using our strategy. In particular, and for comparability to previous work, we focus on CSIDH-512 [10].

Selected Parameters. The implementation of CSIDH-512 is defined over the finite field $\mathbb{F}_p$, where $p = 4\prod_{i=1}^{n=74} \ell_i - 1$ with ℓ_1 to ℓ_{73} being all odd primes from 3 to 373 and $\ell_{74} = 587$. Depending on the application, there are two methods to sample class elements. The first method involves sampling an element from a defined key space of size 2^{256}, as in [2,10,19]. The second method involves sampling "directly" from the ideal class group, provided the class order is known. Although our initial assessment using the first method points to a 17% to 22% computational reduction (depending on the defined key space), in this study, we will analyze our strategy using the second method, as it has been used in relevant applications that can benefit from our work, including [1,14,20]. Thus, given that $\mathbf{cl}(\mathcal{O})$ is cyclic with generator $\mathfrak{l}$ and order O, we sample the action $\mathfrak{e}_i$ in a form of $\mathfrak{l}^e$, where $e \leftarrow_\$ \mathbb{Z}_O$ and $l \in \mathcal{L}$. We then use a reduction to obtain a short[1] vector $\boldsymbol{e}^{(i)} = (\boldsymbol{e}_1^{(i)}, \ldots, \boldsymbol{e}_n^{(i)})$ by solving the closest vector problem in the lattice of relations, as described in [5,7].

Cost Metric. The symbols **M**, **S**, and **A** are used to denote the operation count for multiplication, squaring, and addition (or subtraction) on the finite field $\mathbb{F}_p$, respectively. Similar to [2], we use two metrics to reflect the total number of operations, named Metric 1 and Metric 2, defined by $\mathsf{Sum}((\mathbf{M}+\mathbf{S}+\mathbf{A}) \odot (1,1,0))$ and $\mathsf{Sum}((\mathbf{M}+\mathbf{S}+\mathbf{A}) \odot (1,0.8,0.05))$, respectively. To evaluate a group action, we use the implementation of the variable-time CSIDH-action function presented in the most recent version of CTIDH software, which is similar to the implementation in CSI-FiSh [5]. The main difference is the use of an efficient point sampler subroutine, namely Elligator [4]. For cost measurement, we use the operation counter in the CTIDH's software to obtain the count for the number of $\mathbb{F}_p$-operations in the class group action of CSIDH-512. Because the costs vary based on the execution set size, we ran an experiment on various sizes, comparing the cost of two different metrics with an action system size of two.

Experimental Results. In every experiment, the elements of execution set were randomly chosen from the key space, ensuring that any speedup reflects typical scenarios. Table 1 provides the cost for computing an execution set of size $c = 50$ with an action system of size $c' = 2$, which shows a roughly 14% reduction in computational cost in both Metric 1 and 2. Also, as depicted in Fig. 1, we present our experimental results measuring the computational costs for action system of size two of execution sets ranging in size from 2 to 58 with an increment of four.

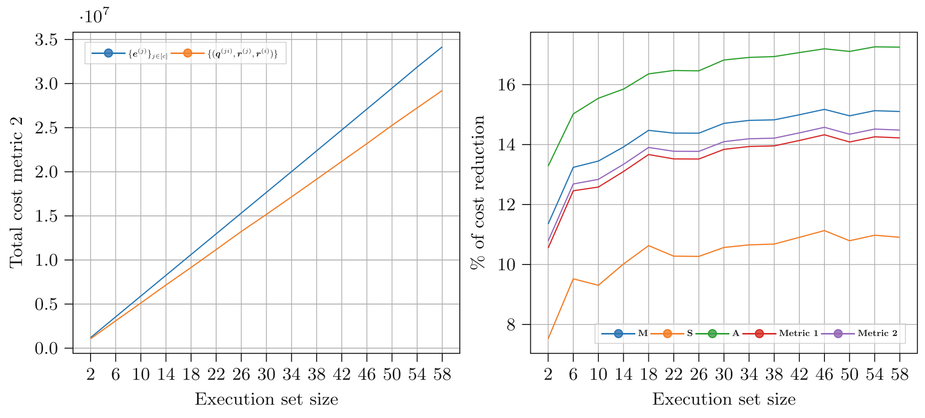
Table 1. The computational cost for computing an execution set of size 50 with an action system of size two, i.e., executing 25 action systems of size 2. The experiment was conducted using the source code from [5] and [3], with the latter being built using the makefile from its fork [22].

	M	S	A	Cost	
				Metric 1	Metric 2
Individual computation [2,3]	23,310,847	6,177,830	23,651,547	29,488,677	29,435,689
Strategic computation (this work, Sect. 3)	19,796,236	5,488,757	19,651,986	25,284,993	25,169,841
Reduction:	**15%**	**11.15%**	**16.91%**	**14.25%**	**14.49%**

4 Private Action Set Evaluation

This section presents methods for reducing the number of field arithmetic operations in the constant-time version of CSIDH, namely Algorithms 2 of CTIDH. In particular, we are concerned here with situations where the protocol requires the computation of a secret action system, such as those described in [1,6,7,15,20].

[1] A vector with small difficulty.



(a) Total cost with Metric 2. (b) Percentage of cost reduction.

Fig. 1. Summary of our results for computing an execution set of size c ranging from 2 to 58 with a CSIDH512's action system of size $c' = 2$.

Hence, we need to consider the possibility of a physical attack vector (by a malicious observer) that may be used to obtain the secret set of group actions, i.e., $\{\boldsymbol{e}^{(j)}\}_{j\in[c]}$.

Basic Idea. At a high level, our technique consists of three parts. First, we compute the torsion points T_0 and T_1 that correspond to positive and negative secret exponents with distinct orders in each execution round, in contrast to previous work, CTIDH [2] and [19]. With our proposed subroutine in Algorithm 4, GetTorsionPoint, we eliminate the inner-loop point multiplication at Line 20 in Algorithm 2. Hence, we present a new constant-time implementation of CSIDH in Algorithm 3. Secondly, we evaluate the execution set strategically by constructing action systems using an approach similar to that described in Sect. 3. This method helps reduce the number of torsion points computed and the isogeny evaluations required. Finally, since every system starts from the same elliptic curve E_A, we present a method to eliminate the redundant computation of the torsion points T_0 and T_1.

1. New constant-time implementation. Here we give a description of our constant-time implementation of CSIDH algorithm. First, we concentrate on the inner loop (Lines 7–20) of CTIDH Algorithm 2. In this algorithm, Line 9 computes the torsion points with the right order to compute all isogenies in a k-iteration. As a result, this computation is performed using constant-time xMULConstTime to conceal the secret isogeny degree. Further, at the end of each iteration, we need to push the other torsion point (i.e., T_{1-s}) to attain the required torsion degree for computing the subsequent isogeny; however, we are required to push both points when $\epsilon_i = 0$, which leaks this information. Hence, Line 20 always pushes both resulting in an extra overhead, which can be overcome by removing Line 20 and replacing Line 9 with the subroutine GetTorsionPoint as described in Algorithm 4.

Our constant-time algorithm in Algorithm 3 utilizes the GetTorsionPoint subroutine, ensuring that both points T_0 and T_1 have the right order with respect to the sign of the secret vector ϵ. Hence, we do not have to push both torsion points, which eliminates the extra overhead (i.e., the scalar multiplication at Line 20). In Algorithm 3, the subroutine RandmCoinToss takes a sign ϵ_i, batch number I_i, and position J_i, and performs a coin toss with a success probability $\alpha_{\epsilon_i,I_i,J_i}$, which will be defined later.

Algorithm 3: New constant-time group action.

```
   Parameters: $(B \in \mathbb{Z}_{>0}) \leq n, N \in \mathbb{Z}^B_{>0}, m \in \mathbb{Z}^B_{\geq 0}$
   Require: $e \in \mathcal{K}_{N,m}, E_A \in \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$
   Ensure: $A$ with $E'_A = (\prod_i \ell_i^{e_i}) \star E_A$
1  $(\mu_1, \ldots, \mu_B) \leftarrow (m_1, \ldots, m_B)$;
2  while $\mu_1, \ldots, \mu_B \neq (0, \ldots, 0)$ do
3      Let $I := \{1 \leq i \leq B : \mu_i > 0\}$ of size $k$, sorted in ascending order;
4      for $1 \leq i \leq k$ do
5          $J_i \leftarrow$ some $j$ so that $e_{i,j} \neq 0$ if such a $j$ exists, else $j \leftarrow_{\$} [N_{I_i}]$;
6          $\epsilon_i \leftarrow \mathsf{sign}(e_{I_i,J_i})$; // 1 if $e_{I_i,J_i} > 0$; 0 if $e_{I_i,J_i} = 0$; -1 o.w.;
7          $\tilde{f}_i \leftarrow \mathsf{RandmCoinToss}(\epsilon_i, I_i, J_i)$; // $\tilde{f}_i \in \{0,1\}$ is a random coin
           based on $\epsilon_i, I_i$, and $J_i$;
8      $(T_0, T_1) \leftarrow \mathsf{UniformRandomPoints}(A)$;
9      $(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ where $r = 4\prod_{i \notin I} \prod_{j=1}^{N_i} \ell_{i,j}$;
10     $(T_0, T_1) \leftarrow \mathsf{GetTorsionPoint}((T_0, T_1), \tilde{f}, \epsilon, I, J, N)$;
11     $\epsilon' \leftarrow 1$;
12     for $j = k$ to 1 do
13         $\epsilon' \leftarrow \mathsf{cmov}(\epsilon_j, \epsilon', \epsilon_j \neq 0$ and $\tilde{f}_j \neq 0)$;
14         $s \leftarrow \mathsf{SignBit}(\epsilon')$;            // 1 if $\epsilon' < 0$, o.w. 0;
15         $P \leftarrow T_s$;
16         for $i = j - 1$ to 1 do
17             $P \leftarrow \mathsf{xMULConstTime}(P, \ell_{I_i,J_i}, I_i)$;
18             if $i = j - 1$ then
19                 $T_s \leftarrow \mathsf{cmov}(P, T_s, \epsilon_i = 0$ or $\tilde{f}_i = 0)$;
20         $(f_j, P) \leftarrow \mathsf{PointAccept}(P, T_s, \epsilon_j, \tilde{f}_j)$;
21         if $f_j = 1$ then
22             $(A', (T'_0, T'_1)) \leftarrow \mathsf{MatryoshkaIsogeny}(A, P, (T_0, T_1), I_j, J_j)$;
23             $(A, T_0, T_1) \leftarrow \mathsf{cmov}((A', T'_0, T'_1), (A, T_0, T_1), \epsilon_j \neq 0)$;
24         if $j = k$ then
25             $T_0 \leftarrow \mathsf{xMULConstTime}(T_0, \ell_{I_j,J_j}, I_j)$;
26     for $1 \leq i \leq k$ do
27         $(\mu_{I_i}, e_{I_i,J_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i,J_i} - f_i\epsilon_i)$;
28 return $A$;
```

Algorithm 4: Torsion point computation, GetTorsionPoint.

```
Input: $(T_0, T_1)$, $\tilde{f} \in \{0,1\}^k$, $\epsilon \in \{-1,0,1\}^k$, $I \in \mathbb{Z}_{\geq 0}^{k \leq B}$, $J \in \mathbb{Z}_{>0}^k$, $N \in \mathbb{Z}_{>0}^B$
Output: Updated $T_0$ and $T_1$
1  $\epsilon' \leftarrow 1$;
2  for $i = k$ to $1$ do
3      ClearSet $\leftarrow \{1, \ldots, N_{I_i}\}$;
4      $\epsilon' \leftarrow$ cmov$(\epsilon_i, \epsilon', \epsilon_i \neq 0$ and $\tilde{f}_i \neq 0)$;
5      for $j \in$ ClearSet do
6          if $\epsilon' = 1$ then
7              $T_1 \leftarrow$ xMUL$(T_1, \ell_{I_i,j})$;
8          else if $\epsilon' = -1$ then
9              $T_0 \leftarrow$ xMUL$(T_0, \ell_{I_i,j})$;
10
11     $j_{target} \leftarrow J_i$;
12     ClearSet $\leftarrow$ ClearSet $\setminus \{j_{target}\}$;
13     for $j \in$ ClearSet do
14         if $\epsilon' = 1$ then
15             $T_0 \leftarrow$ xMULConstTime$(T_0, \ell_{I_i,j}, I_i)$;
16         else if $\epsilon' = -1$ then
17             $T_1 \leftarrow$ xMULConstTime$(T_1, \ell_{I_i,j}, I_i)$;
18
19 return $(T_0, T_1)$;
```

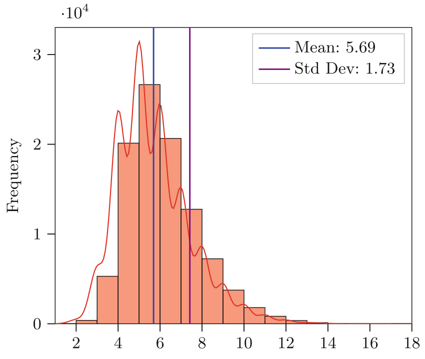
Implementation Reasoning and Security. In Algorithm 3, we compute one isogeny per batch in each round (Lines 12–25) using the torsion point with the correct order from the subroutine GetTorsionPoint. In GetTorsionPoint, the first for-loop operation (Lines 5–10 in Algorithm 4) clears the order of the point with the opposite sign by performing scalar multiplication for all primes within the batch for all target batches I. This process is independent of the secret sign, allowing the use of the inexpensive scalar multiplication xMUL instead of xMULConstTime. The second for-loop performs, for each batch in I, a scalar multiplication by $\prod_{i \in I} \prod_{j \in \mathsf{ClearSet} \setminus \{J_i\}} \ell_{i,j}$. Therefore, we use xMULConstTime to conceal ℓ_{i,J_i} that represents the selected secret isogeny. In this case, the number of multiplications is $\sum_{i \in I} N_i - 1$; therefore, the number of multiplications is independent of the secret ϵ_i and rather depends on the vector I, which is public information. Additionally, when $\epsilon_i = 0$, the subroutine GetTorsionPoint performs the multiplication using the sign of the previous non-zero ϵ'.

Kernel Point Failure. In Algorithm 3, Lines 16–17 produce a possible kernel generator, denoted as P. With probability $1 - 1/\ell_{I_j,J_j}$, P is a kernel of order ℓ_{I_j,J_j}, indicated by $f'_j = 1$, where ℓ_{I_j,J_j} is related to J_j, the secret isogeny degree. Conversely, with probability $1/\ell_{I_j,J_j}$, P is likely a point at infinity, indicated by $f'_j = 0$. As pointed out in [2], the value of f'_j depends on J_j. Therefore, as in [2], we deliberately increase the failure probability from $1/\ell_{I_j,J_j}$ to $1/\ell_{I_j,1}$, with $\ell_{I_j,1}$ being the first prime in the batch $(\ell_{I_j,1}, \ldots, \ell_{I_j,N_{I_j}})$. To achieve this, we use a biased coin flip variable $\tilde{f}_j$ with a success probability $\alpha_{\epsilon_j,I_j,J_j}$ (utilizing $\mathsf{RandmCoinToss}$). We then set f_j given both f'_j and $\tilde{f}_j$ (employing $\mathsf{PointAccept}$). Specifically, for $\epsilon_j \neq 0$, the subroutine $\mathsf{PointAccept}$ yields $(f_j,\, P)$ with $f_j = 1$ exclusively when P is not at infinity *and* the coin flip $\tilde{f}_j$ succeeds with probability $\alpha_{\epsilon_j,I_j,J_j} = \frac{\ell_{I_j,J_j}(\ell_{I_j,1}-1)}{\ell_{I_j,1}(\ell_{I_j,J_j}-1)}$. In instances where $\epsilon_j = 0$, it returns $(f_j,\, T_s)$ with f_j being based solely on the coin flip $\tilde{f}_j$, with probability $\alpha_{0,I_i,J_i} = 1 - 1/\ell_{I_j,1}$. As a result, the value of f_j is rendered independent of both J_j and ϵ_j.

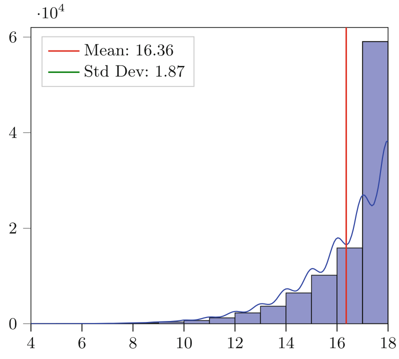
MatryoshkaIsogeny. As in CTIDH, $\mathsf{MatryoshkaIsogeny}$ performs a fixed number of operations based only on input $i \in I$, regardless of the input point order. In Algorithm 3, we thus feed a dummy input point to $\mathsf{MatryoshkaIsogeny}$ for secret with a value of zero ($\epsilon_j = 0$) using $\mathsf{PointAccept}$, so that the computational cost remains the same regardless of ϵ_j's value.

2. Strategic computation of a private action system. In this section, we propose a constant-time strategy for evaluating a private set of actions, which is a variant to the strategy in Sect. 3. In this discussion, we define a *round of execution* as a single iteration that evaluates a group action, involving the computation of k-isogenies for the targeted batches I, as specified in Algorithm 3, Lines 12–25. To fully compute an action, it requires at least $\mathsf{max}(m_1, \ldots, m_B)$-rounds. To compute a private set of actions, we use our constant-time strategy to eliminate d rounds of execution, referred to as d-common round(s) $\boldsymbol{q}^{(ij)}$. When constructing the action system, the common round must execute a common (overlapped) action set *in all targeted batches*. This is essential to ensure that the execution of the common round involves a fixed number of isogeny computations, which is equal to the number of targeted batches, regardless of the overlapped action $\boldsymbol{q}^{(ij)}$. To fully utilize the strategy, it is essential that the d-common round(s) include d overlapping actions within each of the targeted batches (i.e., the difficulty of $\boldsymbol{q}^{(ij)} = \#$ of target batches $\times d$). This contrasts with the approach outlined in Sect. 3, which may not necessarily require d overlapping actions within each of the targeted batches. In other words, for each batch that does not overlap, we will be forced to use a dummy isogeny, not only negating the gain we are seeking but also possibly resulting in incorrect output. Fortunately, due to the distribution of keys in each batch's key space, overlapping is highly likely to occur, particularly in moderate size execution sets and with $d = 1$. For example, in batches 3 through 13 of CTIDH-512, the bulk of keys in the key space have a batch sum close to the batch sum bound m_i, and having the largest element less than half of the batch sum m_i, as shown in Fig. 2. Hence, the vast majority of keys have a flat spectrum of small values, maximizing the likelihood of common round overlaps across batches within a key set.

In this work, we employ a single common round of execution (i.e., $d = 1$ with the difficulty of $\boldsymbol{q}^{(ij)} = \#$ of target batches) for targeted batches between 3 to 13 of CSIDH-512. However, and for further computational reduction, two or more common rounds can be utilized in hybrid form. For example, for an execution set of size c, 20% of the action systems can be executed with two common rounds, the remaining 80% with one round, and so forth.



(a) The distribution of the maximum absolute value.



(b) The distribution of the sum of secret elements $(e_{11,1}, \ldots, e_{11,8})$, such that $m_{11} = 18$.

Fig. 2. The distribution of the sum of secret elements $(e_{11,1}, \ldots, e_{11,8})$ and the distribution of the maximum absolute value in the secret vector (i.e., $\mathsf{max}(|e_{11,1}|, \ldots, |e_{11,8}|)$) for Batch 11 over 100 thousands random distinct keys.

In addition, the efficiency of our strategy depends critically on the torsion point being full order in the common execution round, since a torsion point that is not of full order will prevent us from fully executing the common round. There are two ways to avoid this issue: i) select a targeted overlapped-batches with a low failure probability, such as batches with larger primes, and ii) use a deterministic seed for the UniformRandomPoints subroutine (in the first round of the execution) that guarantees full-torsion points to be generated.

Elligator. As in the previous works [2,17,19], we use Elligator : $E_A \rightarrow T_{0/1}$ [4] in Algorithm 3 to construct UniformRandomPoints. Also, and given E_A, the Elligator initially can be executed using a deterministic seed, stored as public parameters, resulting in full-order points, i.e., T_0 and T_1 both have order $(p+1)/4$.

3. Reducing torsion point computation. To evaluate group actions (e.g., $\boldsymbol{q}^{(ij)}$), a torsion point of a specific degree must be sampled for each execution round. Duplicate point sampling and multiplication can be eliminated from redundant torsion-point computations, as some or all action systems in the execution set share the same initial E_A. Accordingly, this technique is applied in the first round of execution or whenever we have actions starting from the same E_A. Specifically, we adjust our torsion point computation to determine two pairs of torsion points–one pair for each corresponding action, as detailed in Algorithm 5. The latter includes a subroutine shown in Algorithm 6, which is identical to Algorithm 4 except that it uses xMULConstTime instead of xMUL at lines 10–11.

Analysis of Our Strategy. In this section, we analyze the computational cost of constant-time CSIDH implemented using our strategy.

Selected Parameters. In order to be able to provide a comparative analysis based on computational cost, we use parameters similar to those of CTIDH. In particular, we consider CSIDH-512 [10], which is characterized by the finite field $\mathbb{F}_p$, where $p = 4\prod_{i=1}^{74} \ell_i - 1$ with ℓ_1 to ℓ_{73} being all odd primes from 3 to 373 and $\ell_{74} = 587$. For the secret key space, we consider CTIDH-512's proposal $\mathcal{K}_{N,m}$ introduced in [2], which requires CTIDH 208 odd-isogeny evaluations and a minimum of 18 torsion point samplings and computations. Further, our work does not make use of the improved version of CTIDH, namely SECSIDH [8], since the latter is focused on improvements based on optimization of field arithmetic (e.g., using the Karatsuba algorithm for field multiplication) and employs the same algorithm as CTIDH, whereas we aim to reduce the *number* of field arithmetic operations. In this work, we modify CTIDH's algorithms in combination with our previous computation strategy, both of which are independent of field-arithmetic level optimizations. Furthermore, SECSIDH works on CSIDH with very large parameters from CSIDH-2048 to CSIDH-9216. While their techniques are more effective with these parameters, they appear to have limited practical applications, as the performance of the applications in question heavily relies on knowing the order of the ideal class $\mathbf{cl}(\mathcal{O})$.

Cost Metric. To evaluate the action systems, we construct our algorithms using the constant-time CTIDH functions presented in the most recent version of the **high-ctidh** software [3] built with the makefile from its fork [22], including Elligator, xMULConstTime, and MatryoshkaIsogeny. For cost measurement, we

Algorithm 5: Initial round torsion point computation

```
Input: $E_A \in \mathcal{E}_{\mathbb{F}_p}(\mathcal{O})$, $(\tilde{f}^1, \tilde{f}^2) \in \{0,1\}^{2k}$, $(\epsilon^1, \epsilon^2) \in \{-1,0,1\}^{2k}$, $I \in \mathbb{Z}_{\geq 0}^{k \leq B}$, $(J^1, J^2) \in \mathbb{Z}_{>0}^{2k}$
Output: $(T_0^1, T_1^1), (T_0^2, T_1^2)$
1  $(T_0, T_1) \leftarrow$ UniformRandomPoints$(A)$
2  $(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ where $r = 4\prod_{i \notin I}\prod_{j=1}^{N_i} \ell_{i,j}$
3  CommonTargets $\leftarrow$ list of $B$-empty lists
4  for $i = k$ to $1$ do
5      $j' \leftarrow_{\$} \{1, \ldots, N_{I_i}\} \setminus \{J_i^1, J_i^2\}$
6      CommonTargets$_i \leftarrow$ cmov$((J_i^1, J_i^2), (J_i^1, j'), J_i^1 \neq J_i^2)$
7      CommonTargets$_i \leftarrow$ cmov$([N_{I_i}]$, CommonTargets$_i, N_{I_i} \leq 2)$
8      ClearSet $\leftarrow \{1, \ldots, N_{I_i}\} \setminus \{$CommonTargets$_i\}$
9      for $j \in$ ClearSet do
10         $T_0 \leftarrow$ xMULConstTime$(T_0, \ell_{I_i,j}, I_i)$
11         $T_1 \leftarrow$ xMULConstTime$(T_1, \ell_{I_i,j}, I_i)$
12 $(T_0^1, T_1^1) \leftarrow$ GetTorsionPoint2$((T_0, T_1), \tilde{f}^1, \epsilon^1, I, J^1,$ CommonTargets$)$
13 $(T_0^2, T_1^2) \leftarrow$ GetTorsionPoint2$((T_0, T_1), \tilde{f}^2, \epsilon^2, I, J^2,$ CommonTargets$)$
14 return $(T_0^1, T_1^1), (T_0^2, T_1^2)$
```

Algorithm 6: GetTorsionPoint2 (revised from Algorithm 4)

```
Input: $(T_0, T_1)$, $\tilde{f} \in \{0,1\}^k$, $\epsilon \in \{-1,0,1\}^k$, $I \in \mathbb{Z}_{\geq 0}^{k \leq B}$, $J \in \mathbb{Z}_{>0}^k$,
       $\mathsf{ClearSets} \in \mathbb{Z}_{>0}^k$
Output: Updated $T_0$ and $T_1$
1  $\epsilon' \leftarrow 1$;
2  for $i = k$ to 1 do
3      $\mathsf{ClearSet} \leftarrow \mathsf{ClearSets}_i$;
4      $\epsilon' \leftarrow \mathsf{cmov}(\epsilon_i, \epsilon', \epsilon_i \neq 0 \text{ and } \tilde{f}_i \neq 0)$;
5      for $j \in \mathsf{ClearSet}$ do
6          if $\epsilon' = 1$ then
7              $T_1 \leftarrow \mathsf{xMULConstTime}(T_1, \ell_{I_i,j}, I_i)$;
8          else if $\epsilon' = -1$ then
9              $T_0 \leftarrow \mathsf{xMULConstTime}(T_0, \ell_{I_i,j}, I_i)$;
10
11     $\mathsf{ClearSet} \leftarrow \mathsf{ClearSet} \setminus \{J_i\}$;
12     for $j \in \mathsf{ClearSet}$ do
13         if $\epsilon' = 1$ then
14             $T_0 \leftarrow \mathsf{xMULConstTime}(T_0, \ell_{I_i,j}, I_i)$;
15         else if $\epsilon' = -1$ then
16             $T_1 \leftarrow \mathsf{xMULConstTime}(T_1, \ell_{I_i,j}, I_i)$;
17
18 return $(T_0, T_1)$;
```

use the arithmetic operation counter in the **high-ctidh** software to obtain the count for the number of operations on $\mathbb{F}_p$ evaluating the execution set.

Table 2. The computational cost for computing 2 action systems of size 2, i.e., $\{\{\boldsymbol{e}^{(1)}, \boldsymbol{e}^{(2)}\}, \{\boldsymbol{e}^{(3)}, \boldsymbol{e}^{(4)}\}\}$. The experiment was conducted using the source code from [2], which is built on the makefile from its fork [22].

	M	S	A	Cost	
				Metric 1	Metric 2
Individual computation [2,3]	1,788,198	468,887	2,001,842	2,257,086	2,263,400
Strategic computation (this work, Sect. 4)	1,676,746	400,398	1,954,800	2,077,144	2,094,805
Reduction:	**6.23%**	**14.60%**	**2.35%**	**7.97%**	**7.45%**

Results. Our results are compared with individual computations of the execution set carried out using group action implemented in **high-ctidh** software [3,22]. We initially benchmarked our constant-time Algorithm 3 (and its subroutine Algorithm 4) alone, without our strategic method, against the state-of-the-art CTIDH algorithm [3,22]. Our results show a reduction of 2.75% for field multiplication, 11.08% for squaring, 4.48% for Metric 1, and 3.96% for Metric

2; however, we observe a 1.12% increase in field addition. Furthermore, Table 2 illustrates roughly up to 8% reduction in computational cost for an execution set of size 4 with an action system of size 2.

5 Conclusion

In this work, we have proposed multiple techniques of practical importance to reduce the number of field arithmetic operations required to evaluate ideal class group actions on a set element. Our first proposal is a batching strategy to efficiently evaluate an action system. We have analyzed our strategy for action set sizes ranging between 2 to 58, where it shows a reduction of around 14% of field arithmetic operations. Secondly, we have proposed an improved constant-time algorithm that is merged with our batch strategy to evaluate a private set of action system. Our analysis indicates an improvement of roughly up to 8%, with the improved constant-time CSIDH implementation alone reducing computation by around 4%.

References

1. Atapoor, S., Baghery, K., Cozzo, D., Pedersen, R.: CSI-SharK: CSI-FiSh with sharing-friendly keys. In: Simpson, L., Baee, M.A.R. (eds.) ACISP 2023. LNCS, vol. 13915, pp. 471–502. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35486-1_21
2. Banegas, G., et al.: CTIDH: faster constant-time CSIDH. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 351–387 (2021). https://doi.org/10.46586/tches.v2021.i4.351-387
3. Banegas, G., et al.: CTIDH: faster constant-time CSIDH, software (2024). https://ctidh.isogeny.org/software.html
4. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 967–980. ACM (2013). https://doi.org/10.1145/2508859.2516734
5. Beullens, W.: CSI-FiSh: github repository. https://github.com/KULeuven-COSIC/CSI-FiSh/
6. Beullens, W., Disson, L., Pedersen, R., Vercauteren, F.: CSI-RAShi: distributed key generation for CSIDH. In: Cheon, J.H., Tillich, J.-P. (eds.) PQCrypto 2021 2021. LNCS, vol. 12841, pp. 257–276. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81293-5_14
7. Beullens, W., Kleinjung, T., Vercauteren, F.: CSI-FiSh: efficient isogeny based signatures through class group computations. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part I. LNCS, vol. 11921, pp. 227–247. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34578-5_9
8. Campos, F., et al.: On the practicality of post-quantum TLS using large-parameter CSIDH. IACR Cryptology ePrint Archive, p. 793 (2023). https://eprint.iacr.org/2023/793

9. Castryck, W., Decru, T.: CSIDH on the surface. In: Ding, J., Tillich, J.-P. (eds.) PQCrypto 2020. LNCS, vol. 12100, pp. 111–129. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44223-1_7
10. Castryck, W., Lange, T., Martindale, C., Panny, L., Renes, J.: CSIDH: an efficient post-quantum commutative group action. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 395–427. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_15
11. Cervantes-Vázquez, D., Chenu, M., Chi-Domínguez, J.-J., De Feo, L., Rodríguez-Henríquez, F., Smith, B.: Stronger and faster side-channel protections for CSIDH. In: Schwabe, P., Thériault, N. (eds.) LATINCRYPT 2019. LNCS, vol. 11774, pp. 173–193. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30530-7_9
12. Chávez-Saab, J., Chi-Domínguez, J., Jaques, S., Rodríguez-Henríquez, F.: The SQALE of CSIDH: sublinear Vélu quantum-resistant isogeny action with low exponents. J. Cryptogr. Eng. **12**(3), 349–368 (2022). https://doi.org/10.1007/S13389-021-00271-W
13. Cheng, H., Fotiadis, G., Großschädl, J., Ryan, P.Y.A., Rønne, P.B.: Batching CSIDH group actions using AVX-512. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 618–649 (2021). https://doi.org/10.46586/tches.v2021.i4.618-649
14. Cozzo, D., Smart, N.P.: Sashimi: cutting up CSI-FiSh secret keys to produce an actively secure distributed signing protocol. In: Ding, J., Tillich, J.-P. (eds.) PQCrypto 2020. LNCS, vol. 12100, pp. 169–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44223-1_10
15. De Feo, L., Galbraith, S.D.: SeaSign: compact isogeny signatures from class group actions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 759–789. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_26
16. De Feo, L., Meyer, M.: Threshold schemes from isogeny assumptions. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) PKC 2020. LNCS, vol. 12111, pp. 187–212. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45388-6_7
17. Meyer, M., Campos, F., Reith, S.: On lions and elligators: an efficient constant-time implementation of CSIDH. In: Ding, J., Steinwandt, R. (eds.) PQCrypto 2019. LNCS, vol. 11505, pp. 307–325. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25510-7_17
18. Meyer, M., Reith, S.: A faster way to the CSIDH. In: Chakraborty, D., Iwata, T. (eds.) INDOCRYPT 2018. LNCS, vol. 11356, pp. 137–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05378-9_8
19. Onuki, H., Aikawa, Y., Yamazaki, T., Takagi, T.: (Short paper) a faster constant-time algorithm of CSIDH keeping two points. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 2019. LNCS, vol. 11689, pp. 23–33. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26834-3_2
20. de Saint Guilhem, C.D., Pedersen, R.: New proof systems and an OPRF from CSIDH. In: Tang, Q., Teague, V. (eds.) PKC 2024. LNCS, vol. 14603, pp. 217–251. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57725-3_8
21. Silverman, J.H.: Advanced Topics in the Arithmetic of Elliptic Curves. Springer, New York (1994). https://doi.org/10.1007/978-1-4612-0851-8
22. Stainton, D., Appelbaum, J.: high-ctidh (Version fix_private_constructor_rng). https://git.xx.network/elixxir/high-ctidh/
23. Stewart, I., Tall, D.: Algebraic Number Theory and Fermat's Last Theorem. CRC Press, Boca Raton (2015)
24. Vélu, J.: Isogénies entre courbes elliptiques. CR Acad. Sci. Paris, Séries A **273**, 305–347 (1971)